Table des matières

De choses et d'autres

Rôle de ce volume

Ce document discute de modalités d'ordonnancement et de tests de *cédulabilité* dans un STR. Il s'agit d'une ébauche pour le moment.

Bonne lecture!

Ordonnancement périodique statique

Certains ordonnanceurs ont pour rôle d'assurer l'ordonnancement de tâches dont les conditions d'exécution sont connues *a priori*. C'est souvent le cas de systèmes embarqués au rôle très circonscrit, mais qui sont aussi fréquemment des systèmes assurant des fonctions essentielles, comme par exemple la surveillance de conditions sur des systèmes physiques et la réaction juste à temps face à des conditions pointues.

Cette section s'intéresse aux tâches périodiques dont les paramètres d'exécution sont connus a priori, donc aux ***tâches périodiques statiques***.

Tâche et paramètres d'exécution

Supposons un ensemble de tâches $\gamma = \{T_0, T_1, \dots, T_n\}$ tel que $T_i: (P, C)$. Ici, C est le temps de calcul de la tâche et P est sa période.

Ainsi, l'exécution de T_i doit démarrer (*Release*) au début de chaque intervalle P et se poursuivre pour une durée C. L'ordonnanceur assure la constance du rythme (la partie P, contraintes de constance et d'immédiateté) dans la mesure où les tâches s'exécutent réellement pour la durée prévue (la partie C, contrainte de brièveté), mais les propriétés P et C sont des contraintes propres à la tâche T_i elle-même.

Cycles majeur et mineur

Deux paramètres clés des ordonnancements périodiques statiques sont son *cycle majeur* et son *cycle mineur*.

Le **cycle majeur** d'un ordonnancement statique de tâches périodiques est le plus petit intervalle d'unités de temps menant à un cycle complet d'exécution pour toutes les tâches impliquées – exprimé autrement, il s'agit de la plus petite période à l'intérieur de laquelle toutes les tâches d'un ensemble donné se seront exécutées au moins une fois.

Cela signifie qu'il n'y a pas lieu de prévoir un ordonnancement statique plus long que la longueur du cycle majeur pour l'ensemble des tâches impliquées : en pratique, ce cycle se répétera, tout simplement.

On peut dire que le cycle majeur d'un ensemble de tâches est la période de l'ensemble examiné de manière globale.

Calculer le cycle majeur d'un ensemble γ équivaut à évaluer le **plus petit commun multiple** (PPCM) des périodes P des tâches de cet ensemble.

Le **cycle mineur** d'un ordonnancement statique de tâches périodiques, quant à lui, est le plus petit intervalle de temps au bout duquel une « décision » doit nécessairement être prise par l'ordonnanceur – le plus petit intervalle de temps susceptible de mener au passage d'une tâche à l'autre dans un ordonnancement donné.

Évidemment, le mot « décision » est quelque peu abusif ici, du fait que l'ordonnancement est statique, mais l'idée est que le début d'un cycle mineur est un point sensible pour le lancement d'une tâche périodique; connaître le cycle mineur permet de construire un ordonnancement de manière plus efficace, en réduisant les moments candidats à provoquer un lancement de tâche.

En pratique, une tâche en exécution ne peut chevaucher le moment où on passe d'un cycle mineur à un autre; les débuts de cycle mineurs sont aussi des débuts de tâches.

Calculer le cycle mineur d'un ensemble γ équivaut à évaluer le **plus grand commun diviseur** (PGCD) des périodes *P* des tâches de cet ensemble.

Le **taux d'occupation** d'une tâche T dans un ordonnanceur donné est la partie d'un cycle majeur que le temps de calcul de T occupera. Cela équivaut à :

$$occupation(\gamma, T_i) = \frac{\left(\frac{cycle\ majeur(\gamma)}{P_i}\right) C_i}{cycle\ majeur(\gamma)}$$

Éliminer le diviseur transforme ce taux (réel) en quantité discrète. Évidemment, le taux global d'occupation d'un ordonnanceur est la somme des taux d'occupation de ses tâches :

$$occupation(\gamma) = \sum_{i=0}^{n} occupation(\gamma, T_i)$$

Pour réduire les erreurs sur les réels, on évaluera typiquement l'occupation de manière discrète, à l'aide d'unités d'exécution, et on ne divisera par le cycle majeur de γ qu'au moment d'évaluer le taux d'occupation souhaité.

Conditions « d'ordonnançabilité »

Une *condition nécessaire* mais pas suffisante d'ordonnançabilité pour un ensemble de tâches périodiques statiques γ est que le taux d'occupation soit inférieur ou égal à `100 %` :

$$occupation(\gamma) \leq 1$$

Si on retire le diviseur du calcul de l'occupation d'une tâche, le test d'ordonnançabilité ci-dessus devient une comparaison avec le cycle majeur de γ plutôt qu'avec la constante 1.

Exemple `00` : soit l'ensemble de tâches périodiques statiques suivant :

$$\gamma = \{(P = 5, C = 5)\}$$
$$cycle\ majeur(\gamma) = 5$$
$$cycle\ mineur(\gamma) = 5$$
$$occupation(\gamma) = 1$$

Cet ensemble est banal. Une seule tâche y occupe la totalité du temps de calcul. L'ensemble y est ordonnançable de manière triviale.

Exemple `01` : soit l'ensemble de tâches périodiques statiques suivant :

$$\gamma = \{(P = 6, C = 4)\}$$
$$cycle\ majeur(\gamma) = 6$$
$$cycle\ mineur(\gamma) = 6$$
$$occupation(\gamma) \cong 0{,}667\ ou\ \frac{2}{3}$$

Cet ensemble est aussi banal. Une seule tâche doit s'y exécuter. Deux tiers du processeur seulement y sont accaparés. L'ensemble y est ordonnançable de manière triviale.

Exemple 02 : soit l'ensemble de tâches périodiques statiques suivant :

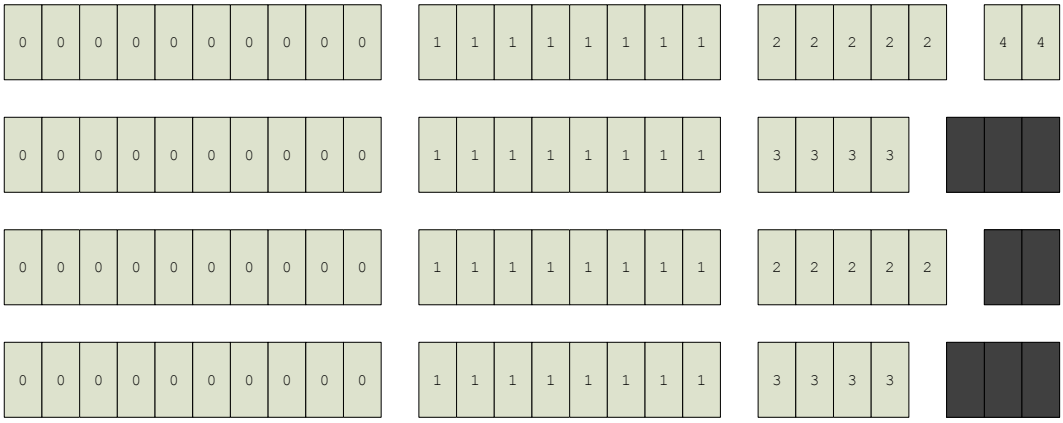
$$\gamma = \{(P = 25, C = 10), (P = 25, C = 8), (P = 50, C = 5), (P = 50, C = 4), (P = 100, C = 2)\}$$

$$cycle\ majeur(\gamma) = 100$$

$$cycle\ mineur(\gamma) = 25$$

$$occupation(\gamma) = 0{,}92$$

Il est possible de construire un ordonnancement pour γ. En effet :



La représentation ci-dessus montre le cycle majeur (taille 100) et quatre cycles mineurs (chacun de taille 25), un par ligne. Chaque tâche T_i apparaît comme une case identifiée par la mention i (p. ex. : T_0 occupe les cases marquées 0, T_1 occupe les cases marquées 1, …). Les cases ombragées sont des cases disponibles.

Il est simple de vérifier ici que les périodes et les temps de calcul de chaque tâche sont respectés.

Exemple 03 : soit l'ensemble de tâches périodiques statiques suivant :

$$\gamma = \{(P = 5, C = 6)\}$$

$$cycle\ majeur(\gamma) = 5$$

$$cycle\ mineur(\gamma) = 5$$

$$occupation(\gamma) = 1{,}2\ ou\ \frac{6}{5}$$

Autre ensemble banal. Une seule tâche doit s'y exécuter, mais son temps de calcul excède sa période. L'occupation du processeur dépasse 100 %. L'ensemble γ n'y est pas ordonnançable, ce qui est constaté encore une fois de manière triviale.

Exemple 04 : soit l'ensemble de tâches périodiques statiques suivant :

$$\gamma = \{(P = 6, C = 4), (P = 12, C = 3)\}$$

$$cycle\ majeur(\gamma) = 12$$

$$cycle\ mineur(\gamma) = 6$$

Cet ensemble est moins banal que les précédents. Deux tâches y sont prévues, qu'on peut identifier par $T_0 : (P = 6, C = 4)$ et $T_1 : (P = 12, C = 3)$ le cycle majeur correspond à $PPCM(12,6)$, donc 12, alors que le cycle mineur correspond à $PGCD(12,6)$, donc 6.

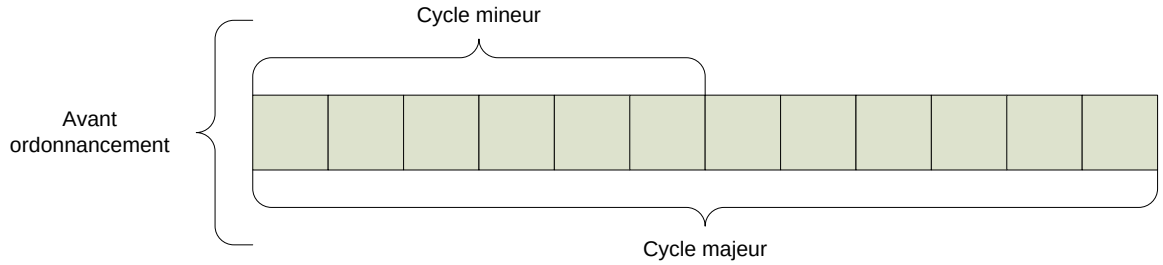
Ici, l'occupation totale de γ est inférieure à `100 %` puisque

$$occupation(\gamma, T_0) = \frac{\left(\frac{12}{6}\right)4}{12} = \frac{8}{12} = \frac{2}{3}$$

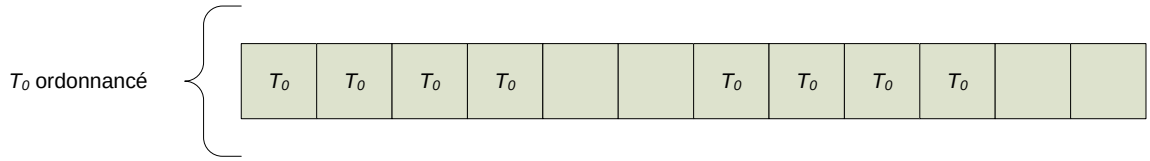
$$occupation(\gamma, T_1) = \frac{\left(\frac{12}{12}\right)3}{12} = \frac{3}{12} = \frac{1}{4}$$

$$occupation(\gamma) = \frac{2}{3} + \frac{1}{4} = \frac{8}{12} + \frac{3}{12} = \frac{11}{12} \cong 0{,}91667$$

Pourtant, l'ensemble γ n'y est pas ordonnançable. Ceci se constate par construction d'un ordonnancement possible pour γ. Avant ordonnancement, on peut représenter l'ordonnanceur comme un tableau d'unités d'exécution discrètes de même taille, la taille de ce tableau étant donnée par le cycle majeur de l'ensemble γ :



L'ordonnancement d'une première tâche (ici, $T_0 : \{P = 6, C = 4\}$) mène à un remplissage partiel de ce tableau :



Une fois T_0 ordonnancé, il reste à ordonnancer T_1, or T_1 a un temps d'exécution de 3, et il ne reste aucun espace pour une tâche de cette durée dans l'ordonnanceur. Conséquemment, $\gamma = \{(P = 6, C = 4), (P = 12, C = 3)\}$ n'est pas ordonnançable.

Vérification des conditions « d'ordonnançabilité »

Il est évidemment possible de valider *a priori* les conditions d'ordonnançabilité d'un ensemble de tâches périodiques statiques. Certaines sont simples :

- calculer le cycle majeur et le cycle mineur de l'ensemble;
- évaluer le taux d'occupation global de cet ensemble;
- s'assurer qu'il soit inférieur ou égal à 1.

Ceci permet de détecter rapidement certains ensembles qu'il est impossible d'ordonnancer (l'exemple `03` plus haut en est un exemple).

D'autres sont plus complexes, comme l'illustre l'exemple `04`. Pour cette raison, si le test simple du taux d'occupation réussit, il faut procéder à la construction d'un ordonnancement pour l'ensemble de tâches sous examen. L'ensemble est évidemment ordonnançable seulement si cette construction réussit.

La vérification dynamique des conditions est un problème relativement simple à solutionner à l'aide de techniques conventionnelles (un tableau de booléens et une liste d'instances d'une classe représentant une tâche constituent un bon point de départ).

Une classe `Tache_` pourrait par exemple se présenter comme suit :

```
class Tache_
{
public:
   typedef int id_type;
private:
   int periode_;
   int calcul_;
   id_type id_;
   static id_type id_cur;
public:
   Tache_(int periode, int calcul)
      : periode_(periode), calcul_(calcul), id_(id_cur++)
   {
   }
   id_type id() const
      { return id_; }
   int periode() const
      { return periode_; }
   int calcul() const
      { return calcul_; }
};
```

En ajoutant des opérateurs de comparaison et quelques autres outils simples, un algorithme pour tenter de bâtir un ordonnancement pour une liste d'instances de `Tache_` est relativement simple (vous pouvez le faire à titre d'exercice).

Notez le choix du nom `Tache_` ici (avec un soulignement à la fin). Pour les fins de l'exercice, nous utiliserons `Tache` à titre de description statique d'une tâche (plus loin).

Vérification statique des conditions « d'ordonnançabilité »

Nous examinerons plutôt comment valider les conditions d'ordonnançabilité de manière statique, donc dès la compilation. En d'autres mots : comment peut-on écrire un ordonnanceur statique qui ne compilera que si l'ensemble des tâches qu'il doit ordonnancer est bel et bien ordonnançable?

Pour y arriver, nous aurons recours à tout un arsenal de techniques de métaprogrammation. Notez que nous obtiendrons en fin de compte un programme à forte résilience, au sens où il ne compilera (en quelque sorte) que si on sait qu'il s'exécutera correctement[1], mais ce gain de robustesse se fera au détriment d'un coût accru en termes de temps de compilation.

Nous apprécierons ainsi l'apport de tests simples (ceux basés sur le taux d'occupation), qui mènent à un échec rapide de la compilation lorsque cela s'avère opportun, et qui évitent parfois d'entrer dans des échafaudages complexes, *allant jusqu'à la construction statique d'un ordonnancement complet pour un ensemble de tâches*.

Éléments de base

Nous couvrirons ici quelques outils de base qui sous-tendent l'ensemble du reste de notre démarche. La plupart de ces éléments vous sont s'ailleurs probablement connus à ce stade, dû à des lectures antérieures.

Représenter une constante entière par un type

Il est parfois – assez souvent, pour être honnête – utile de représenter une constante entière par un type à part entière.

La technique appliquée dans cet exemple est empruntée à **Dave Abrahams** et à **Aleksey Gurtovoy**, deux icônes de *Boost*, dans leur (excellent, mais costaud) volume *C++ Template Metaprogramming*[2].

Avec cette technique, l'entier `N` peut être représenté de manière unique par le type `int_<N>` contenant la valeur `int_<N>::VAL`, qui vaut… `N`.

```
template <int N>
   struct int_
   {
      enum { VAL = N };
   };
```

À travers les divers types `int_`, il est possible d'insérer l'équivalent « type » de constantes entières dans des structures de données statiques, comme des listes de types, et de les manipuler de manière statique.

Cette technique est d'une grande simplicité, mais est aussi d'une utilité inestimable.

[1] … dans la mesure où la description *a priori* des tâches ne ment pas, évidemment.

[2] Voir http://h-deb.clg.qc.ca/Liens/Suggestions-lecture.html#cpp-template-metaprogramming pour des détails.

Listes de types

Les listes de types ont été couvertes dans ***STR—Volume 03***, de même que dans Internet[3]. L'idée est d'**Andrei Alexandrescu**, dans son excellent livre *Modern C++ Design*, et est enrichie ici de quelques prédicats simples pour déduire, de manière statique, la tête et la queue d'une liste de types.

Les listes de types apparaîtront un peu partout dans cette section. Un peu comme la technique de « conversion » de constantes entière en types distincts, cette technique est extrêmement féconde.

Entre autres, nous représenterons un ensemble de tâches à ordonnancer comme une liste de types, où chaque élément de la liste sera un type représentant une tâche à part entière.

[3] Entre autres dans http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Metaprogrammation.html

Représenter une tâche comme un type

Notez que les types `static_if_else`, `Compilable` et `Incompilable` sont décrits à l'***annexe 00***.

Pour construire une liste de tâches périodiques statiques à la compilation, il nous faut donner une représentation statique des tâches. Ici, nous ferons une liste de types, chacun représentant une et une seule tâche.

Une première ligne de protection statique sera directement implémentée à même les valeurs de P et de C : un temps de calcul excédant la période de la tâche est illégal *a priori*, et est traité comme tel.

Pour simplifier le code client, les classes `extract_calcul` et `extract_period` sont définis pour déduire le type décrivant le temps de calcul et la période d'une tâche donnée.

Enfin, un foncteur `afficher_tache`, en partie statique, permettra de décrire sur un flux donné les paramètres particuliers d'une tâche. Combiné aux algorithmes statiques (plus bas), ce foncteur nous permettra de déboguer notre code.

```
class calcul_plus_long_que_periode {};
template <int P, int C>
   struct Tache
      : static_if_else<
           (C <= P),
           Compilable,
           Incompilable<calcul_plus_long_que_periode>
        >::type
   {
      typedef int_<P> periode_type;
      typedef int_<C> calcul_type;
   };
template <class>
   struct extract_period;
template <int P, int C>
   struct extract_period<Tache<P, C> >
      { typedef typename Tache<P, C>::periode_type type; };
template <class>
   struct extract_calcul;
template <int P, int C>
   struct extract_calcul<Tache<P, C> >
      { typedef typename Tache<P, C>::calcul_type type; };
class afficher_tache
{
   std::ostream &os_;
   const char *suffixe_;
public:
   afficher_tache(std::ostream &os, const char *suffixe ="")
      : os_(os), suffixe_(suffixe)
   {
   }
   template <class T>
      void operator()(T)
      {
         os_ << '(' << typename extract_period<T>::type::VAL
             << ',' << typename extract_calcul<T>::type::VAL
             << ')' << suffixe_;
      }
};
```

Outils numériques statiques

Pour réaliser de tâches comme le calcul statique du `PPCM` ou du `PGCD` d'un ensemble de tâches, nous aurons recours à un certain nombre d'outils numériques statiques.

Outils booléens

Il est possible de faire des opérations statiques sur des booléens.

Dans les exemples à droite :

- le type `true_type` représente le concept de vrai alors que le type `false_type` représente le concept de faux;
- le type générique `bool_to_type` transforme les constantes true et false en types `true_type` et `false_type`, respectivement; et
- le type générique `est_bool_type`, qui permet de détecter si un type représente ou non un booléen.

Ces outils permettent de représenter des booléens par des types et de raisonner sur eux à la compilation, tout en tirant profit des mécanismes normaux du langage. Ils sont même plus précis que les booléens usuels, puisqu'il n'existe qu'une seule valeur valide pour le concept *vrai*.

```
class true_type {};
class false_type {};
template <class T>
   struct type_to_bool;
template <>
   struct type_to_bool<true_type>
      { enum { VAL = true }; };
template <>
   struct type_to_bool<false_type>
      { enum { VAL = false }; };
template <bool>
   struct bool_to_type;
template <>
   struct bool_to_type<true>
      { typedef true_type type; };
template <>
   struct bool_to_type<false>
      { typedef false_type type; };
template <class T>
   struct est_bool_type
      { enum { VAL = false }; };
template <>
   struct est_bool_type<true_type>
      { enum { VAL = true }; };
template <>
   struct est_bool_type<false_type>
      { enum { VAL = true }; };
```

Lorsque cela s'avère pertinent, il est possible de réaliser des opérations logiques statiques sur les types représentant de manière non-ambiguë les concepts de vrai et de faux.

Une variante intéressante serait d'implémenter le *et* logique et le *ou* logique sous la forme d'une alternative statique, par exemple :

```
template <class A, class B>
   struc static_or
   {
      typedef typename
         static_if_else<
            bool_to_type<A>::VAL,
            true_type,
            B
         >::type type;
   };
```

qui correspond directement à vrai si `A` représente le concept de vrai et `B` (qu'il soit vrai ou faux) sinon.

```
template <class A, class B>
   struct static_and
   {
      typedef typename
         bool_to_type<
            type_to_bool<A>::VAL && type_to_bool<B>::VAL
         >::type type;
   };
template <class A, class B>
   struct static_or
   {
      typedef typename
         bool_to_type<
            type_to_bool<A>::VAL || type_to_bool<B>::VAL
         >::type type;
   };
template <class>
   struct static_negate;
template <>
   struct static_negate<true_type>
   {
      typedef false_type type;
   };
template <>
   struct static_negate<false_type>
   {
      typedef true_type type;
   };
```

Gestion des intervalles

Dans le respect du principe *DRY*[4], nous définirons la déclinaison statique de l'un des quatre opérateurs relationnels (l'opérateur <, dans le respect des traditions), et nous exprimerons les trois autres sur la base de cet opérateur et de sa négation logique[5].

```
template <int N, int SEUIL>
   struct est_plus_petit_que
   {
      typedef typename
         bool_to_type <(N < SEUIL)>::type type;
   };
template <int N, int SEUIL>
   struct est_plus_petit_egal_que
   {
      typedef typename
         static_negate<
            typename est_plus_petit_que<SEUIL, N>::type
         >::type type;
   };
template <int N, int SEUIL>
   struct est_plus_grand_que
   {
      typedef typename
         est_plus_petit_que<SEUIL, N>::type type;
   };
template <int N, int SEUIL>
   struct est_plus_grand_egal_que
   {
      typedef typename
         static_negate<
            typename est_plus_petit_que<N, SEUIL>::type
         >::type type;
   };
```

[4] Pour *Don't Repeat Yourself*. Attribué à ***Dave Thomas*** et à ***Andrew Hunt***.

[5] Cette démarche est décrite dans XXX.

Vérifier l'appartenance d'un entier à un intervalle connu à la compilation est banal.

```
template <int N, int BORNE_MIN, int BORNE_MAX>
   struct est_dans_inclusif
   {
      typedef typename
         static_and<
            typename
               est_plus_petit_egal_que<BORNE_MIN, N>::type,
            typename
               est_plus_petit_egal_que<N, BORNE_MAX>::type
         >::type type;
   };
template <int N, int BORNE_MIN, int BORNE_MAX>
   struct est_dans_exclusif
   {
      typedef typename
         static_and<
            typename
               est_plus_petit_que<BORNE_MIN, N>::type,
            typename
               est_plus_petit_que<N, BORNE_MAX>::type
         >::type type;
   };
template <int N>
   struct est_positif
   {
      typedef typename
         est_plus_grand_egal_que<N, 0>::type type;
   };
template <int N>
   struct est_negatif
   {
      typedef typename
         static_negate<
            typename est_positif<N>::type
         >::type type;
   };
```

Évidemment, tester ces outils est plutôt simple.

```
#include <iosfwd>
template <int A, int B>
   void tester_operateurs_relationnels(std::ostream &os)
   {
      if (type_to_bool<est_plus_petit_que<A, B>::type>::VAL)
         os << A << " < " << B << std::endl;
      if (type_to_bool<est_plus_petit_egal_que<A, B>::type>::VAL)
         os << A << " <= " << B << std::endl;
      if (type_to_bool<est_plus_grand_que<A, B>::type>::VAL)
         os << A << " > " << B << std::endl;
      if (type_to_bool<est_plus_grand_egal_que<A, B>::type>::VAL)
         os  << A << " >= " << B << std::endl;
   }
```

Outils numériques

Puisque nous allons réaliser plusieurs opérations à la compilation (des sommes, des produits, l'évaluation du `PPCM` et du `PGCD`, *etc.*), nous allons nous doter d'outils numériques eux-mêmes statiques.

Remarquez que, pour la plupart des opérations, le résultat est évalué de manière plutôt directe. Par exemple, la somme de deux entiers statiques est obtenue par leur addition dès la compilation.

Pour d'autres opérations, le résultat est évalué de manière un peu plus subtile. Par exemple, la division statique doit tenir compte de la possibilité que le diviseur soit nul (il en va de même pour le modulo, qui est une variante de la division entière dans laquelle le reste, pas le quotient, est ce qui nous intéresse).

Dans le but d'obtenir des messages d'erreurs significatifs et lisibles, nous utilisons la technique des classes incompilables (voir l'***annexe 00***) qui fera en sorte de bloquer la compilation et d'offrir au programmeur l'équivalent d'un message d'erreur pertinent lorsque le code sera clairement illégal.

Une variante de cette technique, reposant sur des fonctions munies de types gardiens, apparaîtra plus loin.

```
#include "Incompilable.h"
#define ADMETTRE_PPCM_0_0
#define ADMETTRE_PGCD_0_0
template <int U, int V = 0>
   struct addition
   {
      enum { VAL = U + V };
   };
template <int U, int V = 0>
   struct soustraction
   {
      enum { VAL = U - V };
   };
template <int U, int V = 1>
   struct multiplication
   {
      enum { VAL = U * V };
   };
class division_par_zero {};
template <int U, int V = 1>
   struct division
      : static_if_else<
           V,
           Compilable,
           Incompilable<division_par_zero>
        >::type
   {
      enum { VAL = U / V };
   };
template <int U, int V = 1>
   struct modulo
      : static_if_else<
           V,
           Compilable,
           Incompilable<division_par_zero>
        >::type
   {
      enum { VAL = U % V };
   };
```

Examinons maintenant quelques algorithmes numériques statiques

Explications (les gardes sont de moi; plusieurs algos sont de http://www.codeproject.com/cpp/meta_programming.asp)

```
//
// u est-il divisible par v? On
utilise un entier pour se
// permettre une sentinelle avec une
valeur autre que 1 ou 0
//


template <int u, int v>
   struct est_divisible
   {
      typedef typename
         bool_to_type<u % v ==
0>::type type;
   };
//
// cas pathologique de la division
par zéro (non convertible par
// les outils statiques plus haut)
//
template <int u>
   struct est_divisible<u, 0>
   {
      typedef division_par_zero type;
   };
//
// version artihmétique pour certains
algorithmes
//
template <int u, int v>
   struct est_divisible_arith
   {
      typedef typename
         static_if_else<
            (
               type_to_bool<
                  typename
est_divisible<u, v>::type
               >::VAL
            ),
            int_<1>,
            int_<0>
         >::type type;
   };
```

Bla

Explications

```
// ---------------------------
// TODO: test «T est convertible en int»?

//
// Valeur plafond d'une autre, peu importe son
type d'origine
//
template <class T>
   struct plafond
   {
       static int valeur (T src)
       {
          return src > 0 ? src + 1 : src;
       }
   };

//
// Valeur plancher d'une autre, peu importe son
type d'origine
//
template <class T>
   struct plancher
   {
       static int valeur (T src)
       {
          return src > 0 ? src : src - 1;
       }
   };
```

Bla

Explications

```
template <int val>
   struct est_pair
   {
      enum
      {
         VAL = type_to_bool<est_divisible<val,
2>::type>::VAL
      };
   };
template <int val>
   struct est_impair
   {
      enum { VAL = !est_pair<val>::VAL };
   };
```

Bla

Explications

```
template <int, int, class>
   struct pgcd;
template <int u, int v>
   struct pgcd<u, v, garde_usage_interne>
   {
      enum { VAL = pgcd<v, u % v, garde_usage_interne>::VAL };
   };

template <int u>
   struct pgcd<u, 0, garde_usage_interne>
   {
      enum { VAL = u };
   };

#ifdef ADMETTRE_PGCD_0_0

template <>
   struct pgcd<0, 0, garde_usage_interne>
   {
      enum { VAL = 0 };
   };
template <int u, int v>
   int calculer_pgcd()
      { return pgcd<u, v, garde_usage_interne>::VAL; }
#else
template <>
   struct pgcd<0, 0, garde_usage_interne>;
//
// fonction de protection
//
template <int u, int v>
   int calculer_pgcd()
   {
      static_assert<u || v> le_pgcd_de_0_et_de_0_est_illegal;
      unused(le_pgcd_de_0_et_de_0_est_illegal);
      return pgcd<u, v, garde_usage_interne>::VAL;
   }

#endif

template <class T, class U>
   struct static_pgcd
   {
      typedef int_<pgcd<T::VAL, U::VAL, garde_usage_interne>::VAL>
type;
   };
```

Bla

Explications

```
template <int, int, class>
   struct ppcm;
template <int u, int v>
   struct ppcm<u, v, garde_usage_interne>
   {
      enum { VAL = u * v / pgcd<u, v, garde_usage_interne>::VAL };
   };
template <int u>
   struct ppcm<u, 0, garde_usage_interne>
   {
      enum { VAL = 0 };
   };
template <int v>
   struct ppcm<0, v, garde_usage_interne>
   {
      enum { VAL = 0 };
   };
#ifdef ADMETTRE_PPCM_0_0

template <>
   struct ppcm<0, 0, garde_usage_interne>
   {
      enum { VAL = 0 };
   };
template <int u, int v>
   int calculer_ppcm()
      { return ppcm<u, v, garde_usage_interne>::VAL; }
#else
template <>
   struct ppcm<0, 0, garde_usage_interne>;
template <int u, int v>
   int calculer_ppcm ()
   {
      static_assert<u || v> le_ppcm_de_0_et_0_est_illegal;
      unused(le_ppcm_de_0_et_0_est_illegal);
      return ppcm<u, v, garde_usage_interne>::VAL;
   }

#endif

template <class T, class U>
   struct static_ppcm
   {
      typedef int_<ppcm<T::VAL, U::VAL, garde_usage_interne>::VAL>
type;
   };
```

Bla

Explications

```
//
// coprimauté
//
template <int u, int v>
   struct copremiers
   {
      typedef typename
         bool_to_type<
            pgcd<u, v, garde_usage_interne>::VAL == 1
         >::type type;
   };
```

Bla

Explications

```
//
// Nb. diviseurs (construction récursive)
//
template <int, class>
   struct nb_diviseurs;
template <int n>
   struct nb_diviseurs<n, garde_usage_interne>
   {
   private:
      template <int debut, int fin>
         struct nb_diviseurs_intervalle
         {
            enum {
               VAL =
                  (type_to_bool<est_divisible<fin,
debut>::type>::VAL? 1 : 0) +
                   nb_diviseurs_intervalle<debut + 1, fin>::VAL
            };
         };
      template <int fin>
         struct nb_diviseurs_intervalle<fin, fin>
         {
            enum { VAL = 1 };
         };
   public:
      enum { VAL = nb_diviseurs_intervalle<1, n>::VAL };
   };
```

Bla

Explications

```
template <int n>
   int calculer_nb_diviseurs()
   {
      static_assert<type_to_bool<est_positif<n>::type>::VAL>
```

```
le_calcul_du_nombre_de_diviseurs_requiert_un_entier_positif;

unused(le_calcul_du_nombre_de_diviseurs_requiert_un_entier_positif);
      return nb_diviseurs<n, garde_usage_interne>::VAL;
   }
```

Bla

Explications

```
template <int, class>
   struct est_premier;
template <int n>
   struct est_premier<n, garde_usage_interne>
   {
      //
      // Seuls les n qui sont premiers ont deux diviseurs distincts
entre 1
      // et n inclusivement. L'entier 1 n'en a qu'un!
      //
      typedef typename
         bool_to_type<(nb_diviseurs<n, garde_usage_interne>::VAL ==
2)>::type type;
   };
```

Bla

Explications

```
template <int n>
   bool calculer_est_premier()
   {
      static_assert<type_to_bool<est_positif<n>::type>::VAL>
         evaluer_la_primaute_requiert_un_entier_positif;
      unused(evaluer_la_primaute_requiert_un_entier_positif);
      return type_to_bool<est_premier<n,
garde_usage_interne>::type>::VAL;
   }
```

Bla

Explications

```
//
// puissance
//
template <int, int, class>
   struct puissance;
template <int base, int exposant>
   struct puissance<base, exposant,
garde_usage_interne>
   {
      enum { VAL = base * puissance<base,
exposant-1, garde_usage_interne>::VAL };
   };
template <int base>
   struct puissance<base, 0,
garde_usage_interne>
   {
      enum { VAL = 1 };
   };
//
// fonction de protection
//
template <int base, int exposant>
   int calculer_puissance()
   {
      static_assert<base || exposant>
         zero_a_la_zero_est_indefini; // éviter
0^0
      unused(zero_a_la_zero_est_indefini);
      return puissance<base, exposant,
garde_usage_interne>::VAL;
   }
```

Bla

Explications

```
//
// factorielle
//
template <int, class>
   struct factorielle;

template <int n>
   struct factorielle<n, garde_usage_interne>
   {
      enum { VAL = n * factorielle<n-1,
garde_usage_interne>::VAL };
   };
template <>
   struct factorielle<0, garde_usage_interne>
   {
      enum { VAL = 1 };
   };
//
// fonction de protection
//
template<int n>
   int calculer_factorielle()
   {
      static_assert<(n >= 0)>
factorielle_de_n_implique_n_positif;

unused(factorielle_de_n_implique_n_positif);
      return factorielle<n,
garde_usage_interne>::VAL;
   }
```

Bla

Explications

```
template <class T, class U>
   struct static_min
   {
   private:
      enum { test = T::VAL < U::VAL };
   public:
      typedef typename
         static_if_else<
            test,
            T,
            U
         >::type type;
   };
template <class T, class U>
   struct static_max
   {
   private:
      enum { test = U::VAL < T::VAL };
   public:
      typedef typename
         static_if_else<
            test,
            T,
            U
         >::type type;
   };
```

Bla

Explications

```
struct static_add_base
{
   typedef int_<0> neuter;
};
template <class T, class U>
   struct static_add
      : static_add_base
   {
      typedef int_<T::VAL + U::VAL> type;
   };
```

bla

Algorithmes statiques

Bla

Algorithmes sans dépendances

bla

Explications

```
// count_consecutive<TList,T>::VAL est le
nombre d'occurrences
// consécutives de T au début de TList
//
// TODO: rewrite in terms of static_count_if
//
// Depends on: nil
//
template <class TList, class T>
   struct count_consecutive;
template <class T, class Q, class U>
   struct count_consecutive<type_list<T, Q>, U>
   {
      enum { VAL = 0 };
   };
template <class T, class Q>
   struct count_consecutive<type_list<T, Q>, T>
   {
      enum { VAL = 1 +
count_consecutive<Q,T>::VAL };
   };
template <class T>
   struct count_consecutive<type_list<T, Vide>,
T>
   {
      enum { VAL = 1 };
   };
```

bla

Explications

```
//
// Trouver l'indice d'un type dans une liste de
types
//
// Depends on: nil
//
template <class TList, class T>
   struct indice_par_type;
template <class T>
   struct indice_par_type<Vide, T>
   {
      enum { VAL = -1 };
   };
template <class T, class Q>
   struct indice_par_type<type_list<T, Q>, T>
   {
      enum { VAL = 0 };
   };
template <class T, class Q, class U>
   struct indice_par_type<type_list<T, Q>, U>
   {
   private: // type privé, une sorte de
temporaire
      enum
      {
         RechercheDansQueue =
indice_par_type<Q, U>::VAL
      };
   public: // l'indice en tant que tel
      enum
      {
         VAL = RechercheDansQueue == -1? -1 : 1
+ RechercheDansQueue
      };
   };
```

bla

Explications

```
//
// Insère un type à la fin d'une liste de types
//
// Depends on: nil
//
template <class TList, class>
   struct inserer_fin;
template <class T, class Q, class U>
   struct inserer_fin<type_list<T, Q>, U>
   {
```

```
      typedef type_list <
         T, typename inserer_fin<Q, U>::type
      > type;
   };
template <class T, class U>
   struct inserer_fin<type_list<T, Vide>, U>
   {
      typedef type_list <
         T, type_list<
            U, Vide
         >
      > type;
   };
```

bla

Explications

```
//
// Compte le nombre d'éléments d'une liste pour
lesquels un
// prédicat s'avère (important: on considère 1
comme vrai et
// 0 comme faux)
//
// Depends on: nil
//
template <class TList, template <class> class
Pred>
   struct static_count_if;
template <class T, class Q, template <class>
class Pred>
   struct static_count_if<type_list<T, Q>,
Pred>
   {
      enum
      {
         VAL = Pred<T>::VAL +
static_count_if<Q, Pred>::VAL
      };
   };
template <class T, template <class> class Pred>
   struct static_count_if<type_list<T, Vide>,
Pred>
   {
      enum
      {
         VAL = Pred<T>::VAL
      };
   };
```

bla

Explications

```
//
// Insérer un type dans une liste de types
//
// Depends on: nil
//
template <class TList, class T>
   struct static_insert;
template <>
   struct static_insert <Vide, Vide>
   {
      typedef Vide type;
   };
template <class T>
   struct static_insert<Vide, T>
   {
      typedef type_list<T, Vide> type;
   };
template <class T, class Q>
   struct static_insert<Vide, type_list<T, Q> >
   {
      typedef type_list<T, Q> type;
   };
template <class T, class Q, class U>
   struct static_insert<type_list<T, Q>,U>
   {
      typedef
         type_list<T, typename static_insert<Q,
U>::type> type;
   };
```

bla

Explications

```
//
// static_length<TList>::VAL est le nombre
d'éléments dans
// la liste de types TList
//
// Depends on: nil
//
template <class>
    struct static_length;
template <>
   struct static_length<Vide>
   {
      enum { VAL = 0 };
   };
template <class T, class U>
   struct static_length< type_list<T, U> >
   {
      enum { VAL = 1 + static_length<U>::VAL };
   };
```

bla

Explications

```
//
// Supprimer un type d'une liste de types
//
// Depends on: nil
//
template <class TList, class T>
   struct static_remove;
template <class T>
   struct static_remove<Vide, T>
   {
      typedef Vide type;
   };
template <class T, class Q>
   struct static_remove<type_list<T, Q>, T>
   {
      typedef Q type;
   };
template <class T, class Q, class U>
   struct static_remove<type_list<T, Q>, U>
   {
      typedef type_list<T, typename
static_remove<Q, U>::type> type;
   };
```

bla

Explications

```
//
// Trouver un type par son indice dans une
liste de types
//
// Depends on: nil
//
template <class TList, unsigned int i>
   struct type_par_indice;
template <class T, class Q>
   struct type_par_indice<type_list<T,Q>,0>
   {
      typedef T type;
   };
template <class T, class Q, unsigned int i>
   struct type_par_indice<type_list<T,Q>,i>
   {
      typedef typename
         type_par_indice<Q,i-1>::type type;
   };
```

bla

Explications

```
//
// static_transform<TList, Op>
// - applique Op<T> à chaque T de la liste TList
// - le résultat de Op<T> est un type
// - le résultat de static_transform<TList, Op> est
static_transform<TList, Op>::type,
//   une liste de types correspondant élément par élément aux
types dans TList, une
//   fois ceux-ci transformés par Op
//
// Depends on: nil
//
template <class TList, template <class> class Op>
   struct static_transform;
template <class T, class Q, template <class> class Op>
   struct static_transform<type_list<T, Q>, Op>
   {
      typedef type_list<
         typename Op<T>::type, typename static_transform<Q,
Op>::type
      > type;
   };
template <class T, template <class> class Op>
   struct static_transform<type_list<T, Vide>, Op>
   {
      typedef type_list<
         typename Op<T>::type, Vide
      > type;
   };
template <template <class> class Op>
   struct static_transform<Vide, Op>
   {
      typedef Vide type;
   };
```

bla

Explications

```
//
// static_accumulate<TList, Op, Init>
// - applique Op<T,Q> à chaque élément T de TList, où Q est la
liste suivant
//   T dans TList. Ceci provoque un cumul de valeurs représentées
sous forme
//   de types (probablement des int_<>)
// - Op<T,Q> est typiquement une addition ou un produit statique
// - Init est un type représentant la valeur initiale du cumul à
calculer, typiquement
//   int_<0> pour l'addition et int_<1> pour la multiplication
// - le fruit de Op<T,Q> est un type, probablement un int_<>
// - le fruit de static_accumulate<TList, Op, Init> est
static_accumulate<TList, Op, Init>::type,
//   probablement un int_<>
//
// Depends on: nil
//
template <class TList, template <class, class> class Op, class
Init>
   struct static_accumulate;
template <class T, class Q, template <class, class> class Op, class
Init>
   struct static_accumulate<type_list<T, Q>, Op, Init>
   {
      typedef typename
         Op<T, typename static_accumulate<Q, Op, Init>::type>::type
type;
   };
template <class T, template <class, class> class Op, class Init>
   struct static_accumulate<type_list<T, Vide>, Op, Init>
   {
      typedef typename Op<T, Init>::type type;
   };
```

bla

Explications

```
//
// static_for_each<TList, Op>
//
// Applique l'opération Op<T>() à chaque type T
de la liste TList
//
// Depends on: nil
//
template <class TList>
   struct static_for_each;
template <class T, class Q>
   struct static_for_each<type_list<T, Q> >
   {
      template <class Op>
         static void execute(Op oper)
         {
            oper(T());
            static_for_each<Q>::execute(oper);
         }
   };
template <>
   struct static_for_each<Vide>
   {
      template <class Op>
         static void execute(Op)
            { }
   };
```

bla

Explications

```
//
// static_remove_if<TList, Pred>::type est la
liste contenant
// seulement les éléments T tels que
!Pred<T>::VAL
//
// Depends on: nil
//
template <class TList, template <class> class
Pred>
   struct static_remove_if;
template <class T, class Q, template <class>
class Pred>
   struct static_remove_if<type_list<T, Q>,
Pred>
   {
      typedef typename
         static_if_else <
            Pred<T>::VAL,
            typename static_remove_if<Q,
Pred>::type,
            type_list<T, typename
static_remove_if<Q, Pred>::type>
         >::type type;
   };
template <class T, template <class> class Pred>
   struct static_remove_if<type_list<T, Vide>,
Pred>
   {
      typedef typename
         static_if_else <
            Pred<T>::VAL,
            Vide,
            type_list<T, Vide>
         >::type type;
   };
```

Algorithmes à faibles dépendances

bla

Explications

```
//
// static_sum<TList>
// - représente la somme des éléments de la
liste TList
// - le fruit de static_sum<TList> est
static_sum<TList>::type, un int_<>
//   représentant la valeur de la somme
calculée
//
// Depends on:
// - static_accumulate
// - statc_add
//
template <class TList>
   struct static_sum
   {
      typedef typename
         static_accumulate<
            TList, static_add,
static_add_base::neuter
         >::type type;
   };
```

bla

Explications

```
//
// make_int_list<Debut, Fin>::type est la liste
des types
// int_<Debut>, int_<Debut+1>, ..., int_<Fin>
inclusivement
//
// Depends on:
// - int_
//
#include "Incompilable.h"

template <int Debut, int Fin>
   class debut_plus_gros_que_fin { };
template <int Debut, int Fin>
   struct make_int_list
      : static_if_else<
           (Debut < Fin),
           Compilable,
           Incompilable<
              debut_plus_gros_que_fin<Debut,
Fin>
           >
        >::type
   {
      typedef
         type_list<
            int_<Debut>,
            typename
               make_int_list<Debut + 1,
Fin>::type
         > type;
   };
template <int Fin>
   struct make_int_list<Fin,Fin>
   {
      typedef type_list<int_<Fin>,Vide> type;
   };
```

bla

Explications

```
//
// est_dans<TList,T>::VAL est vrai ssi T est
dans TList
//
// Depends on: indice_par_type
//
template <class TList, class T>
   struct est_dans
   {
      enum { VAL = indice_par_type<TList,
T>::VAL != -1 };
   };
```

bla

Explications

```
//
// Pivoter les éléments d'une liste de types
vers la gauche, de
// manière cyclique (en déplaçant le 1er
élément de la liste à la
// fin)
//
// Depends on: inserer_fin
//
template <class TList>
   struct static_pivoter_gauche;
template <class T, class Q>
   struct static_pivoter_gauche<type_list<T, Q>
>
   {
      typedef typename
         inserer_fin<Q, T>::type type;
   };
template <class T>
   struct static_pivoter_gauche<type_list<T,
Vide> >
   {
      typedef type_list<T, Vide> type;
   };
```

bla

Explications

```
//
// static_merge<TL0, TL1>::type est la liste
comprenant
// les éléments de TL0 suivis des éléments de
TL1
//
// Depends on:
// - static_insert
//
template <class TL0, class TL1>
   struct static_merge;
template <class TList, class T, class Q>
   struct static_merge<TList, type_list<T, Q> >
   {
      typedef typename
         static_merge<
            typename static_insert<TList,
T>::type,
            Q
         >::type type;
   };
template <class TList, class T>
   struct static_merge<TList, type_list<T,
Vide> >
   {
      typedef typename
         static_insert<TList, T>::type type;
   };
template <class TList>
   struct static_merge<TList, Vide>
   {
      typedef TList type;
   };
```

Algorithmes à fortes dépendances

bla

Explications

```
//
// make_list<T,N>::type est une liste
comprenant N occurrences
// du type T
//
// Depends on:
// - Compilable
// - Incompilable
// - static_if_else
//
template <int N>
   class invalid_list_length {};
template <class T, int N>
   struct make_list
      : static_if_else<
           (N <= 0),
           Incompilable<
              invalid_list_length<N>
           >,
           Compilable
        >::type
   {
      typedef type_list<T, typename
make_list<T, N-1>::type > type;
   };
template <class T>
   struct make_list<T, 1>
   {
      typedef type_list<T, Vide> type;
   };
```

bla

Explications

```
//
// static_copy_n<TList,N>::type est la liste
des N premiers
// éléments de TList
//
// Depends on:
// - Compilable
// - Incompilable
// - static_if_else
// - static_length
//
class list_too_short {};
template <class TList, int N>
   struct static_copy_n;
template <class T, class Q, int N>
   struct static_copy_n<type_list<T, Q>, N>
      : static_if_else<
           (N <= static_length<type_list<T, Q>
>::VAL),
           Compilable,
           Incompilable<list_too_short>
        >::type
   {
      typedef type_list<
         T, typename static_copy_n<Q, N-
1>::type
      > type;
   };
template<class T, class Q>
   struct static_copy_n<type_list<T, Q>, 0>
   {
      typedef Vide type;
   };
template<>
   struct static_copy_n<Vide, 0>
   {
      typedef Vide type;
   };
```

bla

Explications

```
//
// static_remove_tail<TList,N>::type est la
liste TList de laquelle
// ont été amputés les N derniers éléments
//
// Depends on:
// - static_copy_n
// - static_length
// - static_if_else
// - Compilable
// - Incompilable
//

template <class TList, int N>
   struct static_remove_tail
      : static_if_else<
           (static_length<TList>::VAL <= N),
           Incompilable<list_too_short>,
           Compilable
         >::type
   {
      typedef typename
         static_copy_n<TList,
static_length<TList>::VAL - N>::type type;
   };
```

bla

Explications

```
//
// static_behead<TList,N>::type équivaut à la liste TList
// de laquelle on aurait amputé les N premiers éléments
//
// Depends on:
// - Compilable
// - Incompilable
// - static_if_else
// - static_length
//
template <class TList, int N>
   struct static_behead;
template <class, int>
   struct static_behead_impl;
template <class T, class Q, int N>
   struct static_behead_impl<type_list<T, Q>, N>
   {
      typedef typename
         static_behead<Q,N-1>::type type;
   };
template <class T, class Q, int N>
   struct static_behead<type_list<T,Q>,N>
      : static_if_else<
           (static_length<type_list<T,Q> >::VAL >= N),
           static_behead_impl<type_list<T, Q>, N>,
           Incompilable<list_too_short>
        >::type
   {
   };
template <class T, class Q>
   struct static_behead<type_list<T, Q>,0>
   {
      typedef type_list<T, Q> type;
   };
template <class T>
   struct static_behead<T,0>
   {
      typedef T type;
   };
```

bla

Explications

```
//
// static_sublist<TList,B,E>::type is the list made from
// the sub list (B..E( in TList
//
// Depends on:
// - Compilable
// - Incompilable
// - static_behead
// - static_if_else
// - static_length
// - static_remove_tail
//
template <int Debut, int Fin>
   class bornes_invalides {};
template <class TList, int B, int E>
   class static_sublist
      : static_if_else<
          (B >= E),
          Incompilable<bornes_invalides<B, E> >,
          typename
             static_if_else<
                ((static_length<TList>::VAL) < (E - B)),
                Incompilable<list_too_short>,
                Compilable
             >::type
        >::type
   {
      typedef typename
         static_behead<
            TList, B
         >::type beheaded_list;
   public:
      typedef typename
         static_copy_n<
            beheaded_list, (E - B)
         >::type type;
   };

#endif
```

Bla

Outils d'ordre général

Quelques outils importants en soi n'entrent pas bien dans les catégories ci-dessus. En voici quelques-uns…

Classe `equivalence`

Bla

Explications

```
namespace relation
{
   //
   // Présume que E == E est défini
   //
   template <class E, class Parent = Vide>
      struct equivalence : Parent
      {
         friend bool operator!=(const E &a, const E &b)
            { return ! (a == b); }
      };
   template <class T>
      class test_equivalence
      {
         T &a_, &b_;
      public:
         test_equivalence(T &a, T &b) throw()
            : a_(a), b_(b)
         {
         }
         friend std::ostream& operator <<
            (std::ostream &os, test_equivalence &te)
         {
            if (te.a_ == te.b_)
               os << te.a_ << " == " << te.b_ << std::endl;
            if (te.a_ != te.b_)
               os << te.a_ << " != " << te.b_ << std::endl;
            return os;
         }
      };
   template <class T>
      test_equivalence<T> CreerTestEquivalence(T &a, T &b)
         { return test_equivalence<T>(a, b); }
}
```

bla

Classe `swappable`

Bla

Explications

```
template <class T, class Parent = Vide>
   struct swappable
      : Parent
   {
      friend void swap (T &a, T &b)
         { a.swap(b); }
   };
```

bla

Extensions à `<algorithm>`

Bla

Explications

```
#include <functional>
#include <algorithm>
#include <utility>
#include <iterator>

template <class Itt, class T>
   bool equal_all(Itt debut, Itt fin, T val)
      { return std::find_if(debut, fin,
std::bind1st(std::not_equal_to<T>(), val)) ==
fin; }
```

Bla

Explications

```
template <class Itt, class D>
   void advance_upto(Itt &p, D n, Itt end) //
Inefficace
      { std::advance(p,
std::min(distance(p,end),n)); }
```

Bla

Explications

```
template <class Itt, class D>
   Itt advance_by(Itt p, D n)
   {
      std::advance(p, n);
      return p;
   }
```

Bla

Explications

```
template <class Itt>
   class find_stride_impl
   {
   public:
      typedef typename
         std::iterator_traits<Itt>::value_type
value_type;
      typedef typename

std::iterator_traits<Itt>::difference_type
stride_type;
   private:
      Itt end_;
      stride_type n_;
      value_type val_;
   public:
      find_stride_impl(stride_type n,
value_type val, Itt end)
         : end_(end), n_(n), val_(val)
      {
      }
      bool operator()(Itt pos) const
      {
         return std::distance(pos, end_) >= n_
&& equal_all(pos, advance_by(pos, n_), val_);
      }
   };

template <class Itt, class D, class V>
   find_stride_impl<Itt> find_stride(D n, V
val, Itt end)
      { return
find_stride_impl<Itt>(n,val,end); }
```

Bla

Outils en lien direct avec le domaine d'application

Quelques outils, enfin, sont en lien direct avec le domaine d'application. On n'a qu'à penser au programme principal, bien entendu, mais il y a plus : du banal (des outils pour afficher plus joliment les tâches et les ordonnancements) et du très riche (des outils pour assurer la validation statique des ordonnancements – la clé de notre démarche!).

Classe `decorateur_tache`

Bla

Explications

```
#ifndef DECORATEUR_TACHE_H
#define DECORATEUR_TACHE_H

#include "Tache.h"
#include <iosfwd>

//
// La classe decorateur<V> fait un affichage décoratif
d'identifiants
// de tâches, remplaçant un identifiant choisi (V dans ce cas-
ci) par
// un symbole marquant l'absence de tâche. Elle sert à
afficher le
// fruit de l'ordonnancement statique de tâches périodiques,
plus bas
//
template <int V>
   class decorateur_tache
   {
      char val_;
   public:
      decorateur_tache(const Tache_::id_type n)
         : val_(n == V? '*' : static_cast<char>(n + '0'))
      {
      }
      friend std::ostream& operator<<(std::ostream &os, const
decorateur_tache &dec)
         { return os << dec.val_; }
   };

#endif
```

bla

Décrire un ordonnancement statique

Bla

Explications

```
template <int N>
   struct decrire_cedule_impl
   {
      template <class CedImpl>
         static void executer(std::ostream &os)
         {
            using std::endl;
            using std::boolalpha;
            using std::noboolalpha;
            os << "\n-=-=-=-=-=-=-=-=-[ Niveau " << N << " ]"
               << "\n\nProchaine liste:\n\t";
            static_for_each<
               typename CedImpl::prochaine_liste
            >::execute(afficher_recedulable(os, "\n\t"));
            os << "\nProchaine liste reordonnancee:\n\t";
            static_for_each<
               typename CedImpl::liste_reordonnancee
            >::execute(afficher_recedulable(os, "\n\t"));
            os << "\nPlus urgent:";
            afficher_recedulable(os, "\n")
               (CedImpl::plus_urgent());
            os << "\nProchain:";
            afficher_recedulable(os, "\n")
               (CedImpl::prochain());
            os << "Prochaine cedule: nb prêts: "
               << CedImpl::NB_PRETS
               << ", cedule suivante ok: "
               << boolalpha
               << static_cast<bool>(CedImpl::PROCHAINE_CEDULE_OK)
               << ", on continue: "
               << static_cast<bool>(CedImpl::VAL)
               << noboolalpha
               << endl;
            decrire_cedule_impl<N-1>::executer<typename
CedImpl::prochaine_cedule_impl>(os);
         }
   };
template <>
   struct decrire_cedule_impl<0>
   {
      template <class>
         static void executer(std::ostream &)
            { }
   };
```

Bla

Explications

```
template <class Ced>
   void decrire_cedule(std::ostream &os)
   {
      using std::endl;
      using std::boolalpha;
      using std::noboolalpha;
      os << "\nRecedulables originaux:\n\t";
      static_for_each<
         typename Ced::recedulables
      >::execute(afficher_recedulable(os, "\n\t"));
      decrire_cedule_impl<2>::executer<typename
Ced::prochaine_etape>(os);
   }
```

bla

Déterminer un ordonnancement statique

Bla

Explications

```
#include "type_list.h"
#include "predicats_statiques.h"
#include "static_algorithms.h"
#include "Tache.h"
#include <list>
#include <vector>
#include <algorithm>
#include <iosfwd>
#include <cassert>
```

Bla

Explications

```
//
// calculer_cycle_majeur<TList>
// - calcule le cycle majeur des tâches dans la liste TList
// - le fruit de calculer_cycle_majeur<TList> est
calculer_cycle_majeur<TList>::VAL,
//   la valeur entière du cycle majeur de TList, exprimé en nombre
d'unités discrètes
//   de temps
//
// Le cycle majeur d'un ensemble de tâches périodiques dont les
paramètres
// (P, C) sont connus a priori est le plus petit commun multiple (PPCM)
des
// périodes (P) de ces tâches.
//
// Le cycle majeur d'un ordonnancement statique de tâches périodiques
est
// le plus petit intervalle d'unités de temps menant à un cycle complet
// d'exécution pour toutes les tâches impliquées. Cela signifie qu'il
n'y
// a pas lieu de prévoir un ordonnancement statique plus long que la
longueur
// du cycle majeur pour l'ensemble des tâches impliquées: en pratique,
ce
// cycle se répétera, tout simplement. On peut dire que le cycle majeur
d'un
// ensemble de tâches est la période de l'ensemble examiné de manière
globale.
//
template <class TList>
   struct calculer_cycle_majeur
   {
      enum
      {
         VAL = typename
            static_accumulate<
               typename static_transform<TList, extract_period>::type,
               static_ppcm,
               typename extract_period<typename
static_head<TList>::type>::type
            >::type::VAL
      };
   };
```

Bla

Explications

```
//
// calculer_cycle_mineur<TList>
// - calcule le cycle mineur des tâches dans la liste TList
// - le fruit de calculer_cycle_mineur<TList> est
calculer_cycle_mineur<TList>::VAL,
//   la valeur entière du cycle mineur de TList, exprimé en nombre d'unités
discrètes
//   de temps
//
// Le cycle mineur d'un ensemble de tâches périodiques dont les paramètres
// (P, C) sont connus a priori est le plus grand commun diviseur (PGCD) des
// périodes (P) de ces tâches.
//
// Le cycle mineur d'un ordonnancement statique de tâches périodiques est
// le plus petit intervalle d'unités de temps au bout duquel une décision
doit
// nécessairement être prise par l'ordonnanceur. Évidemment, le mot
«décision»
// est quelque peu abusif ici, du fait que l'ordonnancement est statique,
mais
// l'idée est que le début d'un cycle mineur est un point sensible pour le
// lancement (Release) d'une tâche périodique; connaître le cycle mineur
permet
// de construire un ordonnancement de manière plus efficace, en réduisant
les
// moments candidats à provoquer un lancement de tâche.
//
// En pratique, une tâche en exécution ne peut chevaucher le moment où on
// passe d'un cycle mineur à un autre; les débuts de cycle mineurs sont
aussi
// des débuts de tâches
//
template <class TList>
   struct calculer_cycle_mineur
   {
      enum
      {
         VAL = typename
            static_accumulate<
               typename static_transform<TList, extract_period>::type,
               static_pgcd,
               typename extract_period<typename
static_head<TList>::type>::type
            >::type::VAL
      };
   };
```

Bla

Explications

```
//
// construct_taskset<TList> permet de construire une list<Tache_> à partir d'une
// liste statique de Tache<P,C>
//
// construct_taskset<TList>::execute() pourrait être optimisé, mais je ne suis
// pas certain que cela en vaille la peine... Simplement créer les Tache_<> dans
// un autre ordre serait une importante optimisation, mais cela attribuerait les
// id en ordre inhabituel (dans la liste, les premiers id seraient à la fin et
// les derniers seraient au début)
//
template <class T>
   struct base_construct_taskset
   {
      enum
      {
         P = typename extract_period<T>::type::VAL,
         C = typename extract_calcul<T>::type::VAL
      };
      static Tache_ make_task()
         { return Tache_(P, C); }
   };
template <class TList>
   struct construct_taskset;
template <class T, class Q>
   struct construct_taskset<type_list<T, Q> >
      : base_construct_taskset<T>
   {
      typedef type_list<T, Q> type;
      static std::list<Tache_> execute()
      {
         std::list<Tache_> lst(1, make_task());
         std::list<Tache_> lst2 = construct_taskset<Q>::execute();
         lst.insert(lst.end(), lst2.begin(), lst2.end());
         return lst;
      }
   };
template <class T>
   struct construct_taskset<type_list<T, Vide> >
      : base_construct_taskset<T>
   {
      typedef type_list<T, Vide> type;
      static std::list<Tache_> execute()
         { return std::list<Tache_>(1, make_task()); }
   };
```

Bla

Explications

```
//
// evaluer_occupation<MAJEUR>
//
// L'évaluation de l'occupation que font les tâches périodiques d'un ordonnanceur
// statique repose sur le temps que chacune consomme dans le cycle majeur.
//
// Pour réaliser ce calcul, il faut, pour chaque tâche:
// - identifier le temps de calcul pour un lancement (un Release); et
// - le multiplier par le nombre d'occurrences de la tâche dans le cycle majeur
//
// Le cycle majeur est une caractéristique globale de l'évaluation, mais il est
// possible de spécialiser l'évaluation en tant que telle pour chaque tâche. Ceci
// explique le recours ici à un template interne, qui servira à titre de foncteur
// statique générique
//
// Le fruit de evaluer_occupation<MAJEUR>::eval<T> pour une tâche T donnée est
// evaluer_occupation<MAJEUR>::eval<T>::type, un int_<> décrivant le temps que
// T occupe dans le cycle majeur de durée MAJEUR.
//
template <int MAJEUR>
   struct evaluer_occupation
   {
      template <class T>
         class eval
         {
            typedef typename extract_period<T>::type periode_type;
            typedef typename extract_calcul<T>::type calcul_type;
         public:
            typedef int_<MAJEUR / periode_type::VAL * calcul_type::VAL> type;
         };
   };
```

Bla

Explications

```
//
// static_evaluer_occupation<TList, MAJEUR>
//
// Évalue le temps total d'occupation des tâches de la liste TList dans un
// cycle majeur de durée MAJEUR.
//
// Le fruit de l'évaluation de static_evaluer_occupation<TList, MAJEUR> est
// static_evaluer_occupation<TList, MAJEUR>::VAL, la valeur entière de la
// somme des occupations que font les tâches de TList dans le cycle majeur
// de durée MAJEUR.
//
// Le taux d'occupation des tâches de la liste TList dans un cycle majeur
// de durée MAJEUR est donné par la méthode taux_occupation(). Ce taux doit
// être inférieur ou égal à 1 pour que la liste de tâches TList puisse
// être ordonnancée en pratique
//
template <class TList, int MAJEUR>
   struct static_evaluer_occupation
   {
      enum
      {
         VAL = typename static_accumulate<
                   typename
                      static_transform<
                         TList, typename evaluer_occupation<MAJEUR>::eval
                      >::type,
                   static_add,
                   int_<0>
                >::type::VAL
      };
      static double taux_occupation() throw()
         { return static_cast<double>(VAL) / MAJEUR; }
   };
```

Bla

Explications

```
typedef int_<0> STATIC_FREE_SPACE;
typedef int_<1> STATIC_USED_SPACE;
```

Bla

Explications

```
template <class TList, class T, int N>
   struct seek_n_consecutive;
template <class T, class Q, int N>
   struct seek_n_consecutive<type_list<T, Q>, T, N>
   {
      enum { VAL = seek_n_consecutive<Q, T, N-1>::VAL };
   };
template <class T, class Q>
   struct seek_n_consecutive<type_list<T, Q>, T, 1>
   {
      enum { VAL = true };
   };
template <class T, class Q, class U, int N>
   struct seek_n_consecutive<type_list<T, Q>, U, N>
   {
      enum { VAL = false };
   };
template <class T, int N>
   struct seek_n_consecutive<Vide, T, N>
   {
      enum { VAL = false };
   };
```

Bla

Explications

```
//
// begins_with_n<TList,T,N>::VAL est vrai seulement si
// on trouve au moins N occurrences de T au début de TList
//
template <class TList, class T, int N>
   struct begins_with_n;
template <class T, class Q, int N>
   struct begins_with_n<type_list<T, Q>, T, N>
   {
      enum
         { VAL = begins_with_n<Q, T, N-1>::VAL };
   };
template <class T, class Q>
   struct begins_with_n<type_list<T, Q>, T, 1>
   {
      enum
         { VAL = true };
   };
template <class T, class Q, class U>
   struct begins_with_n<type_list<T, Q>, U, 1>
   {
      enum
         { VAL = false };
   };
template <class T, class Q, class U, int N>
   struct begins_with_n<type_list<T, Q>, U, N>
   {
      enum
         { VAL = false };
   };
```

Bla

Explications

```
//
// behead<TList,T>::type est la liste TList de laquelle on a
// élagué toutes les occurrences de T au début
//
template <class TList, class T>
   struct behead;
template <class T, class Q>
   struct behead<type_list<T, Q>, T>
   {
      typedef typename
         behead<Q, T>::type type;
   };
template <class T, class Q, class U>
   struct behead<type_list<T, Q>, U>
   {
      typedef type_list<T, Q> type;
   };
template <class T>
   struct behead<Vide, T>
   {
      typedef Vide type;
   };
```

Bla

Explications

```
//
// seek_space<TList, N>::VAL est la position dans TList de la 1re occurrence
// d'une séquence de N instances de STATIC_FREE consécutives, ou -1 si
aucune
// telle séquence n'existe
//
template <class TList, int N>
   struct seek_space
   {
      typedef typename
         behead<TList, STATIC_FREE_SPACE>::type free_space_removed;
      typedef typename
         behead<free_space_removed, STATIC_USED_SPACE>::type next_candidate;
      enum
      {
         VAL = begins_with_n<TList, STATIC_FREE_SPACE, N>::VAL?
                  0 : seek_space<next_candidate, N>::VAL == -1?
                  -1 : seek_space<next_candidate, N>::VAL +
(static_length<TList>::VAL - static_length<next_candidate>::VAL)
      };
   };
template <int N>
   struct seek_space<Vide, N>
   {
      enum { VAL = -1 };
   };
```

Bla

Explications

```
//
// TList est une liste de int_<N> pour différents N
//
template <class TASK, class TList>
   class static_can_schedule;
template <class TASK, class T, class Q>
   class static_can_schedule<TASK, type_list<T, Q> >
   {
   public:
      enum
      {
         CALCUL = typename extract_calcul<TASK>::type::VAL,
         PERIODE = typename extract_period<TASK>::type::VAL
      };
      enum
      {
         POS_FIRST_FREE = seek_space<type_list<T,Q>, CALCUL>::VAL
      };
      typedef typename
         static_if_else<
            (POS_FIRST_FREE != -1),
            typename
               static_behead<
                  type_list<T,Q>,POS_FIRST_FREE
               >::type,
            Vide
         >::type remaining_list;
      enum
      {
         NB_FREE = POS_FIRST_FREE == -1?
            0 : count_consecutive<
                   remaining_list, STATIC_FREE_SPACE
                >::VAL
      };
      enum { VAL = NB_FREE >= CALCUL };
   };
```

Bla

Explications

```
template <class T, class Sched>
   struct static_schedule
   {
      // Constantes décrivant la tâche en cours d'ordonnancement
      enum
      {
         CALCUL = typename extract_calcul<T>::type::VAL,
         PERIODE = typename extract_period<T>::type::VAL
      };
      // Ordonnancer une tâche <=> indiquer qu'elle est prise. Ce foncteur
```

```
      // statique est utilisé par static_transform pour remplacer une instance
      // de STATIC_FREE_SPACE par une instance de
      template <class>
         struct schedule
         {
            typedef STATIC_USED_SPACE type;
         };
      // current_period est la période dans laquelle l'ordonnancement en cours
      // doit se situer
      typedef typename
         static_copy_n<Sched, PERIODE>::type current_period;
      // remaining_periods est l'espace restant pour fins d'ordonnancement
      typedef typename
         static_behead<Sched, PERIODE>::type remaining_periods;
      // Trouver le premier moment libre dans la période en cours
      enum
      {
         POS_FIRST_FREE =
indice_par_type<current_period,STATIC_FREE_SPACE>::VAL
      };
      // Ordonnancer T dans la période courante
      typedef typename
         static_sublist<
            current_period, POS_FIRST_FREE, POS_FIRST_FREE+CALCUL
         >::type mid_part_pre_schedule;
      typedef typename
         static_copy_n<
            current_period,POS_FIRST_FREE
         >::type beginning_part;
      typedef typename
         static_transform<
            mid_part_pre_schedule, schedule
         >::type middle_part;
      typedef typename
         static_behead<
            current_period,POS_FIRST_FREE+CALCUL
         >::type end_part;
      typedef typename
         static_merge<
            beginning_part,
            typename
               static_merge<
                  middle_part,
                  end_part
               >::type
         >::type scheduled_part;
      typedef typename
```

```
         static_merge<
            scheduled_part,
            typename
               static_schedule<T, remaining_periods>::type
         >::type type;
   };
template <class T>
   struct static_schedule<T, Vide>
   {
      typedef Vide type;
   };
```

Bla

Explications

```
template <class TaskList, class Sched>
   class construire_cedule_possible;
template <class T, class Q, class Sched>
   class construire_cedule_possible<type_list<T, Q>, Sched>
   {
   public:
      typedef typename
         static_schedule<T,Sched>::type scheduled_list;
      enum { T_SCHEDULABLE = static_can_schedule<T,Sched>::VAL };
      enum
      {
         VAL = T_SCHEDULABLE && // requires lazy eval
               construire_cedule_possible<
                  Q, scheduled_list
               >::VAL
      };
   };
template <class T, class Sched>
   class construire_cedule_possible<type_list<T, Vide>, Sched>
   {
   public:
      typedef typename
         static_schedule<T,Sched>::type scheduled_list;
      enum
      {
         VAL = static_can_schedule<T,Sched>::VAL
      };
   };
```

Bla

Explications

```
template <class TList, class Sched>
   struct static_scheduling;
template <class T, class Q, class Sched>
   struct static_scheduling<type_list<T, Q>,Sched>
   {
      typedef typename
         static_schedule<
            T, Sched
         >::type sched_head_type;
      typedef typename
         static_scheduling<Q, sched_head_type>::type type;
   };
template <class T, class Sched>
   struct static_scheduling<type_list<T, Vide>,Sched>
   {
      typedef typename
         static_schedule<
            T, Sched
         >::type type;
   };
```

Bla

Explications

```
template <class TList>
   class ordonnancable
   {
   public:
      typedef typename
         make_list<
            STATIC_FREE_SPACE,
            calculer_cycle_majeur<TList>::VAL
         >::type empty_schedule;
      enum { VAL =
construire_cedule_possible<TList,empty_schedule>::VAL };
      typedef typename
         construire_cedule_possible<
            TList,empty_schedule
         >::scheduled_list once_scheduled;
      typedef typename
         static_if_else<
            VAL,
            typename static_scheduling <
               TList, empty_schedule
            >::type,
            Vide
         >::type type;
   };
```

Bla

Explications

```
template <int MINEUR, int MAJEUR>
   class cedule_statique
   {
   public:
      typedef std::vector<Tache_::id_type> container_type;
   private:
      container_type v_;
   public:
      typedef container_type::value_type value_type;
      typedef container_type::iterator iterator;
      typedef container_type::const_iterator const_iterator;
      typedef container_type::reverse_iterator reverse_iterator;
      typedef container_type::reverse_iterator const_reverse_iterator;
      iterator begin() throw()
         { return v_.begin(); }
      iterator end() throw()
         { return v_.end(); }
      const_iterator begin() const throw()
         { return v_.begin(); }
      const_iterator end() const throw()
         { return v_.end(); }
      reverse_iterator rbegin() throw()
         { return v_.rbegin(); }
      reverse_iterator rend() throw()
         { return v_.rend(); }
      const_reverse_iterator rbegin() const throw()
         { return v_.rbegin(); }
      const_reverse_iterator rend() const throw()
         { return v_.rend(); }
   private:
      enum { FREE_SPACE = -1 };
      //
      // can_schedule(t, debut, fin) retourne vrai ssi le temps
      // de calcul de t ne dépasse pas distance(debut, fin) et
      // si la plage (debut..debut + t.calcul()( est disponible
      //
      template <class Itt>
         bool can_schedule(Tache_ t, Itt debut, Itt fin) const
         {
            if (distance(debut, fin) < t.calcul())
               return false;
            Itt p = debut;
            advance(p, t.calcul());
            return equal_all(debut, p,
static_cast<value_type>(FREE_SPACE));
         }
      template <class Itt, class ItPred>
```

```
         Itt can_schedule_first(Tache_ t, Itt start_pos, Itt end_pos,
ItPred it_pred) const
         {
            for (Itt p = start_pos; p != end_pos; ++p) // inefficace
               if (it_pred(p)) return p;
            return end_pos;
         }
      template <class Itt>
         bool can_repeat_schedule(Tache_ t, Itt start_pos, Itt end_pos)
const
         {
            while (can_schedule(t, start_pos, end_pos))
               advance_upto(start_pos, t.periode(), end_pos);
            return std::distance(start_pos, end_pos) < t.periode();
         }
      const_iterator can_schedule(Tache_ t) const
      {
         using std::find;
         using std::find_if;
         using std::not_equal_to;
         //
         //
         //
         const_iterator first_sched_threshold = advance_by(begin(),
t.periode());
         const_iterator pos = can_schedule_first(
            t, begin(), first_sched_threshold, find_stride(t.calcul(),
FREE_SPACE, first_sched_threshold)
         );
         while (pos != first_sched_threshold && !can_repeat_schedule(t,
pos, end()))
         {
            pos = find_if(pos, first_sched_threshold,
                          bind1st(not_equal_to<value_type>(),
FREE_SPACE));
            if (pos == first_sched_threshold) return end();
            pos = find(pos, first_sched_threshold, FREE_SPACE);
            pos = can_schedule_first(
               t, pos, first_sched_threshold, find_stride(t.calcul(),
FREE_SPACE, first_sched_threshold)
            );
         }
         return pos != first_sched_threshold? pos : end();
      }
      cedule_statique &unconst_self() const
         { return *const_cast<cedule_statique*>(this); }
      const cedule_statique &const_self()
```

```
         { return *const_cast<const cedule_statique*>(this); }
   public:
      static int cycle_mineur() throw()
         { return MINEUR; }
      static int cycle_majeur() throw()
         { return MAJEUR; }
      cedule_statique()
         : v_(MAJEUR, FREE_SPACE)
      {
      }
      bool operator()(Tache_ t)
      {
         using std::advance;
         using std::distance;
         using std::fill_n;
         const_iterator pos = can_schedule(t);
         if (pos == end()) return false;
         const int NB_FIRES = MAJEUR / t.periode();
         iterator p = begin();
         advance(p, distance(const_self().begin(), pos));
         for (int i = 0; i < NB_FIRES; ++i, advance_upto(p,
t.periode(), end()))
            fill_n(p, t.calcul(), t.id());
         return true;
      }
      friend std::ostream&
         operator<<(std::ostream &os, const cedule_statique &p)
      {
         using std::advance;
         using std::copy;
         using std::endl;
         using std::ostream_iterator;
         for (cedule_statique::const_iterator itt = p.begin(); itt !=
p.end(); advance(itt, p.cycle_mineur()))
         {
            cedule_statique::const_iterator fin = itt;
            advance(fin, p.cycle_mineur());
            copy (itt, fin,
ostream_iterator<decorateur_tache<FREE_SPACE> >(os, " "));
            os << endl;
         }
         return os;
      }
   };
```

bla

Programme principal

Bla

Explications

```
#include "decorateur_tache.h"
#include "algorithm_ext.h"
#include "cedule_statique.h"
#include "Tache.h"
#include "predicats_statiques.h"
#include "type_list.h"
#include "static_assert.h"
#include "traits_types.h"
#include "decrire_cedule.h"
#include "outils_numeriques_statiques.h"
#include <algorithm>
#include <iterator>
#include <iostream>
#include <cassert>
#include <vector>
#include <iosfwd>
#include <list>
```

Bla

Explications

```
class afficher_int_
{
   std::ostream &os_;
   const char *delim_;
public:
   afficher_int_(std::ostream &os, const char
*delim = "")
      : os_(os), delim_(delim)
   {
   }
   template <class T>
      void operator()(T)
         { os_ << T::VAL << delim_; }
};
```

Bla

Explications

```
template<class TList>
   void ceduler(construct_taskset<TList> taskset_constructor)
   {
      using std::copy;
      using std::cout;
      using std::list;
      using std::endl;
      using std::ostream_iterator;
      const int CYCLE_MINEUR = calculer_cycle_mineur<TList>::VAL;
      const int CYCLE_MAJEUR = calculer_cycle_majeur<TList>::VAL;
#ifdef _DEBUG
```

```
      cout << "Cycle mineur: " << CYCLE_MINEUR << endl
           << "Cycle majeur: " << CYCLE_MAJEUR << endl;
#endif
      //
      // Le taux d'occupation du task set est-il inférieur à 1? C'est
une condition
      // nécessaire (mais pas suffisante) pour que le task set puisse
être cédulé...
      //
      static_assert<
         (static_evaluer_occupation<TList, CYCLE_MAJEUR>::VAL <=
CYCLE_MAJEUR)
      > le_task_set_ne_peut_etre_cedule_cause_occupation_superieure_a_1;

unused(le_task_set_ne_peut_etre_cedule_cause_occupation_superieure_a_1);
      static_assert<
         (ordonnancable<TList>::VAL)
      > impossible_de_construire_une_cedule_periodique_statique;
      unused(impossible_de_construire_une_cedule_periodique_statique);
#ifdef _DEBUG
      cout << "Occupation theorique du task set: "
           << static_evaluer_occupation<TList,
CYCLE_MAJEUR>::taux_occupation() * 100
           << '%' << endl;
#endif
      //
      // Générer la liste des tâches
      //
      list<Tache_> lst = taskset_constructor.execute();
#ifdef _DEBUG
      cout << "Taches a couvrir (P, C), portrait dynamique :" << endl <<
'\t';
      copy(lst.begin(), lst.end(), ostream_iterator<Tache_>(cout,
"\n\t"));
      cout << endl;
#endif
      cedule_statique<CYCLE_MINEUR, CYCLE_MAJEUR> ced;
      for (list<Tache_>::iterator itt = lst.begin(); itt != lst.end();
++itt)
         if (!ced(*itt))
         {
            std::cerr << "incapable d'ordonnancer " << *itt << endl;
            break;
         }
      cout << ced << endl;
   }
```

Bla

Explications

```
template <class TList>
   void tester_ceduleur_statique()
   {
      using std::cout;
      using std::endl;
      cout << "\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n"
           << "Taches a ordonnancer (P, C), portrait statique :\n\t";
static_for_each<TList>::execute(afficher_tache(cout, "\n\t"));
      cout<< endl;
      //
      // Construire une cédule
      //
      ceduler(construct_taskset<TList>());
   }
```

Bla

Explications

```
template <class TList>
   void describe_schedule(std::ostream &os)
   {
      using std::endl;
      typedef typename
         static_head<TList>::type current_task;
      typedef typename
         ordonnancable<
            TList
         >::empty_schedule base_schedule;
      typedef typename
         ordonnancable<
            TList
         >::type result_schedule;
      typedef static_schedule<
         current_task, base_schedule
      > ongoing_scheduling;
      //
      //
      //
      static_for_each<
         base_schedule
      >::execute(afficher_int_(os, " "));
      os << endl;
      static_for_each<
         typename ongoing_scheduling::type
      >::execute(afficher_int_(os, " "));
      os << endl;
```

```
      os <<
         static_length<
            typename
ongoing_scheduling::beginning_part
         >::VAL
          << ','
          <<
         static_length<
            typename
ongoing_scheduling::middle_part
         >::VAL
          << ','
          <<
         static_length<
            typename
ongoing_scheduling::end_part
         >::VAL
          << '('
          <<
         static_length<
            typename
ongoing_scheduling::remaining_periods
         >::VAL
          << ')'
          << endl;
      static_for_each<
         result_schedule
      >::execute(afficher_int_(os, " "));
      os << endl;
   }
```

Bla

Explications

```
int main()
{
   //
   // L'ensemble de tâches périodiques
statiques que nous souhaitons ordonnancer
   //
   typedef type_list<
      Tache<25, 10>, type_list<
         Tache<25, 8>, type_list<
            Tache<50, 5>, type_list<
               Tache<50, 4>, type_list<
                  Tache<100, 2>, Vide
               >
            >
         >
      >
```

```
   > task_set_11_1;
   typedef type_list<
      Tache<6, 4>, type_list<
         Tache<12, 3>, Vide
      >
   > task_set_test;
   typedef type_list<
      Tache<6, 4>, Vide
   > task_set_banal;
   typedef type_list<
      Tache<5, 5>, Vide
   > task_set_limite;
   typedef type_list<
      Tache<5, 6>, Vide
   > task_set_illegal;



   using std::cout;
   using std::endl;
/*
   typedef task_set_11_1 ze_task_set;

   describe_schedule<ze_task_set>(cout);

   // ICI: on cherche à savoir pourquoi
!ordonnancable<ze_task_set>::VAL
   cout << "-----" << endl;
   typedef
      static_head<ze_task_set>::type task_0;
   typedef
      static_tail<ze_task_set>::type
remaining_tasks_0;
   typedef

ordonnancable<ze_task_set>::empty_schedule
pre_sched_0;
   typedef static_schedule<
      task_0, pre_sched_0
   >::type sched_0;
   static_for_each<
      sched_0
   >::execute(afficher_int_(cout, " "));
   cout << endl << endl;
   cout << construire_cedule_possible<
               ze_task_set, pre_sched_0
            >::T_SCHEDULABLE << endl << endl;
   //
```

```
//
//
typedef static_head<
   remaining_tasks_0
>::type task_1;
typedef static_tail<
   remaining_tasks_0
>::type remaining_tasks_1;
typedef
   construire_cedule_possible<
      remaining_tasks_0,sched_0
   >::scheduled_list pre_sched_1;
typedef
   static_schedule<
      task_1, sched_0
   >::type sched_1;
static_for_each<
   sched_1
>::execute(afficher_int_(cout, " "));
cout << endl << endl;
cout << construire_cedule_possible<
           remaining_tasks_0, sched_0
        >::T_SCHEDULABLE << endl << endl;
//
//
//
typedef
   static_head<
      remaining_tasks_1
   >::type task_2;
typedef
   static_tail<
      remaining_tasks_1
   >::type remaining_tasks_2;
typedef
   construire_cedule_possible<
      remaining_tasks_1,sched_1
   >::scheduled_list pre_sched_2;
typedef
   static_schedule<
      task_2, sched_1
   >::type sched_2;
static_for_each<
   sched_2
>::execute(afficher_int_(cout, " "));
cout << endl << endl;
cout << construire_cedule_possible<
```

```
            remaining_tasks_1, pre_sched_1
         >::T_SCHEDULABLE << endl << endl;
   //
   //
   //
   typedef
      static_head<
         remaining_tasks_2
      >::type task_3;
   typedef
      static_tail<
         remaining_tasks_2
      >::type remaining_tasks_3;
   typedef
      construire_cedule_possible<
         remaining_tasks_2,sched_2
      >::scheduled_list pre_sched_3;
   typedef
      static_schedule<
         task_3, sched_2
      >::type sched_3;
   static_for_each<
      sched_3
   >::execute(afficher_int_(cout, " "));
   cout << endl << endl;
   //
   // ICI: pourquoi est-ce 0?
   //
   cout << construire_cedule_possible<
            remaining_tasks_2, pre_sched_2
         >::T_SCHEDULABLE << endl << endl;
   typedef static_can_schedule<
      static_head<remaining_tasks_2>::type,
pre_sched_2
   > problem_case;
   cout << "Problem static_schedule: \n"
        << "\tTache {CALCUL, PERIODE}: {" <<
problem_case::CALCUL << ", " <<
problem_case::PERIODE << "}\n"
        << "\tPOS_FIRST_FREE: " <<
problem_case::POS_FIRST_FREE << "\n"
        << "\tNB_FREE: " <<
problem_case::NB_FREE << "\n"
        << "\tBIG_PROBLEM: " <<
problem_case::BIG_PROBLEM << "\n"
        << "\tVAL: " << problem_case::VAL <<
endl << endl;
```

```
   //
   //
   //
   typedef
      static_head<
         remaining_tasks_3
      >::type task_4;
   typedef
      static_tail<
         remaining_tasks_3
      >::type remaining_tasks_4;
   typedef
      construire_cedule_possible<
         remaining_tasks_3,sched_3
      >::scheduled_list pre_sched_4;
   typedef
      static_schedule<
         task_4, sched_3
      >::type sched_4;
   static_for_each<
      sched_4
   >::execute(afficher_int_(cout, " "));
   cout << endl << endl;
   cout << "-----" << endl;
   */
   tester_ceduleur_statique<task_set_11_1>();
   //tester_ceduleur_statique<task_set_test>();
// long à détecter
   tester_ceduleur_statique<task_set_banal>();
   tester_ceduleur_statique<task_set_limite>();

//tester_ceduleur_statique<task_set_illegal>();
// immédiat à détecter
}
```

bla

Tracer un lien avec les exécutifs.

Ordonnancement basé sur les tâches

Exécutifs : peu flexibles, et d'autres défauts :

- ne se prêtent pas aux tâches sporadiques ou apériodiques;
- les tâches à longue période compliquent l'établissement d'un cycle majeur;
- pour cette raison, les tâches longues doivent y être découpées en sous-tâches plus petites, mais cela complique la conception du système (séquencement des sous-tâches).

Pour une cédule de tâches, si on évite la question de la communication entre les tâches, les états possibles d'une tâche sont :

- prêt à être exécuté (*Runnable*);
- suspendu en attente d'un événement temporel (typiquement associé à l'horloge) : périodique;
- suspendu en attente d'un événement non-temporel : sporadique ou apériodique.

Stratégies d'ordonnancement

- à priorité fixe (statique), déterminée *a priori* (en anglais : *Fixed Priority Scheduling*, ou FPS). Dans un STR, la priorité d'une tâche dépend des contraintes TR qu'elle doit respecter, pas de la relative importance de son travail dans le système;
- première échéance d'abord (en anglais : *Earliest Deadline First*, ou EDF). Typiquement, les échéances relatives varient avec le temps, ce qui mène à un schème d'ordonnancement dynamique;
- ordonnancement basé sur les valeurs (en anglais : *Value-Based Scheduling*, ou VBS), plus ou moins bien nommé, où l'ordonnanceur cherche à appliquer de l'intelligence dans la stratégie d'ordonnancement en quantifiant l'importance des tâches à exécuter, typiquement parce qu'il est susceptible d'être débordé.

Chacune peut être appliquée à des tâches préemptives ou non.

Tests d'ordonnançabilité

On veut savoir si une cédule est possible à partir de certaines caractéristiques. On fait alors un **test d'ordonnançabilité**. Un tel test est :

- **suffisant** si, quand le test est positif, cela signifie que les échéances et les contraintes de toutes les tâches seront rencontrées; et
- **nécessaire** si, quand le test est négatif, cela signifie qu'au moins une des tâches ne rencontrera pas ses contraintes.

Un test à la fois nécessaire et suffisant est dit **exact**. Pour bien des problèmes, identifier un test exact est *Intractable* au sens anglais du terme : soit indécidable, soit démesurément complexe sur le plan algorithmique.

Un test d'ordonnançabilité est typiquement appliqué au pire cas des tâches pour un système donné. Un test est **durable** (*Sustainable*) s'il continue à bien prédire l'ordonnançabilité d'un système de tâches quand les conditions s'améliorent.

Quelques a priori

Pour l'étude, ordonnancement simple, quelques *a priori* :

- exécution sur un seul processeur;
- pire cas d'exécution connu a priori pour chaque tâche;
- le changement de contexte d'une tâche à l'autre est présumé négligeable;
- les tâches sont indépendantes les unes des autres;
- l'ensemble des tâches est fermé;
- l'échéance d'une tâche est égale à sa période (*échéance implicite*; une échéance *explicite* est aussi possible).

L'indépendance des tâches implique un lancement (*Release*) à un moment précis de toutes les tâches. Ce moment est **l'instant critique** du système.

Résumé de quelques termes clés pour l'ordonnancement dans les STR

Tâches périodiques, **apériodiques** et **sporadiques** : une ***tâche périodique*** doit être démarrée à rythme régulier ou constant. Une ***tâche apériodique*** est soumise à des contraintes TR autres que celles du rythme de démarrage (par exemple, une échéance à respecter – brièveté). Une ***tâche sporadique*** survient à divers moments (événements physiques, par exemple) et peut être soumise à des contraintes TR, mais intégrée à un STR, elle peut compliquer les tests d'ordonnançabilité.

Quelques termes clés de vocabulaire :

- on nommera **échéance** (en anglais, *Deadline*) le seuil le plus tardif au moment duquel une tâche peut avoir été complétée. Si une tâche dans un STR manque son échéance, son résultat n'est pas valide. Dans un STR strict, manquer une échéance peut être catastrophique. Dans la notation usuelle, le symbole D est associé à ce paramètre;
- on nommera **instant critique** d'un ordonnancement le moment où toutes les tâches seraient prêtes à s'exécuter de manière simultanée. Pour plusieurs stratégies d'ordonnancement, en particulier celles de la famille des ordonnanceurs FPS (plus bas), il s'agit du pire cas de charge pour le système.
- on nommera **période** d'une tâche périodique (évidemment) le rythme auquel cette tâche devra être lancée. *Une tâche périodique doit être ordonnancée à rythme constant*. Dans la notation usuelle, le symbole T est associé à ce paramètre;
- enfin, on utilisera souvent l'acronyme **WCET** qui signifie *Worst Case Execution Time*. Il s'agit d'un élément clé des calculs associés à l'ordonnancement dans les STR. Dans la notation usuelle, le symbole C est associé à ce paramètre.

Il existe de grandes familles d'ordonnanceurs, parmi lesquelles on trouve en particulier :

- celle des ordonnanceurs **DPS**, pour *Dynamic Priority Scheduling*, qui couvre les cas comme celui de l'ordonnancement EDF (plus loin), en supportant des stratégies d'ordonnancement susceptibles d'être validées même quand les priorités changent en cours de route; et
- celle des ordonnanceurs **FPS**, pour *Fixed Priority Scheduling*, qui couvre les cas comme RMS ou DMPO (plus loin) où les priorités des tâches sont fixées *a priori*.

Parmi les algorithmes d'ordonnancement de tâches TR les plus connus, on trouve :

- l'algorithme **DMPO**, pour *Dealing Monotonic Priority Ordering*. DMPO est un algorithme de la famille FPS, où chaque tâche doit avoir une échéance inférieure ou égale à sa période, de même qu'un WCET inférieur ou égal à son échéance (donc $D < T \wedge C < T$). Les tâches sont indépendantes, ne se bloquent pas mutuellement et ne se suspendent pas elles-mêmes. Le coût du passage d'une tâche à l'autre est présumé nul, et la latence de démarrage est aussi présumée nulle;
- l'algorithme **EDF**, pour *Earliest Deadline First*. Avec EDF, on suppose $D = T$. EDF est de la famille DPS, et supporte les priorités dynamiques. Avec cet algorithme, la tâche la plus proche de son échéance est la prochaine à être ordonnancée – conséquemment, l'échéance de chaque tâche doit être connue *a priori*;
- l'algorithme **RMS**, pour *Rate Monotonic Scheduling*. Cet algorithme est très répandu du fait qu'il trouve les priorités de manière optimale si $D = T$. Les contraintes de l'algorithme RMS

sont semblables à celles de l'algorithme DMPO. Avec RMS, chaque tâche a une priorité unique, distincte des autres, basée sur sa période (plus la période est basse, plus la priorité est haute). Les priorités sont fixées *a priori*.

Résumé des symboles clés

Notez que, de manière générale, quand on parlera d'une tâche sans la situer dans un ensemble, on omettra les indices de tâches (p. ex. : on écrira T pour parler de la période d'une tâche et C pour parler de son pire temps d'exécution, plutôt que d'écrire T_i et C_i pour référer à la tâche i).

B_i	**Blocage** de la tâche i, dû à des verrous, des entrées/ sorties, de l'inversion de priorités, *etc.*
C_i	`WCET` de la tâche i, typiquement obtenu de manière analytique (pensez à ***computing*** ou à ***calcul***)
C_i^f	Coût associé au code de tolérance aux pannes (pensez à ***fault***) pour la tâche i, ce qui tient compte entre autres de la gestion des cas d'exceptions
$C(k)$	Pire temps d'exécution dans la section **critique** k (ici, section critique est toute zone susceptible de provoquer un blocage en attente d'une ressource)
D_i	Échéance (pensez à ***deadline***) pour démarrer la tâche sporadique i.
$hp(i)$	Ensemble des tâches $j: P_i < P_j$, donc de plus **haute priorité** que celle de la tâche i. En pratique, dans un `SETR`, seules ces tâches peuvent interférer avec la tâche i.
$hep(i)$	Ensemble des tâches $j: P_i \leq P_j$, donc de priorité supérieure ou égale (***higher-than or equal***) à cette de la tâche i. En pratique, dans un `SETR`, seules ces tâches peuvent interférer avec la tâche i.
I_i	**Interférence** maximale de la tâche i. Notez que la tâche la plus prioritaire n'en aura pas, alors que les autres en auront, par définition. Par I_i, on entend les cas où une tâche $j: P_i < P_j$ empêcherait, de par son exécution, l'ordonnancement de la tâche i
J_i	Ce qu'on nomme le *Release* ***Jitter*** de la tâche i, et qui intervient quand la tâche i est lancée par une tâche $j: j \neq i$ et qui représente le pire cas de latence entre la demande de lancement et le lancement effectif
N	**Nombre** de tâches à considérer (cardinalité de l'ensemble des tâches)
R_i	Temps maximal d'exécution (***Release time***) de la tâche i lorsque l'interférence I_i est considérée dans le calcul. On considère aussi parfois le blocage B_i
P_i	**Priorité** de la tâche i
T_i	Période (je n'ai pas de bonne idée pour les mnémoniques ici; désolé!) de la tâche i
T_f	Intervalle minimal entre deux pannes, lorsque du code de tolérance aux pannes doit être considéré
$usage(k, i)$	Fonction booléenne évaluée à 1 seulement si la ressource k est utilisée par au moins une tâche de priorité $j: P_i > P_j$

Ordonnancement à priorité fixe

Il existe un algorithme pour trouver les priorités de manière optimale, et cet algorithme est dit ***Rate Monotonic Scheduling*** (ou **RMS**; je dois avouer que je ne connais pas le terme français) : chaque tâche a une priorité unique, distincte des autres, basée sur sa période (plus basse période implique plus haute priorité).

Un test possible d'ordonnançabilité est (Liu et Layland 1973) :

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right) \leq N\left(2^{1/N} - 1\right)$$

où N est le nombre de tâches, C_i est le pire temps d'exécution (*Worst Case Execution Time*, ou WCET) de la tâche i, et T_i est la période de la tâche i. Le seuil $N(2^{1/N} - 1)$ converge vers 0,693 pour de grandes valeurs de N.

On peut tracer un diagramme de Gantt des tâches pour visualiser les lancements de chacune (avec préemption et tout le tralala s'il le faut); la longueur de l'intervalle de temps requis pour avoir confiance est la longueur de la plus longue période : à partir de l'instant critique du système, si toutes les tâches rencontrent leur 1re échéance, alors elles rencontreront aussi leurs échéances ultérieures.

Ce test est suffisant mais pas nécessaire. Il existe des ensembles de tâches qui échouent ce test mais sont tout de même ordonnançables (par exemple $\{T_a=80, C_a=40, \text{Prio}=1\}$, $\{T_b=40, C_b=10, \text{Prio}=2\}$, $\{T_c=20, C_c=5, \text{Prio}=3\}$ présente un taux d'occupation total de 100% mais est ordonnançable).

Raffinement : grouper les tâches en *familles* (dans une famille, les périodes des tâches sont des multiples l'une de l'autre), et garder la même équation. Cela réduit N.

Autre raffinement : évaluer (Bini 2007) $\prod_{i=1}^{N}\left(\left(\frac{C_i}{T_i}\right) + 1\right) \leq 2$. Le calcul est plus simple, mais ça reste suffisant sans être nécessaire.

Analyse du temps de réponse

- tests en deux étapes : évaluation analytique du WCET (le facteur C) de chaque tâche, puis comparaisons à la pièce;
- tient compte des interférences (la tâche la plus prioritaire n'en a pas, les autres en ont);
- pour une tâche i, le temps maximal d'exécution requis (A) est $R_i = C_i + I_i$ où I_i est le pire cas possible d'interférence;
- pour une tâche j de priorité supérieure à celle de la tâche i, le nombre de lancements dans un intervalle $[0..R_i)$ sera $\left\lceil \frac{R_i}{T_j} \right\rceil$ et l'interférence maximale sera $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$;
- de manière générale, puisque toutes les tâches plus prioritaires que la tâche i (l'ensemble $hp(i)$) interfèrent avec la tâche i, le calcul de I_i est $I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$;
- l'équation générale pour identifier R_i est, en substituant ce qui précède pour I_i dans (A), $R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$ mais **c'est horrible à résoudre** dû à la présence de R_i des deux côtés de l'équation et dû à la fonction plafond. **L'équation a plusieurs réponses correctes** et **le minimum est la valeur recherchée**.

On peut résoudre l'équation par récurrence, en y allant de la tâche la plus prioritaire (sans interférences) à la moins prioritaire.

Tâches sporadiques et apériodiques

- on adapte le modèle récurrent ci-dessus en considérant T (la période de la tâche) comme l'intervalle minimal entre deux occurrences de l'événement mesuré (en général, c'est une vision pessimiste; certains préfèrent évaluer le temps moyen);
- les tâches sporadiques ont en général une échéance D lors de l'invocation (contrainte de brièveté). L'équation ci-dessus, de manière naïve, peut considérer $D = T$, mais en général les cas $D < T$ et $D > T$ sont possibles;
- **avec $D = T$, l'approche RMS est optimale**.

Apériodicité et $D < T$

- il a été démontré (Leung et Whitehead, 1982) que, pour des tâches sporadiques ou apériodiques, quand $D < T$, l'approche ***Deadline Monotonic Priority Ordering*** (**DMPO**) est optimale;
- avec DMPO, $D_i < D_j \Rightarrow P_i > P_j$;
- une approche RMS pourrait ordonnancer les tâches que peut ordonnancer une approche DMPO.

Interactions et blocage

- supposer l'indépendance des tâches et l'absence d'interactions entre tâches n'est pas raisonnable (verrous, messages; inversion de priorités et autres);
- calculer le blocage maximal d'une tâche i dans un système où K verrous existent est relativement simple. On parle de $B_i = \sum_{k=1}^{K} usage(k,i)C(k)$ où $usage(k,i)$ est une fonction booléenne retournant 0 ou 1 et où $usage(k,i) = 1$ seulement si la ressource k est utilisée par au moins une tâche de priorité inférieure à celle de la tâche i et par au moins une tâche de priorité supérieure à celle de la tâche i. Dans l'équation, $C(k)$ est le pire temps d'exécution dans la section critique k. On suppose que les ressources ne sont pas imbriquées (ou que $usage(k,i)$ se charge du calcul);
- considérant le blocage, l'équation générale pour le temps de réponse associé à une tâche devient $R = C + B + I$;
- ceci signifie que $R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$ ce qui est pessimiste, mais bon.

On peut aussi considérer ce qu'on nomme le ***Release Jitter*** (*J*) d'une tâche sporadique quand elle est susceptible d'être lancée par une autre tâche (typiquement distante) et qu'il y a une latence entre la demande de lancement et le lancement effectif (J_i est alors le pire cas de latence pour le lancement de la tâche *i*).

L'équation en tenant compte devient :

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j.$$

Coût de la tolérance aux pannes

Pour gérer le **coût de la tolérance aux pannes** dans le calcul du temps de réponse d'un système, on doit considérer le coût C_i^f du code de tolérance aux pannes de la tâche i. On inclut ici ce qui a trait aux exceptions, par exemple, et on s'intéresse bien sûr habituellement au pire cas.

En omettant le coût du *Release Jitter*, on obtient l'équation :

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} C_k^f$$

avec $hep(i)$ représentant l'ensemble des tâches de priorité supérieure ou égale à celle de la tâche i. On peut ajouter un facteur F au poids de C_k^f si plusieurs pannes sont susceptibles d'être traitées en séquence.

S'il existe un intervalle minimal T_f entre deux pannes, l'équation devient :

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j + \max_{k \in hep(i)} \left(\left\lceil \frac{R_i}{T_f} \right\rceil C_k^f \right)$$

Choix des priorités

- trouver un schème optimal de priorités pour un ensemble de tâches n'est pas banal;
- un théorème (Audsley, 1993b) indique (je paraphrase) que « si une tâche *P* se voit apposée la plus faible priorité et est faisable dans le respect de ses contraintes, alors s'il existe un ordonnancement de priorités pour l'ensemble des tâches à ordonnancer, il existe un ordonnancement valide où *P* a la plus faible priorité »;
- l'idée est que dans un tel cas, *P* souffre des pires interférences possibles, et ceci indépendamment de l'ordonnancement des priorités des autres tâches;
- on peut trouver un schème optimal (ou presque; s'il y a possibilité de blocage entre les tâches, on peut se rapprocher de l'optimalité si le coût du blocage est faible) de priorités en appliquant successivement cette approche et en enlevant chaque fois de l'ensemble la tâche ayant été placée.

Ordonnancement EDF

- un test d'ordonnançabilité pour EDF remonte aussi à Liu et Layland (1973). Ce test suppose que $D = T$ (donc que l'échéance est égale à la période) :

$$\sum_{i=1}^{N} \left(\frac{C_i}{T_i}\right) \leq 1$$

- la simplicité du test pour EDF est appréciable : ici, dans la mesure où le taux d'utilisation est inférieur à 100%, le critère d'ordonnançabilité est rencontré;
- en général, les algorithmes de la famille FPS sont plus souvent rencontrés dans les STR du fait qu'ils sont plus simples à implémenter que ne le sont les algorithmes de la famille DPS comme EDF;
- avec EDF, toute tâche doit avoir une échéance;
- en pratique, EDF est plus sensible aux surcharges systémiques que ne le sont les algorithmes FPS. Quand le système est débordé, EDF tend à entraîner (effet domino) une cascade de manquements aux contraintes;
- le test d'ordonnançabilité ci-dessus est exact pour EDF mais l'équivalent est strictement suffisant pour FPS. On sait par contre que tout ordonnancement valide avec un algorithme FPS serait aussi valide avec EDF;
- détail important : avec un algorithme FPS, l'instant critique représente le pire cas d'utilisation du système, mais **avec EDF ce n'est pas le cas**.

Avec EDF, on peut évaluer la charge du système à un instant t, sous la forme suivante :

$$h(t) = \sum_{i=1}^{N} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i$$

et on constate alors que, pour respecter les contraintes systémiques, il importe que :

$$\forall t > 0 : h(t) \leq t$$

En pratique, les seules valeurs de t à vérifier sont celles correspondant à des échéances, et il y a une limite supérieure L aux valeurs de t pour lesquelles il est pertinent de calculer $h(t)$. C'est un peu complexe, mais en gros, $\boldsymbol{L = min(L_a, L_b)}$ où

$$L_a = max \left\{ D_1, \ldots, D_N, \frac{\sum_{i=1}^{N} (T_i - D_i)\, {C_i}/{T_i}}{1 - U} \right\}$$

(ici, U est le taux d'utilisation de la tâche, équivalent à $\frac{C}{T}$) et où, en calculant les divers poids ω^i à droite, on finit par frapper le cas $\omega^{j+1} = \omega^j$, dans quel cas $L_b = \omega^j$.

$$\omega^0 = \sum_{i=1}^{N} C_i$$

$$\omega^{j+1} = \sum_{i=1}^{N} \left\lceil \frac{\omega^j}{T_i} \right\rceil C_i$$

Problèmes avec EDF

EDF a ses propres problèmes :

- avec FPS, on a des inversions de priorités, mais avec EDF, on a des **inversions d'échéances** lorsqu'une tâche a besoin d'une ressource qui se trouve prise par une autre tâche ayant une plus longue échéance;
- les solutions aux inversions d'échéances ressemblent aux solutions pour l'inversion de priorités, mais tendent à être un peu plus complexes.

Ordonnancement et multiprocesseurs

- deux grandes approches : **globale** et **partitionnée**;
- l'approche partitionnée repose sur une répartition *a priori* de tâches sur des unités d'exécution. Son grand défi est de déterminer un partitionnement ordonnançable pour un ensemble de tâches donné sur le nombre de processeurs à sa disposition;
- l'approche globale ordonnance les tâches lorsqu'elles deviennent prêtes à être exécutées. Son grand défi est d'identifier une stratégie d'ordonnancement optimale dans les circonstances de son utilisation;
- l'approche globale peut parfois ordonnancer des ensembles de tâches qui ne pourraient pas être partitionnées *a priori*;
- ce secteur a moins de maturité que les précédents, mais quelques résultats intéressants ont été produits (11.14.1, pp. 410+).

Interruptions et tâches sporadiques

- un gestionnaire d'interruptions perturbe l'équilibre d'un système. On évalue ainsi le coût associé au traitement des événements sporadiques dans un système (en considérant σs comme l'ensemble des tâches sporadiques et *IH* comme le coût de gestion de l'interruption) :

$$\sum_{k \in \sigma s} \left\lceil \frac{R_i}{T_k} \right\rceil IH$$

STR, protocoles et entrées/ sorties

Quelques trucs sur lesquels il va falloir élaborer quand j'aurai quelques minutes.

Le Jitter[6]

Déplacement entre la pulsation idéale et la pulsion constatée.

> « Jitter period is the interval between two times of maximum effect (or minimum effect) of a signal characteristic that varies regularly with time. Jitter frequency, the more commonly quoted figure, is its inverse. »

La compensation pour le Jitter

Typique des lecteurs MP3 ou de lecteurs multimédias sur réseaux ± fiables. L'idée est de consommer au préalable une partie du flux et de l'entreposer pour compenser les éventuels déplacements. Cela introduit bien sûr une latence (sorte d'effet Janet Jackson, j'imagine).

> « The maximum jitter that can be countered by a de-jitter buffer is equal to the buffering delay introduced before starting the play-out of the mediastream. In the context of packet-switched networks, the term *packet delay variation* is often preferred over *jitter*. » [wiki]

Le Adaptive Jitter Buffering

Utiliser des tampons de taille variable en fonction des variations sur les taux de transferts. Malheureusement, c'est une technique brevetée :

http://www.freepatentsonline.com/6452950.html

Le Maximum Throughput Scheduling[7]

Technique pour ordonnancer les paquets sur un réseau (souvent sans fil) de manière à optimiser l'utilisation de la bande passante.

> « This is achieved by giving scheduling priority to the least "expensive" data flows in terms of consumed network resources per transferred amount of information »

La question en est une d'évaluation des coûts; sur un réseau sans fil, le chemin le moins coûteux est souvent celui qui doit réaliser le moins d'atténuation (de compensation pour le bruit). Sachant cela, optimiser le débit sur un réseau peut ne pas maximiser les profits pour l'opérateur du réseau.

[6] Voir http://en.wikipedia.org/wiki/Jitter

[7] Voir http://en.wikipedia.org/wiki/Maximum_throughput_scheduling

Le Realtime Transport Protocol (RTP)[8]

Protocole de transport pour des trames multimédia. Réalise entre autres de la compensation pour le *Jitter* et de la compensation pour les paquets arrivant dans le désordre (ce qui va de soi pour la compensation du *Jitter*). Opère sur la couche applicative, pas sur la couche transport.

Peut fonctionner sur TCP, mais c'est superflu : RTP implémente à sa façon des mécanismes de gestion du transport des paquets que TCP implémente aussi, cette fois dans une perspective de robustesse. RTP, de son côté, peut compenser pour certaines pertes (par exemple de contenu audio) en appliquant des algorithmes de dissimulation des erreurs, évitant de ce fait des allers/ retours sur le réseau. RTP lui-même ne gère pas les pertes de paquets; pour le protocole, ce problème revient à l'application.

RTP est typiquement combiné à d'autres protocoles, dont RTCP (ci-dessous), SIP (pour le signalement – la communication de l'état de la connexion entre les homologues), et un protocole de description du média véhiculé par la session (par exemple SDP[9], décrit par RFC 4566[10]). Une variante chiffrée de RTP, SRTP[11], existe, et repose sur ZRTP[12] pour la négociation des clés de chiffrement de données.

Sous RTP, le protocole lui-même ne connaît pas les formats multimédias véhiculés. Ceux-ci sont décrits par des profils, joints au flux de communication. Sans surprises, parmi les profils déjà définis, on trouve des trucs comme MP3 (audio) ou MP4 et H.264 (vidéo).

Le Realtime Transport Control Protocol (RTCP)[13]

Protocole cousin de RTP qui a pour rôle de faire circuler des infos sur la qualité de service de RTP. Opère sur la couche applicative, pas sur la couche transport.

Quand RTP et RTCP travaillent ensemble, RTP propage sur les ports pairs, alors que RTCP pour un port RTP donné sur le prochain port impair.

Détails techniques sur RTP et RTCP

Une session de communication RTP repose sur une adresse IP et une paire de ports, pour RTP et pour RTCP. Les ports utilisés sont habituellement pigés dans la plage ouverte à tous (1024+).

> «An RTP sender captures the multimedia data, which is then encoded, framed and transmitted as RTP packets with appropriate timestamps and increasing sequence numbers. Depending on the RTP Profile in use, the Payload Type field is set. The RTP receiver, captures the RTP packets, detects missing packets and may perform

[8] Voir http://en.wikipedia.org/wiki/Real-time_Transport_Protocol

[9] Pour en savoir plus, voir http://en.wikipedia.org/wiki/Session_description_protocol ou encore http://www.konnetic.com/Documents/KonneticSDPTechnicalOverview.pdf

[10] Voir http://tools.ietf.org/html/rfc4566

[11] Voir http://en.wikipedia.org/wiki/Secure_Real-time_Transport_Protocol ou http://tools.ietf.org/html/rfc3711

[12] Voir http://en.wikipedia.org/wiki/ZRTP

[13] Voir http://en.wikipedia.org/wiki/RTP_Control_Protocol

> reordering of packets. The frames are decoded depending on the payload format and presented to the end user». [wiki]

Le format : http://www.networksorcery.com/enp/protocol/rtp.htm

Une référence plutôt touffue : http://www.cs.columbia.edu/~hgs/rtp/

RTP et RTCP sont définis par RFC 3550 : http://tools.ietf.org/html/rfc3550

Version antérieure dans RCC 1889 : http://tools.ietf.org/html/rfc1889

Implémentation C++ et complètement libre de droits de RTP, conforme à RFC 3550 : http://research.edm.uhasselt.be/~jori/page/index.php?n=CS.Jrtplib (la documentation en ligne peut être consultée sur http://research.edm.uhasselt.be/jori/jrtplib/documentation/index.html).

Variante intéressante : le Stream Control Transmission Protocol (SCTP)

Protocole de transport (qui peut, si aucun support natif n'est offert, être implémenté à travers un tunnel TCP) décrit par RFC 4960[14] qui décompose un message en *chunks* (en *motons*) et fait en sorte que le destinataire reçoive un message entier (plutôt qu'une trame de taille maximale fixée *a priori* – UDP – ou qu'un flux de *bytes* – TCP. Les types de *motons* sont définis dans le RFC[15].

SCTP supporte le *multi-streaming*, ou transmission en parallèle de plusieurs messages. Sous SCTP, les messages sont numérotés, et peuvent ne pas arriver en ordre à la destination. L'application qui recevra les messages décidera de les traiter en fonction de leur ordre d'émission ou de leur ordre de réception.

Tester le tout –générateurs de trafic

http://www.cs.columbia.edu/~hgs/internet/traffic-generator.html

[14] Voir http://tools.ietf.org/html/rfc4960

[15] Voir http://tools.ietf.org/html/rfc4960#section-3.2

Annexe 00 – Extraits de code choisis

Suivent quelques extraits de code utilisés en soutien au propos des section de ce volume sans être centraux pour ce propos.

Listes de types

Les listes de types ont été couvertes dans ***STR—Volume 03***, de même que dans Internet[16].

L'idée est d'**Andrei Alexandrescu**, dans son excellent livre *Modern C++ Design*, et est enrichie ici de quelques prédicats simples pour déduire, de manière statique, la tête et la queue d'une liste de types.

Nous utilisons ici une version enrichie de deux primitives statiques, soit `static_head` et `static_tail`, qui permettent respectivement de déduire le type en tête d'une liste statique et d'obtenir la queue de la liste.

```
class Vide {};
template <class T, class U>
   struct type_list
   {
      typedef T tete;
      typedef U queue;
   };
template <class TList>
   struct static_head;
template <class T, class Q>
   struct static_head<type_list<T, Q> >
   {
      typedef T type;
   };
template <class TList>
   struct static_tail;
template <class T, class Q>
   struct static_tail<type_list<T, Q> >
   {
      typedef Q type;
   };
```

Une classe `Incompilable`

Il est parfois utile de bloquer la compilation de certains types pour lesquels il serait préférable de spécifier immédiatement la raison d'une erreur de compilation.

Vous trouverez des explications plus détaillées dans Internet[17].

```
class Compilable
{
};
template <class Raison>
   class Incompilable
   {
      static static_assert<false> bloquer_la_compilation();
      enum { bidon = sizeof(bloquer_la_compilation()) };
   };
```

[16] Entre autres dans http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Metaprogrammation.html

[17] Voir http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Incompilable.html

Les assertions statiques

Certaines expressions sont sémantiquement illégales *a priori*, mais sont syntaxiquement correctes. Pour ces expressions une assertion statique est toute indiquée.

Vous trouverez plus d'explications dans Internet[18], en particulier dans la documentation de *Boost*. La technique est d'ailleurs si répandue qu'avec C++ 11, le mot `static_assert` est devenu un mot clé du langage.

```
template <bool>
   struct static_assert;
template <>
   struct static_assert<true>
   {
   };
template <class T>
   void unused(const T &)
      { }
```

Les alternatives statiques

Il est possible de choisir un type parmi deux à l'aide d'une alternative statique[19]. Il est d'ailleurs possible de réaliser des alternatives statiques imbriquées, du fait que le fruit d'une alternative statique est elle-aussi un type.

```
template <bool, class, class>
   struct static_if_else;
template <class SiVrai, class SiFaux>
   struct static_if_else <true, SiVrai, SiFaux>
   {
      typedef SiVrai type;
   };
template <class SiVrai, class SiFaux>
   struct static_if_else <false, SiVrai, SiFaux>
   {
      typedef SiFaux type;
   };
```

Les traits de types

Les traits de types utilisés ici rejoignent ceux décrits par **Andrei Alexandrescu** dans *Modern C++ Design*, mais plusieurs ont une contrepartie standard directe dans C++ 11.

Le synopsis des services clés apparaît à droite, mais référez-vous à ***STR—Volume 03*** pour plus de détails.

```
meme_type<T,U>::VAL;
est_const<T>::VAL
est_volatile<T>::VAL
est_reference<T>::VAL
supprimer_const<T>::type
supprimer_volatile<T>::type
supprimer_reference<T>::type
est_primitif<T>::VAL
est_entier<T>::VAL
est_virgule_flottante<T>::VAL
est_signe<T>::VAL
est_non_signe<T>::VAL
traits_type<T>
```

[18] Voir http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Metaprogrammation.html

[19] Voir http://h-deb.clg.qc.ca/Sujets/Divers--cplusplus/Metaprogrammation.html