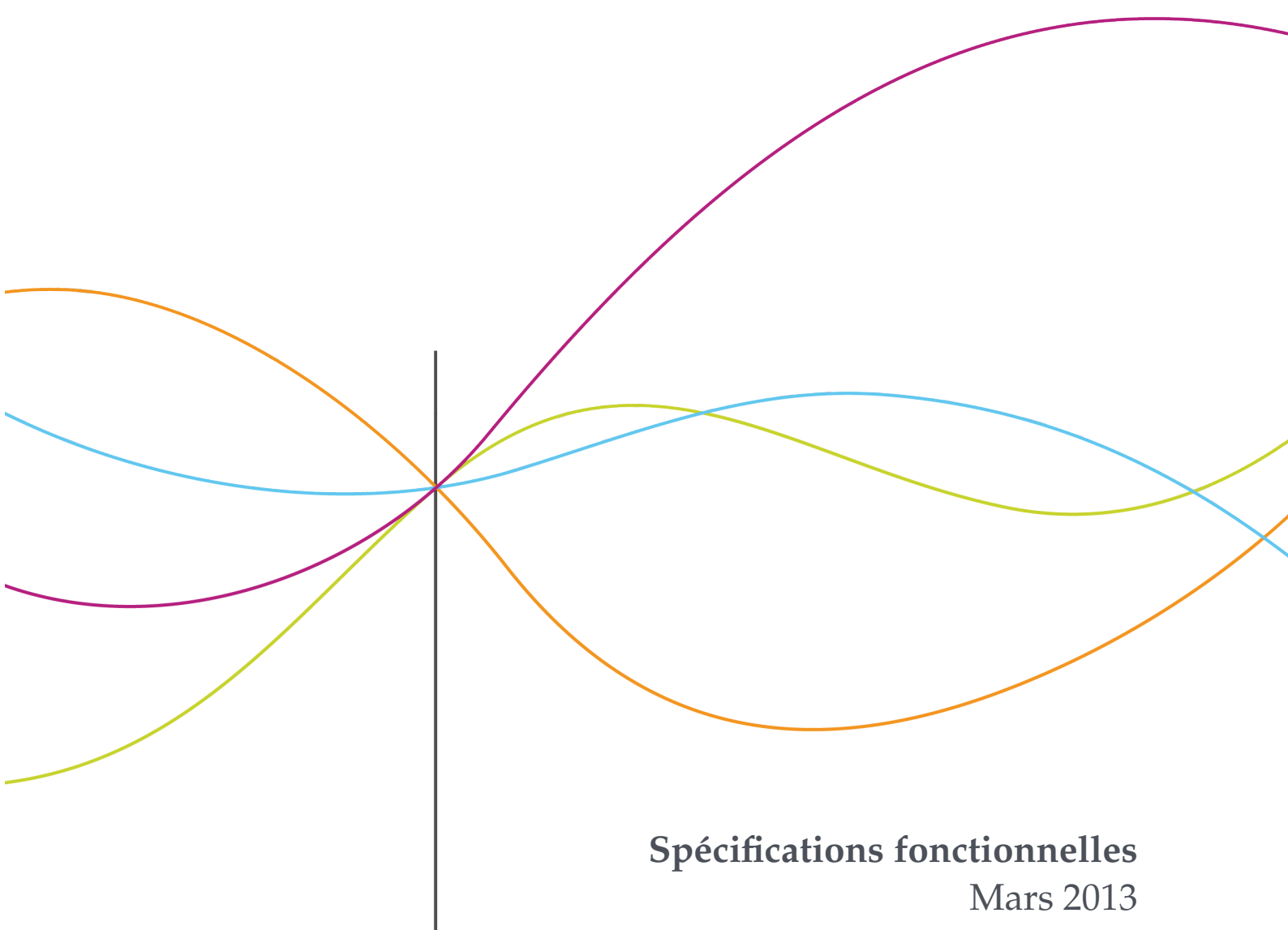


Département Génie Mathématique

Affectation spatiale sous contraintes



Spécifications fonctionnelles
Mars 2013

Adeline Bailly & Alexandre Quemy

Table des matières

Introduction	1
1 La documentation des interfaces de chaque module	2
2 Les tests unitaires	3
2.1 Assignment	3
2.1.1 AssignmentFactory	3
2.1.2 Kuhn	3
2.2 Constraint	3
2.2.1 CustomConstraint	3
2.2.2 ConstraintSystem	3
2.3 Core	4
2.3.1 controller	4
2.3.2 stepContinue	4
2.3.3 timeContinue	4
2.4 Environment	4
2.4.1 Environnement	4
2.4.2 Eval	4
2.4.3 Resource	5
2.4.4 Space	5
2.4.5 Task	5
2.4.6 TaskSpot	5
2.5 Graph	6
2.5.1 simpleIndex	6
2.6 IA	6
2.6.1 ia	6
2.6.2 manualModel	6
2.7 Utils	6
2.7.1 baseLogger	6
2.7.2 logger	6
2.7.3 timeManagement	6

Introduction

Chapitre 1

La documentation des interfaces de chaque module

Chapitre 2

Les tests unitaires

2.1 Assignment

2.1.1 AssignmentFactory

On testera la fonction `KuhnInstance(std::map<Resource<Dim, Type, Data>*, int> _resource, std::vector<TaskSpot<Dim, Type>*> _taskSpot)` sur un cas simple : une ressource et une tâche dont il est facile de calculer le coût.

Ainsi qu'un cas où une ressource (ou une tâche) n'existe pas. La fonction devrait alors retourner une erreur.

2.1.2 Kuhn

On va tester la fonction `operator()`, deux cas sont à tester.

Cas simple où l'on a une ressource pour une tâche. C'est à dire un cas où l'on peut connaître facilement l'affectation que l'on aura.

Cas où l'on a pas le même nombre de ressources que de tâches.

2.2 Constraint

2.2.1 CustomConstraint

On va tester la fonction `operator()`. Selon les valeurs des paramètres, on vérifiera si la bonne fonction de rappel est appelée.

2.2.2 ConstraintSystem

On va tester la fonction `operator()`.

2.3 Core

2.3.1 controller

On va tester la fonction `run()`, pour cela on va utiliser une fonction qui nous permettra de compter le nombre de passage ie le nombre d'étapes. Il suffira de vérifier que le compteur a atteint le nombre d'étapes prévues.

2.3.2 stepContinue

On doit tester la fonction `_check()`. Deux cas sont à tester, lorsque l'argument `steps > 1` (retourne vrai) et lorsqu'il est ≤ 1 (retourne faux).

2.3.3 timeContinue

On doit tester la fonction `_check()`.

Si le chronomètre n'a pas été lancé, la fonction doit le lancer (cf logger).

Ensuite, on testera un cas où la durée limite n'a pas été atteinte, et un autre où elle a été dépassée.

On testera également le cas d'erreur où la durée est négative.

2.4 Environnement

2.4.1 Environnement

On doit tester la fonction `update(double _time)`, cette fonction met à jour l'environnement en fonction d'un paramètre de temps.

- Cas d'erreur : Si on passe un paramètre négatif, une erreur doit apparaître. En effet, cela reviendrait à un retour dans le passé.
- Cas aux limites : Si le paramètre passé est nul, on doit obtenir le même environnement qu'avant la mise à jour.
- Cas nominal : Enfin, si on passe un paramètre positif, l'environnement peut être différent, en effet celui-ci change avec le temps.

2.4.2 Eval

On doit tester la fonction `EvalLoop(Eval<Data> _eval, std::vector<Data*>& _data)`, cette fonction évalue les données selon la fonction `_eval`.

Les différents cas à tester sont :

- Si le vecteur de données est nul, la fonction doit renvoyer une table nulle (ou une erreur ?)
- Si la fonction d'évaluation est « constante », elle doit évaluer toutes les données de la même manière.

2.4.3 Resource

Quatre fonctions sont à évaluer.

La première est la fonction `update(double _time)`. Pour cette fonction, trois cas sont à tester.

- $time < 0 \Rightarrow$ Retourne une erreur
- $time = 0 \Rightarrow$ Pas de modification par rapport à avant la mise à jour

- $_time > 0 \Rightarrow$ Deux cas peuvent survenir et sont donc à tester. Le premier correspond au cas où aucune modification ne survient, par exemple lorsqu'une ressource n'a pas été réaffecté. Le second cas correspond au cas où au moins une modification de la ressource, par exemple lorsque la ressource a été réaffecté.

La seconde est la fonction `move(Direction _dir, double _time)`. Si le temps est négatif, on doit avoir une erreur. Si le temps est nul, les coordonnées de la ressource doivent rester inchangées. Si le temps est positif, les coordonnées de la ressource peuvent changées. On testera les différentes directions possibles, qui dépendent de l'espace.

La troisième est la fonction `isBusy()`, celle-ci teste si la fonction peut être réaffectée ou non. On doit tester si une ressource libre devient occupée après avoir été affectée.

La quatrième est la fonction `colliding()`. Cette fonction teste si la ressource ne rentre pas en collision avec un autre objet dynamique. Deux cas sont à tester, le cas où il y a une collision (la fonction doit renvoyer `true`), et le cas où il n'y a pas de collision (`false`).

2.4.4 Space

Une fonction à tester : `inSpace(Coordinate<Dim, Type> _coord)`, qui vérifie si les coordonnées sont dans l'espace.

- Premier cas : Les coordonnées ne sont pas dans l'espace. La fonction doit retourner `false`.
- Deuxième cas : Les coordonnées sont dans l'espace. On va vérifier sur chaque extrémité de l'espace (aux frontières) que la fonction renvoie bien vraie.

2.4.5 Task

On doit tester la fonction `update(std::function<int(int&)> _f)`. Cette fonction est associée à une fonction d'évaluation (`_f`), pour le test, on va créer une fonction qui renvoie toujours la même chose. Il suffira donc de tester si la valeur de la tâche est égale à ce que renvoie la fonction.

2.4.6 TaskSpot

On doit tester la fonction `update()`. Cette fonction met à jour la tâche à l'aide de la fonction associée à l'emplacement de travail. De la même manière, que pour la fonction `update` de `Task`, on crée une fonction dont on connaît le résultat. On teste ensuite si la nouvelle valeur de la tâche est la valeur qu'on attendait.

2.5 Graph

2.5.1 simpleIndex

?? Don't know

2.6 IA

2.6.1 ia

On va tester la fonction `update(double _time)`, qui met à jour le model. Lorsque le paramètre de temps est négatif, cela doit retourner une erreur. Lorsqu'il est nul, cela ne doit rien changer ; enfin s'il est positif, il peut y avoir des changements.

2.6.2 manualModel

On va tester la fonction `update(double _time, SpatialData& _spatialData)`. Lorsque le paramètre de temps est négatif, cela doit retourner une erreur. Lorsqu'il est nul, cela ne doit rien changer ; enfin s'il est positif, il peut y avoir des changements.

2.7 Utils

2.7.1 baseLogger

2.7.2 logger

2.7.3 timeManagement