Vérification formelle de conditions d'ordonnançabilité de tâches temps réel périodiques strictes

Daniel de Rauglaudre[1]

1: Inria Paris - Rocquencourt
Domaine de Voluceau - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex France
daniel.de_rauglaudre@inria.fr

Résumé

Nous formalisons en Coq le problème de l'ordonnancement de tâches périodiques strictes non préemptives et prouvons formellement le théorème dû à Jan Korst [1] donnant la condition nécessaire et suffisante pour l'ordonnançabilité de telles tâches.

Introduction

Dans le domaine de l'ordonnancement de tâches temps réel, typiquement dans les systèmes embarqués, se pose le problème de l'ordonnancement de tâches périodiques. Des modèles mathématiques de tels problèmes [2] [3] existent mais n'ont jamais, jusqu'ici, été transcrites dans un assistant de preuve tel que Coq. Il n'y a donc aucun "état de l'art" sur la question.

Il nous a donc paru intéressant de voir si ces modèles mathématiques pouvaient être prouvés formellement. C'est d'autant plus important que les systèmes embarqués sont souvent soumis à des contraintes de sécurité (risque de pertes humaines) et qu'il est préférable de disposer de bases solides, c'est-à-dire de théorèmes vérifiés formellement, pour le développement des logiciels embarqués.

Dans ce cadre, nous avons choisi, pour démarrer, un théorème le plus simple possible dans le contexte relativement simple des tâches non préemptives, à périodicité stricte et sur un seul processeur.

Cet article décrit une transcription en Coq [4] de ce théorème et de deux de ses corollaires. Les deux premières parties décrivent le contexte et les définitions utilisées. La troisième partie présente le théorème. La quatrième partie indique comment le modèle d'ordonnancement et la preuve du théorème ont été écrits en Coq.

1. Contexte

Nous nous intéressons à l'ordonnancement d'un système de tâches sur un seul processeur. Le temps est discret, c'est-à-dire qu'il est décrit par un entier naturel. Les tâches sont indépendantes : elles ne communiquent pas et n'ont pas de contraintes de dépendance les unes par rapport aux autres.

Il s'agit de tâches périodiques, chacune ayant une période donnée constante et, à l'intérieur de cette période, devant exécuter un certain *travail* qui consomme un certain nombre d'unités de temps du processeur, toujours le même pour une tâche donnée.

L'algorithme d'ordonnancement, sur lequel est basé le théorème est dit *non préemptif* (on ne peut pas suspendre une tâche tant qu'elle n'a pas fini l'exécution de son travail) et à *périodicité stricte* (son travail périodique doit impérativement démarrer au début de chacune de ses périodes). C'est un algorithme sans aucun choix. Cette hypothèse sur le type d'ordonnanceur considéré dans cet article est réaliste pour les systèmes temps réel, notamment lors de l'acquisition des données et de la commande du système controlé.

On ne s'intéresse pas ici à ce que font ces tâches. L'important est qu'une tâche monopolise le processeur pendant la durée de son travail.

2. Définitions

Tâche

Une tâche est définie par trois entiers naturels :

- sa phase ϕ, c'est-à-dire sa première date de démarrage,
- sa durée d (non nulle),
- sa période P (non nulle).

Avant sa phase ϕ, la tâche n'utilise pas le processeur. Ensuite, elle démarre un premier travail pendant la durée d, en consommant donc d unités de temps du processeur et répète ce travail avec la périodicité P.

Système de Tâches

Un système de tâches est une liste de tâches, éventuellement vide.

Périodicité stricte

Un système de tâches est ordonnançable par périodicité stricte si chaque tâche peut exécuter la totalité de la durée de son travail à chacune de ses périodes et sans aucun délai.

L'algorithme non préemptif de périodicité stricte construit cet ordonnancement et peut donc échouer si deux tâches demandent l'accès au processeur en même temps à une certaine date.

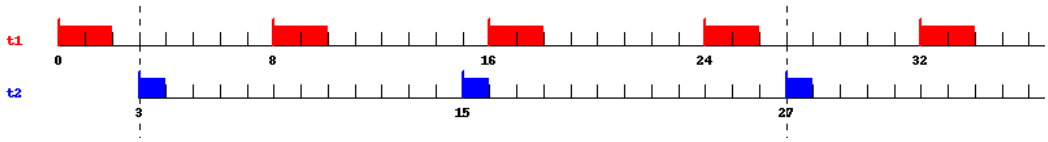

FIGURE 1 – Exemple : $(\phi_1 = 0, d_1 = 2, P_1 = 8)$ $(\phi_2 = 3, d_2 = 1, P_2 = 12)$

3. Théorème

Le théorème, dû à Jan Korst[1], que nous avons traduit en Coq s'énonce ainsi :

Théorème 1. *Un système de tâches est ordonnançable par périodicité stricte si et seulement si, pour chaque paire de tâches τ_i et τ_j du système, de paramètres respectifs (ϕ_i, d_i, P_i) et (ϕ_j, d_j, P_j), telles que $\phi_i \leq \phi_j$ et $g = pgcd\ (P_i, P_j)$, on a :*

$$d_i \leq (\phi_j - \phi_i)\ mod\ g \leq g - d_j \tag{1}$$

Démonstration. La preuve de Jan Korst consiste à supposer que $\phi_i = 0$ (en décalant éventuellement les tâches vers la gauche de ϕ_i) et à regarder les intervalles $[kg, kg+g-1]$ pour $k \in \mathbb{Z}$. Pour la condition suffisante, on constate que les d_i premières unités de temps de cet intervalle peuvent être consacrées à l'exécution de la tâche τ_i et les $g - d_i$ suivantes à la tâche τ_j. Si la condition (1) tient, alors la taille des unités de temps consacrées à chaque tâche est suffisante et le système est ordonnançable.

Pour la condition nécessaire, on raisonne par contraposée. On considère à nouveau les intervalles $[kg, kg + g - 1]$ pour $k \in \mathbb{Z}$. Si la condition (1) ne tient pas, alors l'exécution de τ_j recouvre les premières unités de temps consacrées à τ_i tous les P_j/g intervalles de temps. □

En Coq, il faut être plus précis et entrer davantage dans les détails. Par exemple, pour la réciproque, il faudra donner quels intervalles de temps précisément sont concernés, c'est-à-dire quelles sont les valeurs de k pour lesquelles il y a conflit entre les tâches et sur quelles unités de temps ; pour cela, il nous faudra utiliser le théorème de Bachet-Bézout, ce qui n'est pas explicite dans la preuve de Jan Korst.

Remarque. L'expression (1) impose en particulier que :

$$d_i + d_j \leq pgcd\ (P_i,\ P_j)$$

Autrement dit, si deux tâches ont des périodes premières entre elles, donc de PGCD égal à 1, le système ne peut pas être ordonnançable, chaque durée d'exécution valant au moins 1 (entiers non nuls).

4. Preuve formelle du théorème

4.1. Modélisation du problème de l'ordonnancement

Pour les entiers, nous avons choisi d'utiliser le type prédéfini `nat` de Coq, dits "entiers de Peano", qui représente les entiers naturels, constitué du constructeur `O` (lettre "o" majuscule) qui représente `0` (nombre zéro) et du constructeur `S` paramétré par un `nat` et qui représente le successeur du `nat` en question (autrement dit, `S(n)` correspond à `n+1`) :

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

Pour la plupart des théorèmes, nous avons utilisé la bibliothèque standard de Coq. Nous avons ajouté le PGCD sur les `nat` qui n'y est pas, et redéfini la division euclidienne sous forme de calcul, plutôt que d'utiliser celle de la bibliothèque standard qui l'est sous forme de définition inductive, un peu moins pratique à utiliser à notre goût. Il est possible que notre code gagnerait à utiliser d'autres bibliothèques plus complètes sur les entiers[5], mais cela supposerait un long travail de refonte des preuves.

L'ordonnancement est une liste d'unités de temps. Une unité de temps est une valeur du type `time_unit`. Le constructeur `E` indique que le processeur est inactif, tandis que `A(i)` qu'il exécute la tâche i :

```
Inductive time_unit := E : time_unit | A : nat → time_unit.
```

Une tâche est définie par le type `Task` qui est un enregistrement dont les champs sont ceux définis à la section 2. Le champ `id` y est ajouté, pour identifier les tâches quand on teste l'algorithme (mais il n'est pas utilisé dans les preuves) :

```
Record Task :=
  mkTask { id : nat; phase : nat; duration : nat; period : nat }.
```

Un ordonnancement est une liste infinie de valeurs de type `time_unit`. L'algorithme de périodicité stricte consiste à ordonnancer chaque tâche indépendamment, comme si elle était seule sur le processeur, et à fusionner les ordonnancements obtenus, avec échec si deux ordonnancements contiennent le constructeur `A` à une même date.

Pour l'ordonnancement d'une tâche, nous avons défini la fonction `nth_time_unit` qui, pour une date donnée `t`, renvoie la valeur de l'unité de temps à cette date :

```
Definition nth_time_unit t task : time_unit ::=
  if lt_dec t (phase task) then E
  else
    let t' := (t - phase task) mod period task in
    if lt_dec t' (duration task) then A (id task) else E.
```

La fusion des ordonnancements est faite par la fonction `merge_unit_list` qui renvoie la valeur de l'unité de temps à la date `t` donnée en paramètre, ou peut échouer, en renvoyant `None` si les tâches sont en conflit à cette date :

```
Fixpoint merge_unit_list t tasks : option time_unit :=
  match tasks with
  | [] => Some E
  | [task ... tl] =>
      match nth_time_unit t task with
      | E => merge_unit_list t tl
      | A k =>
          match merge_unit_list t tl with
          | None => None
          | Some E => Some (A k)
          | Some (A _) => None
          end
      end
  end.
```

La fonction `schedulable` dit si une liste de tâches est ordonnançable en demandant qu'à chaque date `t`, il n'y ait aucun conflit entre les tâches :

```
Definition schedulable tasks : Prop :=
  ∀ t, merge_unit_list t tasks ≠ None.
```

La fonction `schedule` crée un ordonnancement qui est une liste infinie (*Stream* en Coq) d'unités de temps :

```
Definition schedule tl :=
  let f :=
    cofix loop t :=
```

```
      match merge_unit_list t tl with
      | None => Streams.const E
      | Some u => Streams.Cons u (loop (S i))
      end in
  f 0.
```

4.2. Test de l'algorithme d'ordonnancement

L'algorithme d'ordonnancement peut être testé en interactif, grâce à la commande d'évaluation `Eval compute` de Coq. Pour l'affichage, nous avons défini la fonction `take` qui renvoie la liste des `n` premiers éléments d'un `Stream`. Voici un exemple d'utilisation :

```
Eval compute in take 30
  (schedule [mkTask 1 0 1 4, mkTask 2 1 1 6, mkTask 3 2 1 4 ... []]).
```

4.3. Correction de l'algorithme d'ordonnancement

La correction de l'algorithme d'ordonnancement n'a pas été faite dans le cadre de ce travail. Il y aurait trois théorèmes à écrire (probablement liés les uns aux autres) :

1. Un théorème pour vérifier que l'algorithme construit bien un *ordonnancement* correct, c'est-à-dire que pour chaque tâche de durée `d`, l'ordonnancement : a/ ne contient pas d'unités de temps pour cette tâche avant sa phase, et que b/ après sa phase, elle contient `d` unités de temps entre un début de période et le début de la période suivante.
2. Un théorème pour vérifier que l'algorithme est bien *non préemptif*, c'est-à-dire que les exécutions d'une tâche de durée `d` sont constituées de `d` unités de temps contiguës.
3. Un théorème pour vérifier que l'algorithme est bien de *périodicité stricte*, c'est-à-dire que les exécutions de chaque tâche démarrent exactement au début de chacune de ses périodes.

Il est un peu délicat d'évaluer la difficulté de telles preuves, compte-tenu du fait que les preuves formelles soulèvent souvent des difficultés inattendues, même quand elles paraissent a priori faciles. Nous pensons qu'il faudrait entre une journée et trois semaines pour y parvenir, selon les difficultés rencontrées.

4.4. Preuve du sens direct du théorème en Coq

4.4.1. Énoncé

Dans le langage de Coq, le théorème est énoncé ainsi :

```
Theorem Jan_Korst : ∀ tasks : list Task,
  (∀ τ, τ ∈ tasks → 0 < duration τ ≤ period τ)
  → List.ForallOrdPairs korst_test_two_tasks tasks
    → schedulable tasks.
```

avec :

```
Definition korst_formula τ1 τ2 :=
  let g := gcd (period τ1) (period τ2) in
  duration τ1 + duration τ2 ≤ (phase τ2 - phase τ1) mod g + duration τ2 ≤ g.
```

```
Definition korst_test_two_tasks τ1 τ2 :=
  phase τ2 < phase τ1 / korst_formula τ2 τ1
  phase τ1 < phase τ2 / korst_formula τ1 τ2.
```

Démonstration. La preuve est faite par contradiction. Nous avons dû prouver préalablement trois lemmes :

Lemme 1. *Si une tâche de phase ϕ, de durée d et de période P occupe le processeur à la date t, alors t est après ϕ et est à moins de d d'une période de la tâche, c'est-à-dire :*

$t \geq \phi \wedge (t - \phi) \bmod P < d$

Démonstration. Par définition de l'ordonnancement d'une tâche. □

Lemme 2. *Quels que soient a b c et k, entiers naturels (k non nul) :*

Si $a \bmod (k.b) < c$, alors $a \bmod b < c$

Démonstration. Par induction sur b et définition du modulo. □

Lemme 3. *Quels que soient a b c, entiers naturels :*

Si $a \bmod c + b \bmod c < c$, alors
$(a + b) \bmod c = a \bmod c + b \bmod c$

Démonstration. Par utilisation d'autres lemmes plus élémentaires sur le modulo dont nous ne donnons pas le détail ici. □

Pour notre théorème, après élimination des cas simples (liste vide, tâches qui ne sont pas en conflit) qui se traitent par deux inductions sur la liste des tâches, l'essentiel de la preuve consiste à démontrer qu'il y a contradiction si les hypothèses sont vérifiées mais que, cependant, deux tâches τ_i et τ_j accèdent au processeur à la même date t. Ce dernier cas se démontre par des combinaisons sur les opérations élémentaires, les modulos et les comparaisons dont voici le détail :

La preuve étant symétrique entre τ_i et τ_j, nous supposerons, sans perdre en généralité, que $\phi_i \leq \phi_j$. Dans le code, cela consiste à appeler les lemmes avec les paramètres d'un un ordre ou l'autre, suivant le sens de l'inégalité.

On commence par appliquer le lemme 1 sur nos deux tâches, ce qui nous donne :

$$t \geq \phi_i \ \wedge \ (t - \phi_i) \bmod P_i \ < \ d_i$$
$$t \geq \phi_j \ \wedge \ (t - \phi_j) \bmod P_j \ < \ d_j$$

Les nombres étant des multiples de leur PGCD (par définition du PGCD), et en notant g le PGCD de P_i et de P_j, on peut remplacer P_i par $k_i\ g$ et P_j par $k_j\ g$:

$$(t - \phi_i) \bmod (k_i.g) \ < \ d_i$$
$$(t - \phi_j) \bmod (k_j.g) \ < \ d_j$$

Le lemme 2 nous permet de supprimer k_i et k_j de nos expressions en ne gardant que le modulo g pour les deux :

$$(t - \phi_i) \bmod g \ < \ d_i \tag{2}$$
$$(t - \phi_j) \bmod g \ < \ d_j \tag{3}$$

On transforme l'expression (2) en :

$$((t - \phi_j) + (\phi_j - \phi_i)) \ mod \ g \ < \ d_i \tag{4}$$

On peut appliquer le lemme 3 sur l'expression (4). La condition du lemme est vérifiée en combinant par transitivité les inégalités (2) et (3) ci-dessus avec le fait que $d_i + d_j \leq g$ d'après l'expression (1) de la section 3 que nous avons en hypothèse et on en déduit que :

$$(t - \phi_j) \ mod \ g + (\phi_j - \phi_i) \ mod \ g \ < \ d_i$$

Or, toujours d'après l'expression (1) de la section 3 :

$$d_i \ \leq \ (\phi_j - \phi_i) \ mod \ g$$

Ce qui, par transitivité, imposerait que :

$$(t - \phi_j) \ mod \ g \ < \ 0$$

Ce qui est impossible dans les entiers naturels.

□

4.5. Preuve de la réciproque en Coq

La réciproque se démontre en passant par sa *contraposée* et en utilisant le théorème de *Bachet-Bézout*, ce qui amène à deux remarques préliminaires (4.5.1 et 4.5.2) avant la preuve proprement dite (4.5.3).

4.5.1. Contraposée et Décidabilité

En logique classique, la démonstration d'une proposition par sa contraposée suffit à démontrer la proposition. Par contre, en logique intuitioniste (la logique par défaut de Coq), une proposition implique certes sa contraposée :

$$(A \ \Rightarrow \ B) \ \Rightarrow \ (\overline{B} \ \Rightarrow \ \overline{A})$$

mais la réciproque n'est pas nécessairement vraie. Faire une preuve d'une proposition par sa contraposée n'est donc pas, dans le cas général, une preuve d'une proposition. Cependant, dans le cas particulier où la proposition B est *décidable* - c'est-à-dire si on peut toujours choisir B ou $\overline{B}$ - la réciproque est vraie :

$$(\overline{B} \ \Rightarrow \ \overline{A}) \ \Rightarrow \ (A \ \Rightarrow \ B)$$

Dans notre cas, la proposition B est la formule (1) de la section 3. La preuve de la réciproque a donc nécessité de démontrer que cette expression est bien décidable, ce qui est vrai par le fait que les inégalités sur les entiers sont décidables, puis que cette décidablité se transmet par la quantification universelle sur les listes de tâches.

Pour finir, la bibliothèque de Coq fournit un théorème, `Decidable.contrapositive`, qui permet de transformer la contraposée d'une proposition par la proposition elle-même sous l'hypothèse de décidabilité.

4.5.2. Bachet-Bézout sur les entiers naturels

Pour notre modèle, nous avons choisi d'utiliser le type `nat` qui représente les entiers naturels (donc non signés). Or, le théorème de Bachet-Bézout, qui s'énonce ainsi :

Théorème (Bachet-Bézout)**.** *Si a et b sont deux entiers relatifs non tous deux nuls, et g leur PGCD, alors il existe deux entiers relatifs u et v tels que :*

$$a.u + b.v = g$$

se trouve bien dans la bibliothèque de Coq, mais s'applique sur les entiers relatifs (signés) et n'est donc pas utilisable directement dans notre cas.

Cependant, nous pouvons remarquer qu'en réalité, ce théorème peut s'appliquer sur les entiers naturels, sous la forme suivante :

Théorème 2 (Bachet-Bézout sur les entiers naturels)**.** *Si a et b sont deux entiers naturels non tous deux nuls, et g leur PGCD, alors il existe deux entiers naturels u et v tels que :*

$$a.u = b.v + g$$

Démonstration. Comme la démonstration classique de Bachet-Bézout sur les entiers relatifs, en passant par un lemme qui discrimine les deux cas $(u \geq 0 \wedge v < 0)$ et $(u < 0 \wedge v \geq 0)$. Le premier cas, avec $v < 0$, conduit directement au résultat en faisant passer $b.v$ de l'autre côté de l'égalité. Et on peut ramener le deuxième cas au premier, en remarquant que si (u, v) est un couple solution, alors $(u + kb, v - ka)$ l'est aussi, quel que soit k entier relatif, ce qui permet, par un choix judicieux de k, de changer les signes des solutions pour u et v. □

4.5.3. Contraposée de la réciproque

Elle s'énonce ainsi :

```
Theorem Jan_Korst_conversion_contraposition : ∀ tasks : list Task,
  (∀ t, t ∈ tasks → 0 < duration t ≤ period t)
  → ExistsOrdPairs (fun τ1 τ2 => not (korst_test_two_tasks τ1 τ2)) tasks
    → not (schedulable tasks).
```

Démonstration. On suppose donc que, pour deux tâches τ_i et τ_j, l'expression (1) de la section 3 n'est pas vérifiée et on montre qu'alors, ces tâches sont en conflit à une certaine date, et donc que le système des tâches n'est pas ordonnançable, ce qui est contradictoire.

Pour cela, on applique le théorème 2 (Bachet-Bézout sur les entiers naturels) sur les périodes P_i et P_j des deux tâches (en notant g leur PGCD), ce qui s'exprime par l'existence de u et v, entiers naturels, tels que :

$$P_i.u = P_j.v + g \tag{5}$$

Si nous considérons la division euclidienne de $\phi_j - \phi_i$ par g, nous pouvons écrire :

$$\phi_j - \phi_i = g.q + r \ \wedge \ r < g \tag{6}$$

avec :

$$r = (\phi_j - \phi_i) \ mod \ g \tag{7}$$

Or, nous avons, dans nos hypothèses, la négation de l'expression (1) de la section 3, ce qui s'exprime, en remplaçant $(\phi_j - \phi_i)\ mod\ g$ par r par l'égalité (7) ci-dessus :

$$d_i > r \vee \ r > g - d_j \tag{8}$$

Ce qui nous amène à considérer deux cas :

1. Soit c'est la première inéquation qui est en cause, c'est-à-dire :

$$d_i > r \tag{9}$$

Montrons que, dans ce cas, il y a conflit, c'est-à-dire que les deux tâches τ_i et τ_j demandent l'accès au processeur à la date t suivante :

$$t = \phi_j + v.q.P_j \tag{10}$$

En effet, on peut, d'une part, réécrire (10), en utilisant les expressions (5) et (6), sous la forme :

$$t - \phi_i = (u.q)\ .\ P_i + r$$

Ce qui exprime que, à la date t, nous sommes à la $(u.q)$-ième période de τ_i, décalée de r. Et comme, d'après (9), r est inférieur à la durée d_i de la tâche, nous sommes dans la zone où τ_i s'exécute, puisque les tâches s'exécutent au début de leurs périodes.

D'autre part, on peut réécrire (10), sous la forme :

$$t - \phi_j = (v.q)\ .\ P_j$$

Ce qui exprime que, à la date t, nous sommes exactement au début d'une exécution de la tâche τ_j, plus précisément à sa $(v.q)$-ième période.

Il y a donc bien conflit entre les tâches.

2. Le deuxième cas possible de la négation de (1), exprimée par (8) est :

$$r > g - d_j \tag{11}$$

Cette fois-ci, le conflit entre les tâches τ_i et τ_j est à la date t suivante :

$$t = \phi_i + P_i.u.(q+1) \tag{12}$$

Et on peut vérifier, de la mème manière, que les deux tâches s'éxécutent en même temps à cette date t, en réécrivant cette expression sous les deux formes suivantes :

$$\begin{aligned} t - \phi_i &= (u.(q+1))\ .\ P_i \\ t - \phi_j &= (v.(q+1))\ .\ P_j + (g - r) \end{aligned}$$

Ce qui termine la démonstration. □

4.5.4. Réciproque

Elle s'énonce ainsi :

```
Theorem Jan_Korst_conversion : ∀ tasks : list Task,
  (∀ t, t ∈ tasks → 0 < duration t ≤ period t)
  → schedulable tasks
    → List.ForallOrdPairs korst_test_two_tasks tasks.
```

Démonstration. Par la contraposée de la réciproque (section 4.5.3) et la démonstration que l'expression (1) de la section 3 est bien décidable (cf. section 4.5.1). □

4.6. Corollaires

Nous avons également prouvé formellement les corollaires suivants :

4.6.1. Somme des durées

Le théorème 1 est parfois présenté sous la forme suivante, plus simple, avec une condition suffisante d'ordonnançabilité :

Corollaire 1. *Un système de tâches est ordonnançable par périodicité stricte si la somme des durées des tâches est inférieure ou égale au PGCD de leurs périodes :*

$$\sum_{\tau_k} d_k \ \leq \ pgcd\ P_k$$

Démonstration. L'énoncé du théorème ne précise pas les phases des tâches, il faut démontrer qu'il existe une solution pour ces phases qui satisfasse la formule (1).

Il suffit de faire démarrer les tâches les unes derrière les autres, c'est-à-dire en choisissant 0 comme phase de la première tâche et, par récurrence, la phase de la n-ième tâche comme somme de la phase et de la durée de la n-1-ième tâche :

$$\phi_n = \phi_{n-1} + d_{n-1}$$

La différence entre les phases de deux tâches quelconques τ_i et τ_j, avec $i < j$, est alors inférieure au PGCD de leurs périodes, car :

$$\phi_j - \phi_i = \sum_{i<k\leq j} (\phi_k - \phi_{k-1}) = \sum_{i<k\leq j} d_{k-1} \leq \sum d_k \leq pgcd\ P_k \leq pgcd\ (P_i, P_j)$$

Et donc :

$$(\phi_j - \phi_i)\ mod\ pgcd\ (P_i, P_j) = \phi_j - \phi_i$$

Par conséquent, la formule (1) s'applique puisque, d'une part :

$$d_i = \phi_{i+1} - \phi_i \leq \phi_j - \phi_i = (\phi_j - \phi_i)\ mod\ pgcd\ (P_i, P_j)$$

et, d'autre part :

$$\phi_j - \phi_i = \sum_{i<k\leq j} d_{k-1} = \sum_{i\leq k\leq j} d_k - d_j \leq \sum d_k - d_j \leq pgcd\ P_k - d_j \leq pgcd\ (P_i, P_j) - d_j$$

□

Remarque. La réciproque est fausse, en toute généralité. Par exemple, la liste des trois tâches suivantes, définies par les couples (durée, période) suivants :

(1, 4) (1, 4) (1, 6)

est ordonnançable (elle respecte le théorème 1 en choisissant les phases correctement ; voir la figure 2), alors que la somme de leurs durées est égale à 3 et que leur PGCD est égal à 2 :

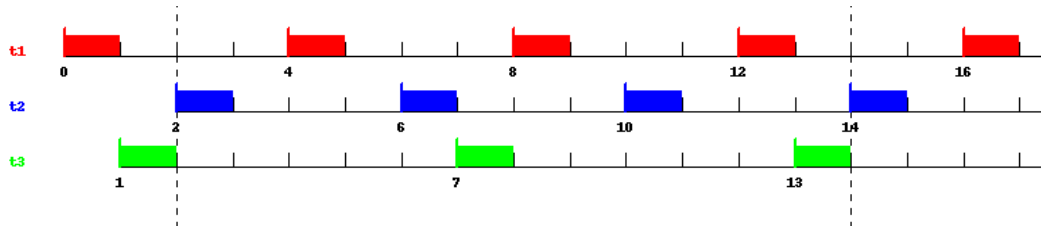

FIGURE 2 – Trois tâches ordonnançables, bien que le corollaire 1 ne s'applique pas.

4.6.2. Cas de deux tâches

Pour deux tâches, le théorème 1 se simplifie en :

Corollaire 2. *Un système de deux tâches de durées respectives d_1 et d_2 et de périodes respectives P_1 et P_2 est ordonnançable par périodicité stricte si et seulement si :*

$$d_1 + d_2 \leq pgcd(P_1, P_2)$$

Démonstration. Par le théorème 1, en faisant démarrer la deuxième tâche juste après la première, comme pour le corollaire 1, c'est-à-dire :

$$\phi_2 = \phi_1 + d_1$$

La réciproque est ici valide (contrairement au cas de `n` tâches, pour `n` supérieur à 3). □

5. Conclusion

Ce travail nous a permis de montrer que le domaine de l'ordonnancement de tâches périodiques semble pouvoir se décrire et se prouver formellement avec un assistant de preuve tel que Coq. À notre connaissance, aucun travail similaire n'a été fait jusqu'ici.

Nous avons commencé une autre formalisation dans la cas d'ordonnancements préemptifs avec priorité fixe, qui sont beaucoup plus complexes et plus intéressants que les ordonnancements du cadre de cet article. Un théorème est en cours de vérification formelle et devrait faire l'objet d'un autre article : il s'agit d'un théorème donnant une propriété fondamentale du temps de réponse des tâches.

Il existe d'autres types d'ordonnancement avec de nombreux types de contraintes qui pourraient donner lieu à d'autres formalisations et vérifications : tâches avec périodes harmoniques, tâches ayant des contraintes de dépendance, préemption avec coût, temps continu, multi-processeur, etc. La recherche est en pointe actuellement, sur ces sujets [6].

Code source

Le code source Coq de cette preuve peut être téléchargé à l'adresse :

`http://pauillac.inria.fr/~ddr/publi/coq_korst/`

La preuve formelle fait environ 1200 lignes de Coq sans compter une quinzaine de lemmes sur le PGCD et la division euclidienne qu'il a fallu ajouter.

Remerciements

Merci à François Delebecque, Laurent George, Mohamed Marouf, Patrick Meumeu, Yves Sorel, Cécile Stentzel et Pierre Weis pour leurs remarques judicieuses et leurs encouragements.

Références

[1] Jan Korst. *Periodic multiprocessor scheduling*. PhD thesis, Eindhoven university of technology, 1992.

[2] C.L. Liu and James Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment, 1973.

[3] Jane W. S. W. Liu. *Real-Time Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[4] The Coq. The Coq Proof Assistant : Reference Manual : Version 7.2. Technical Report RT-0255, INRIA, February 2002.

[5] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Rapport de recherche RR-6455, INRIA, 2008.

[6] P. Meumeu Yomsi. *Prise en compte du coût exact de la préemption dans l'ordonnancement temps réel monoprocesseur avec contraintes multiples*. PhD thesis, Université de Paris Sud, Spécialité Physique, 02/04/2009.

[7] P. Meumeu Yomsi and Y. Sorel. Non-schedulability conditions for off-line scheduling of real-time systems subject to precedence and strict periodicity constraints. In *Proceedings of 11th IEEE International Conference on Emerging technologies and Factory Automation, ETFA'06, WIP*, Prague, Czech Republic, September 2006.

[8] P. Meumeu Yomsi and Y. Sorel. Schedulability analysis with exact number of preemptions and no idle time for real-time systems with precedence and strict periodicity constraints. In *Proceedings of 15th International Conference on Real-Time and Network Systems, RTNS'07*, Nancy, France, March 2007.

[9] O. Kermia and Y. Sorel. Schedulability analysis for non-preemptive tasks under strict periodicity constraints. In *Proceedings of 14th International Conference on Real-Time Computing Systems and Applications, RTCSA'08*, Kaohsiung, Taiwan, August 2008.

[10] Omar Kermia. *Ordonnancement temps réel multiprocesseur de tâches non préemptives avec contraintes de précédence, de périodicité stricte et de latence*. PhD thesis, Université de Paris Sud, 2009.

[11] M. Marouf and Y. Sorel. Schedulability conditions for non-preemptive hard real-time tasks with strict period. In *Proceedings of 18th International Conference on Real-Time and Network Systems, RTNS'10*, Toulouse, France, November 2010.

[12] M. Marouf and Y. Sorel. Scheduling non-preemptive hard real-time tasks with strict periods. In *Proceedings of 16th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA'11*, Toulouse, France, September 2011.

[13] M. Marouf, L. George, and Y. Sorel. Schedulability analysis for a combination of preemptive strict periodic tasks and sporadic tasks. In *Proceedings of the 10th Workshop on Models and Algorithms for Planning and Scheduling Problems, MAPSP'11*, Nymburk, Czech Republic, June 2011.