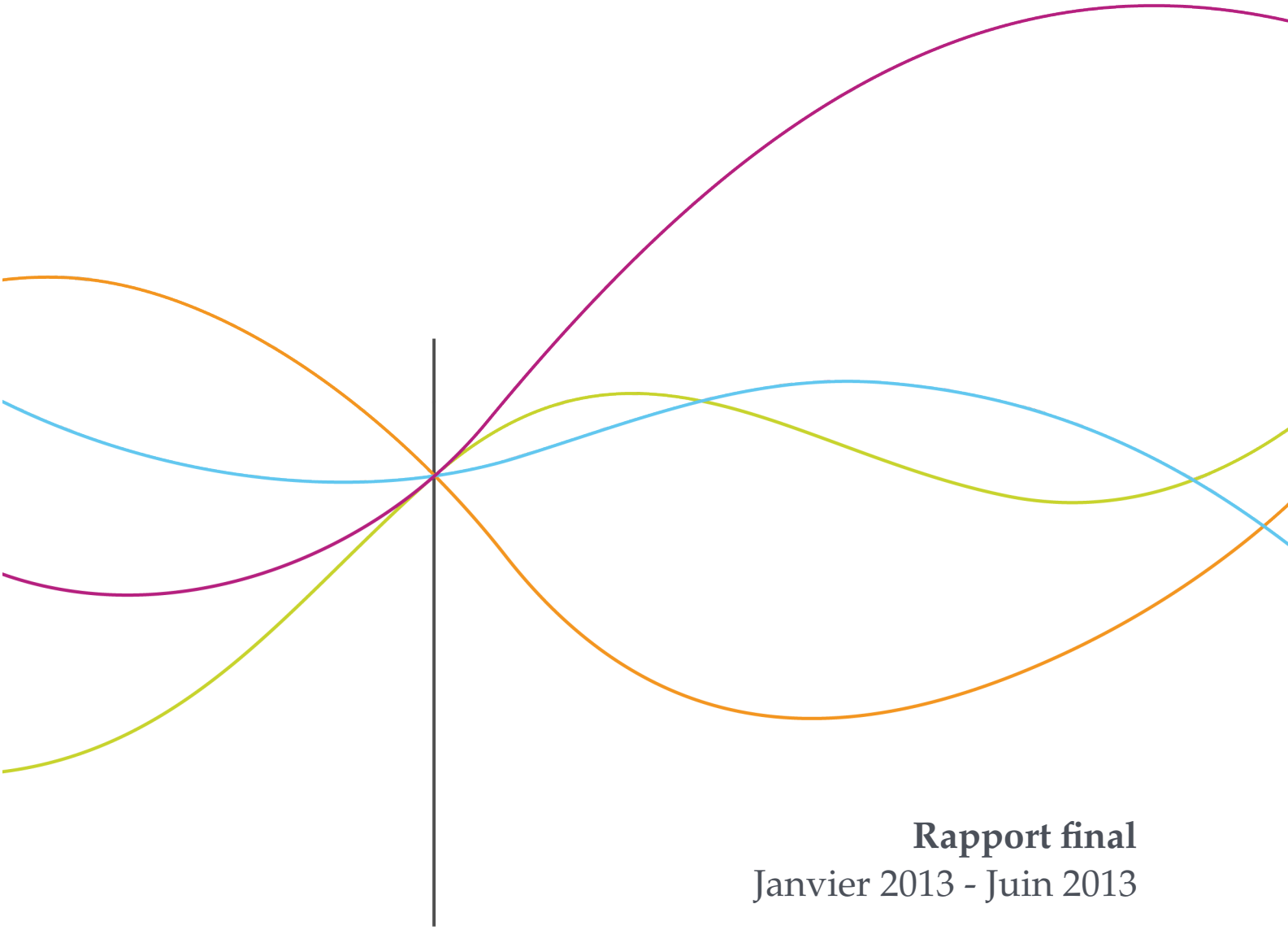
INSA ROUEN
À taille humaine,
à l'échelle du monde

Département Génie Mathématique

Affectation spatiale sous contraintes



Rapport final

Janvier 2013 - Juin 2013

Adeline Bailly, & Alexandre Quemy

Table des matières

Introduction

Le présent document est une synthèse de tous les documents de travail produit au cours du projet. Les précédents documents sont introduits tels quels, avec simplement des paragraphes supplémentaires, dont les titres sont en bleu dans la table des matières afin de faire gagner du temps.

Les ajouts de ce rapport sont, dans un premier temps un description de l'implémentation et du livrable, un guide utilisateur et un manuel d'installation, ainsi que dans second temps, une conclusion sur l'ensemble du projet.

Chapitre 1

Analyse des besoins

1.1 Présentation du sujet

L'affectation est un problème d'optimisation combinatoire qui consiste, dans sa version simple, à affecter un certain nombre de ressources disponibles à un certain nombre de tâches dans l'objectif d'optimiser une fonction objectif. Il peut s'agir de minimiser des coûts ou maximiser les bénéfices. Ce problème peut être résolu en temps polynomial par la méthode hongroise également appelée algorithme de Kuhn.
On peut étendre ce problème par des objectifs multiples, des contraintes changeantes en temps réel ou encore des contraintes probabilités traduisant une observation ou une connaissance partielle de l'environnement. On peut alors parler plus largement de planification, qui est un domaine ouvert de l'intelligence artificielle où de nombreuses formulations font apparaître des problèmes de la classe NP-difficile ou NP-complet sur lesquels de nombreuses équipes de recherches travaillent.

Dans le cadre de notre projet C++ nous avons choisi de réaliser un (début de) framework permettant la modélisation et l'implémentation de problème de planification spatiale sous contraintes. Il s'agira de proposer des briques logicielles permettant la création et la gestion d'IA pour la résolution de ce problème.
Enfin, pour illustrer la résolution temps réel des problèmes, en plus du framework, une application de simulation d'une Intelligence Artificielle sera développée. Il s'agira d'une application graphique montrant des unités d'un jeu vidéo affectées à divers postes pour maximiser divers objectifs pouvant changer au court du temps (équilibrage de ressources, objectif de construction, par exemple).

1.2 Framework : Description du système et objectifs

Le système doit permettre pour l'utilisateur la création d'une intelligence artificielle pour résoudre le problème d'affectation.
La création de l'IA se fait en plusieurs étapes :

- Définition de l'environnement (tâches, objectifs & contraintes, unités)
- Définition d'une méthode d'apprentissage et d'un modèle d'observation de l'environnement
- Création d'une ou plusieurs stratégies de résolution du problème d'affectation

Une stratégie classique consiste à :
- Discrétiser l'espace
- Indexer les unités, les tâches et en règle générale les objets de l'environnement
- Définir une fonction de coût
- Définir un algorithme d'affectation

Une stratégie peut être vu comme un algorithme dont on va pouvoir changer les briques évoquées ci-dessus.

Cela fait apparaître plusieurs axes de travail qui seront autant de modules du projet :
- L'IA et l'environnement
- Un module d'algorithmes d'indexation spatiale et de partitionnement spatial
- Un module d'algorithmes de plus court chemin (composant principal d'une fonction de coût)
- Un modèle d'algorithmes d'affectation

On pourrait imaginer un module dédié aux méthodes d'apprentissage et au modèle d'observation mais étant donné le temps à consacrer au projet, on choisira un modèle simple, déterministe et omniscient (c'est à dire que l'on aura pas d'apprentissage, que l'on considérera avoir toujours prit la bonne décision et l'ensemble de l'environnement est connu à chaque instant par l'IA).

1.2.1 Module d'IA et d'environnement

Il s'agit du module de plus « haut niveau » qui doit donner une API simple permettant de décrire un environnement spatial et ses objets (unités, tâches, obstacles), les contraintes liées aux tâches et enfin de quoi concevoir des stratégies données à l'IA (qu'on pourra intervertir pour par exemple constater de leurs différents effets).

Il devrait également reposer sur une technique d'apprentissage (par exemple : Algorithmes d'apprentissage par renforcement) et un modèle de prise de décision (par exemple : Processus de décision markovien partiellement observable) mais comme précisé plus haut, nous n'aurons jamais le temps d'envisager divers modèles de la sorte.

1.2.2 Module d'indexation spatiale et de partitionnement spatial

Ce module devra proposer deux choses :
- Des structures de données permettant l'indexation et le partitionnement spatial
- Des algorithmes permettant de réaliser l'indexation et le partitionnement

Il s'agira de proposer une API la plus unifiée possible pour permettre de changer de structures de données indépendamment de l'algorithme afin de pouvoir comparer leurs performances par exemple.

On peut citer quelques structures de données qui pourront être étudiées : les Arbre kd, R Tree, Hilbert R Tree, BSP, QuadTree ...

L'objectif de ce module est d'avoir des algorithmes qui vont permettre à l'IA de transformer un environnement « brut », à savoir un plan avec une liste d'objets dessus, en une ou plusieurs structures plus intelligente pour réduire la complexité de certaines heuristiques de plus court chemin ou d'affectation.

1.2.3 Module d'algorithmes de plus court chemin

Étant donné que nous travaillons avec une dimension spatiale, les fonctions de coûts d'affectation d'une unité à une tâche vont être majoritairement basées sur la distance à parcourir. De plus, il faudra ensuite que chaque unité puisse se déplacer vers la tâche qui lui a été affectée le plus rapidement.

L'objectif de ce module est donc de proposer divers algorithmes de plus court chemin, toujours avec une API la plus unifiée possible.

1.2.4 Module d'affectation

Il existe plusieurs approches pour l'affectation et ce module doit proposer plusieurs algorithmes pour résoudre le problème une fois l'évaluation des solutions possibles obtenues. On peut citer l'algorithme de Kuhn mais également la recherche des voisins les plus proches où l'algorithme des axes principaux.

1.2.5 Note sur l'implémentation et les objectifs

Le projet étant ambitieux à la vue de la multitude des algorithmes existants tant pour le plus court chemin que pour l'indexation spatiale voire pour la description des contraintes et objectifs, le travail réalisé sera axé sur la modélisation et l'architecture du framework. L'implémentation sera partielle mais les bonnes pratiques de développement occuperont une part importante de celle-ci dans le but de valoriser le livrable et permettre une implémentation complète en dehors du cadre de ce projet et d'ajouter de nouveaux algorithmes facilement.

On choisira de n'implémenter qu'un algorithme de chaque module (pour pouvoir proposer une IA fonctionnelle). Notre choix s'est porté sur des algorithmes classiques et relativement simple d'implémentation :

- Partitionnement : Simple grille et QuadTree
- Indexation : R-Tree
- Plus Court Chemin : A*
- Affectation : Méthode hongroise

1.3 Application : Description du système et objectifs

L'application servira de démonstration à l'IA construite avec le Framework. Il s'agira d'un petit jeu de stratégies en temps réel.
L'environnement sera une carte 2D avec des murs (les obstacles), des ouvriers (les ressources), et des mines d'or et des mines de pierres (les tâches, qui sont donc au nombre

de deux). L'action de miner durera un certain temps qui occupera un ouvrier et lui permettra de faire augmenter le compteur général d'or ou de pierres.

Les fonctionnalités suivantes permettront à l'utilisateur d'interagir avec le monde pour voir la réaction de l'IA :

- Supprimer / Créer un ouvrier
- Augmenter / Diminuer le nombre d'or / pierres en réserve
- Fixer un ratio entre or et pierres
- Fixer un objectif à atteindre pour l'or ou/et la pierre

1.4 Notes sur les besoins

Le découpage en modules a été respectée et les besoins n'ont pas changés. Seuls l'implémentation a été encore un peu réduite par rapport à ce qui a été prévu, le rapport ayant été écrit lorsque nous pensions être 4 pour réaliser le projet.
Pour le détail de l'implémentation, se référer à la section correspondante.

Concernant l'application de démonstration, pour la même raison, elle a été abandonnée mais le temps nous a finalement permis d'en créer une version très simplifiée.

Chapitre 2

Modélisation

2.1 Identification des briques logicielles

2.1.1 Module d'IA et d'environnement

Dans ce module, on distingue plusieurs entités :

- L'environnement, espace dans lequel évolue les objets. En pratique, généralement $\mathbb{R}^2$ ou $\mathbb{R}^3$.
- Les objets, étant de plusieurs types :
  1. Les ressources, que l'on peut affecter à une tâche et qui sont donc mobiles dans l'environnement.
  2. Les tâches, fixes dans l'environnement et qui permettent de modifier un ou plusieurs objectifs.
  3. Les obstacles, fixes dans l'environnement et qui modélisent la structure de l'espace.
- L'IA, qui va selon une ou plusieurs stratégies observer l'environnement et prendre des décisions pour résoudre le problème d'affectation.
- Les contraintes et objectifs, qui permettent de définir des seuils (minimal, maximal, proportion) par rapport à un type de tâche.
- Les stratégies, qui définissent un déroulement du processus d'affectation (fonction de coût, algorithmes utilisés)
- Les modèles d'apprentissage et de prise de décision (non étudié ici).

2.1.2 Module d'indexation spatiale et de partitionnement spatial

Les entités de ce module sont :

- Les arbres, servant à stocker l'information spatiale et y accéder rapidemment.
- Les techniques de construction de ces arbres. Il s'agit principalement d'algorithmes qui sont plus spécifiques à chaque structure de données et qui peuvent les améliorer. On peut citer par exemple le Z-order qui permet de construire de manière efficace des Quadtree ou des arbres de Hilbert.

2.1.3 Module d'algorithmes de plus court chemin

Les entités de ce module sont simplement les algorithmes travaillant sur l'environnement transformé grâce aux techniques d'indexation et de partitionnement spatial.

2.1.4 Module d'affectation

Les entités de ce module sont simplement les algorithmes qui doivent permettre la résolution de l'affectation.

2.2 Cas d'utilisation

2.2.1 Environnement et IA

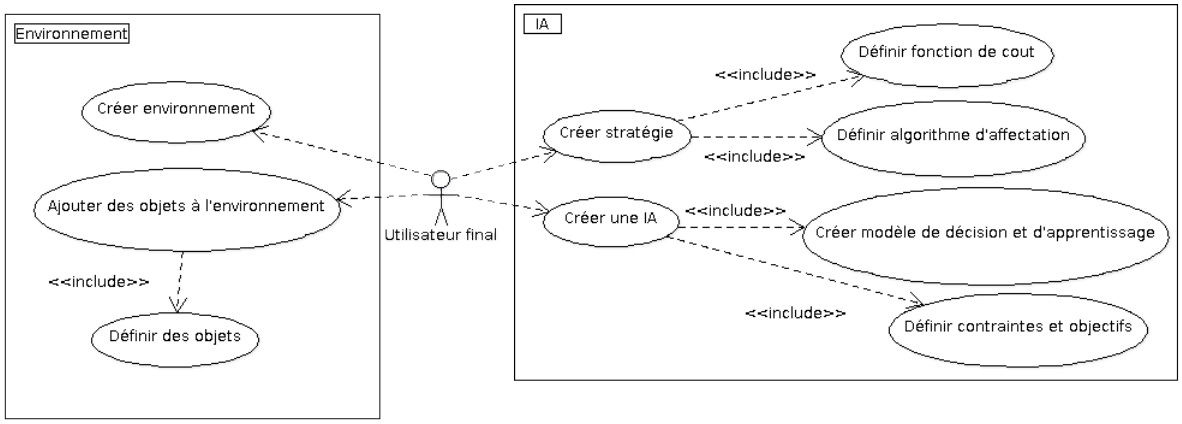


FIGURE 2.1 – Cas d'utilisation généraux

2.2.2 Ressources

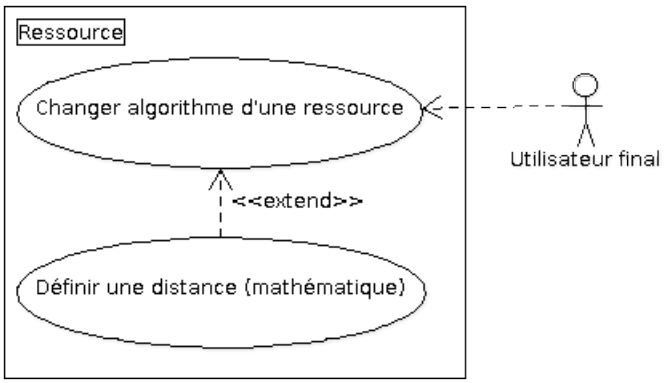


FIGURE 2.2 – Cas d'utilisation liés aux ressources

2.2.3 Framework

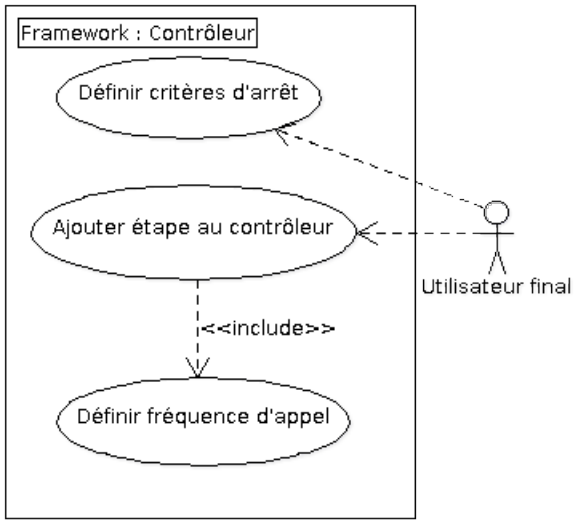


FIGURE 2.3 – Cas d'utilisation liés au framework

2.2.4 Partitionnement, Indexation

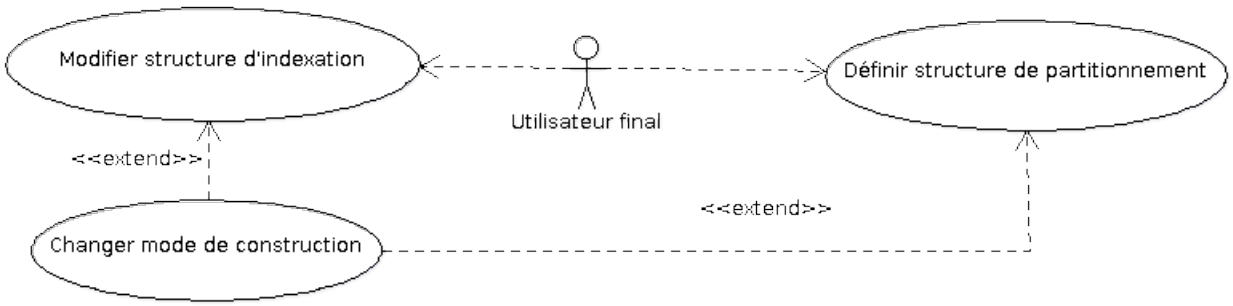


FIGURE 2.4 – Cas d'utilisation liés au partitionnement et à l'indexation

2.3 Modélisation des entités

2.3.1 Tâche

Une tâche est définie par une représentation numérique. Ne doit pas être confondue avec l'emplacement de travail (TaskSpot) qui est un objet concret de l'environnement qui peut modifier la tâche.

2.3.2 Contraintes

Une contrainte est une assertion et sera modélisée sous forme de foncteur renvoyant vrai ou faux. Il existe plusieurs types de contraintes ayant plusieurs fonctionnements. Chacune possède une priorité qui pourra représenter soit un ordre de traitement par l'IA soit une pénalité lors de l'évaluation.

Note : Il serait intéressant de pouvoirs détecter les contraintes impossibles à tenir en fonction de l'environnement ou tout simplement parce que deux contraintes s'opposent. Le problème de satisfaction de contraintes est à lui tout seul un problème NP-difficile dans la plupart des cas.

On supposera durant ce projet que ce point ne pose pas de soucis et que l'utilisateur n'entrera pas des contraintes qui pourront amener le système à se bloquer ou à ne pouvoir satisfaire l'ensemble des contraintes dans un temps raisonnable.

Par la suite il pourra néanmoins être intéressant de proposer des algorithmes de déduction de contraintes et de résolution du problème CSP (Constraint satisfaction problem).

Contraintes de seuils

Une contrainte de seuil (appliquée à une tâche) est composée d'un opérateur $\leq$ ou $\geq$, et d'une valeur numérique.

Ces contraintes possèdent deux fonctions de callback qui doivent se déclencher lorsque l'on franchi le seuil défini, d'un côté et de l'autre (ie, cela implique de garder en mémoire la valeur lors de la vérification précédente).

Optionnellement, on peut définir un intervalle de tolérance non nécessairement symétrique.

Contraintes de de proportionnalité

Défini une relation de proportionnalité entre 2 ou plusieurs tâches. On peut imaginer par exemple vouloir 30 % de pierres contre 70 % d'or dans le cadre d'un jeu de statégie.

Là encore, on peut définir de manière optionnelle un intervalle de tolérance et deux fonctions de rappel qui seront appelées en cas de changement de statut de la contrainte (passage du respect au non respect et inversement).

Contraintes personnalisées

Une contrainte personnalisée est une contrainte modélisée par une fonction renvoyant un booléen : vrai si la condition est vérifiée, faux sinon. Elles servent à exprimer des contraintes diverses sur les tâches mais aussi sur d'autres éléments de l'environnement en fonction du

problème de l'utilisateur.

Ces contraintes portent également deux fonctions de rappel pour le changement de statut de la contrainte.

2.3.3 Système de contraintes

Il s'agit d'un système comprenant les tâches ainsi que les contraintes. Les tâches / contraintes sont potentiellement indépendantes de l'IA qui va résoudre le problème puisqu'elles appartiennent au problème lui même.

Ainsi le but du système de contraintes est de centraliser les tâches et les contraintes pour en faciliter l'accès par l'utilisateur et l'IA.
Le système pourra également proposer des méthodes de résolution du CSP évoquées plus haut.

Note : La description des contraintes peut se faire de manière plus générale et plus souple par l'utilisateur grâce à un design pattern interpréteur (que nous n'implémenterons pas dans l'immédiat).

2.3.4 L'environnement

Il est composé d'un espace au sens mathématique, ainsi que de 3 types d'objets : les obstacles, les points de travail et les ressources. Il a donc principalement un rôle de conteneur.

Étant donné que la création de l'environnement peut très vite devenir complexe, un design pattern builder est envisagé, permettant par exemple de charger un fichier d'environnement depuis différents types de fichiers. Ce sera d'ailleurs le cas pour l'application d'exemple.

L'environnement est à la fois un observateur et un observable. Il est observé par l'IA et observe les objets dynamiques qu'il contient. Ainsi il s'agit d'un médiateur entre les objets qu'il contient et l'IA.

2.3.5 L'espace

L'espace est simplement défini par un nombre de dimensions et le type de ses coordonnées. Il contient également les limites de l'espace. L'espace peut donc être imaginé comme un parallélotope droit.

2.3.6 Les coordonnées

Les coordonnées permettent de localiser l'ensemble des objets dans l'espace. Elles présentent deux caractéristiques : le nombre de composantes et le type de ces composantes. Il s'agit d'un vecteur stockant n composantes d'un type numérique.

2.3.7 Les objets

Les objets évoluent dans l'environnement et plus particulièrement dans l'espace qu'il contient. Ainsi, tous les objets possèdent des coordonnées qui doivent être du même type que ceux de l'espace considéré.

Il existe deux types d'objets (pour le moment) : les objets statiques et les objets dynamiques. Chacun de ces deux types donnent lieu à une hiérarchie d'objets.

Tous les objets implémentent une géométrie qui permettra d'effectuer des calculs de collision selon diverses techniques. Par exemple on peut définir un objet par un polygone convexe, un polygone quelconque, un carré, un cercle. Chaque représentation à ses avantages et inconvénients sur les méthodes de collisions : algorithme du point dans un polygone (convexe), Bounding-Box, rayon de collision, etc.

Tous les objets possèdent également une fonction update qui sert à mettre à jour leur état interne régulièrement.

Les objets statiques

Il s'agit des objets de l'environnement qui ne peuvent pas bouger. Leur intérêt principal est de définir des obstacles dans l'espace, bloquant le passage aux ressources.

↪ **Les obstacles**

Il s'agit du seul objet statique concret dont nous avons perçu l'intérêt pour l'instant. Il est là pour modéliser, comme son nom l'indique, un obstacle dans l'espace.

Les objets dynamiques

Ce sont des objets qui sont capables d'évoluer.

↪ **Les ressources**

Elles représentent des objets dont l'IA dispose pour pouvoir les affecter à diverses tâches en vu de satisfaire les contraintes du système de contraintes. Il s'agit encore d'une classe abstraite dont le but est de clarifier l'architecture des objets. On retrouvera différents types de ressources ayant des comportements différents.

Ainsi on peut imaginer une ressource qui n'est pas réaffectable lorsqu'elle commence une tâche qui lui est donnée par l'IA, on peut imaginer une ressource qui n'accepte qu'un seul type de tâche, etc. Toutes ses variantes dépendent du problème et seront rajoutées à mesure que le besoin s'en fera sentir.

Dans notre cas on n'envisagera qu'une unité simple qui pourra être réaffectée n'importe quand, et une unité qui ne pourra pas être réaffectée.

Chaque ressource est un agent au sens où il est partiellement autonome. Il évolue indépendamment de l'IA qui ne fait que lui communiquer un ordre (une affectation). C'est la ressource qui se déplace de son propre chef pour se déplacer vers l'objectif. Ainsi chaque

ressource possède un algorithme de plus court chemin ainsi qu'une vitesse de déplacement.

Comme l'intérêt du partionnement spatial et de l'indexation astucieuse des objets réside dans la diminution du temps de calcul des algorithmes en général, cela implique que les ressources doivent accéder aux structures de partitionnement et d'indexation. Ainsi ce n'est pas l'IA qui s'occupe de stocker ces structures (on peut imaginer dans un jeu vidéo que l'IA soit remplacé par un joueur).

↪ **Les emplacements de travail**

Ce sont les représentations physiques d'une tâche. C'est à dire des objets qui vont pouvoir modifier une tâche (son « compteur ») en fonction de certaines conditions.
Ainsi, ce sont des observateurs d'une tâche particulière qu'elle notifie en cas de changement.

Là encore, on peut imaginer beaucoup de variantes : des emplacements temporaires – l'emplacement est détruit au bout de tant de temps d'utilisation, des emplacements avec un compteur – l'emplacement se détruit après tant d'utilisation, etc.

2.3.8 Les politiques de chevauchement

Les objets sont paramétrés par une politique de chevauchement : certains objets peuvent ne pas en chevaucher d'autres.
Il s'agit concrètement d'un entier qui permet de définir une priorité sur le chevauchement. Un objet ne peux pas surpasser un objet ayant une valeur plus haute que la sienne.
Des constantes sont prévues pour définir des objets « fantômes » et des objets que l'on ne peut pas chevaucher, comme les obstacles par exemple.

2.3.9 Les fonctions de coûts

Il existe plusieurs types de fonctions de coûts dont le rôle est similaire : évaluer un objet ou une quantité en fonction de différents paramètres. Classiquement, on peut évaluer la satisfaction d'une contrainte, les paires (ressource / emplacement de travail), la situation globale.
La manière dont l'évaluation est faite dépend de la stratégie adoptée.

L'implémentation est très libre et souple au vu de ce que le C++ permet de faire, surtout avec le nouveau standard.

2.3.10 Les stratégies

Il s'agit d'une manière de résoudre le problème d'affectation pour satisfaire les contraintes. Elle comporte une manière d'évaluer la situation et un algorithme d'affectation.

Une stratégie très simple consiste à ne traiter la satisfaction que d'une contrainte à la fois. Il s'agira de la seule stratégie que nous implémenterons.

L'évaluation de la situation est faite en fonction de la stratégie. Voici la stratégie que nous implémenterons (visible également sur le diagramme de séquence du contrôleur de l'IA) :

- Evaluation des contraintes
- Evaluation des paires ressource / emplacement
- Evaluation globale des ressources
- Evaluation de la situation

Voici un exemple concret des fonctions de coûts associés : La contrainte x non satisfaite : cette contrainte portait sur la tâche y dont la valeur est trop faible.

- Evaluation de l'intérêt des taskSpots
  # Le coût des taskSpots rattachés à y est 1, sinon 0.
- Evaluation d'un individu
  # Distance des couples (Indi , TaskSpot)
- Evaluation globale d'individu
  # evalIndi * evalTaskSpot
- Evaluation situation
  # Somme des évaluations globales des individus

Dans cet exemple très simple on ne veut affecter des ressources qu'aux emplacements susceptibles de faire augmenter la quantité non satisfaite. Ainsi dans l'algorithme d'affectation, les couples (individu, emplacement) où l'emplacement n'est pas rattaché à cette quantité seront nuls et donc ne pourront être choisi.

L'évaluation de la situation est un indicateur pour l'IA afin qu'elle puisse décider si elle doit changer de stratégie ou non.

2.3.11 L'IA

Il s'agit d'un conteneur de stratégies qui sont organisées selon un modèle (couple d'un modèle de décision et d'observation), pour pouvoir plus facilement traiter le problème d'affectation.

2.3.12 Arbres

Utilisés pour indexer les objets, pour partitionner l'espace. On retrouve beaucoup d'arbres différents ayant chacun leurs avantages et désavantages. En voici une liste assez complète qui pourront être implémentés par la suite :

- AtlasGrid
- R-tree
- Quadtree
- Octree
- BSP
- KD-tree
- implicit KD-tree
- vantage KD-tree
- R+-Tree
- R-Tree
- X-Tree
- M-Tree
- Hilbert R-Tree

- UB-Tree

2.3.13 Données spatiales

Etant donné que plusieurs objets qui ne se connaissent pas doivent accéder aux structures de données spatiales (que ce soit la partition de l'espace ou l'index des objets dynamiques), il nous faut une classe dédié à cela.
Cet objet possède 3 structures de données qui indexent la structure de l'espace (donc les objets statiques), les tâches et les ressources à partir des conteneurs basiques (des tableaux) de l'environnement.

Lors de l'initialisation, le contrôleur créera cet objet automatiquement et passera une référence aux ressources (en passant par l'environnement) et à l'IA.

2.3.14 Les algorithmes

Chaque type d'algorithme définit une hiérarchie de classes héritant toutes d'une classe abstraite définissant l'interface de ce type d'algorithme.
On utilisera un design pattern Factory pour permettre une instanciation aisée pour l'utilisateur.

Algorithmes de plus court chemin

Les algorithmes de plus court chemin renvoient une liste de noeuds à parcourir pour une ressource pour arriver à son objectif. Il se peut qu'il y ait des déplacements à prévoir au sein d'un noeud mais c'est à la ressource de s'en occuper.

Algorithmes d'affectation

Les algorithmes d'affectation prennent en entrée les emplacements de travail
Etant donné que les algorithmes ne travaillent pas nécessairement sur le même type de données (par exemple la méthode hongroise travaille sur une matrice alors que la recherche du plus proche voisin peut faire appel à des structures d'arbre, etc.), une étape de transformation des données peut avoir lieu.

2.4 Architecture du framework

Cette section s'attele à décrire l'architecture globale du framework :

2.4.1 Contrôleur

Le contrôleur principal gère le déroulement globale de la simulation. Il s'agit d'une boucle principale avec des critères d'arrêt définis par l'utilisateur (nous proposerons un critère temporel).
La boucle principale va effectuer en permanence les même actions, dans le même ordre. Ces actions sont des appels à des contrôleurs locaux ou des actions définis par l'utilisateur parce qu'il en a besoin dans son application : interception et traitement des entrées claviers, affichage ...

Par défaut, seules deux actions sont présentes : l'appel au contrôleur local de l'IA et au contrôleur local de l'environnement.

Ces actions seront modélisées sous forme d'une liste de fonctions à appeler dans l'ordre. Chaque fonction sera assortie d'un délais d'activation : toutes les 100ms par exemple.

2.4.2 Logger

Un logger est absolument nécessaire pour un framework afin de repérer le déroulement du flux d'exécution et de localiser d'éventuel(s) problème(s) ou simplement visualiser le processus de calcul. C'est d'ailleurs très utile pour faire des tests unitaires ou d'intégration.

Nous mettrons en place un logger sous forme de Design Pattern Chaine de Responsabilité. Chaque maillon de la chaîne sera un logger avec un niveau de log différent (DEBUG, INFO, ERROR ...) et le message transite de maillon en maillon jusqu'à ce que le bon niveau soit atteint.
On pourra utiliser une sérialisation dans un fichier texte ou non.

Il existera une instance statique du logger pour permettre une utilisation globale. On n'utilisera pas de DP Singleton d'une part parce qu'il est très dur à rendre thread-safe et d'autre part parce pour une raison ou une autre, l'utilisateur final trouvera peut être une utilité à avoir plusieurs loggers.

2.4.3 Le modèle multi-thread

Nous n'implémenterons pas le modèle multithread pour des raisons de temps mais nous le rendrons facilement intégrable, notamment avec la fonction du contrôleur principal. En effet, les actions du contrôleur sont, normalement, toutes indépendantes et peuvent donc être lancées en parallèle.

Pour rendre l'application thread-safe à un niveau de granularité plus fin, on utilisera le paramétrage par politique qui permettra de découpler totalement le modèle multi-thread du reste et d'activer / désactiver le support multi-thread à la volée.

2.4.4 Les observateurs / observables

Le design pattern Observateur occupe une place importante notamment lors de la récupération d'information sur l'environnement par l'IA mais également par la notification de mise à jour de plus petites entités au sein de l'environnement (les objets). C'est pourquoi nous avons imaginés deux types de politiques liées à l'observation : une politique active et une politique passive.

Politique active

La politique active est le comportement classique lorsque l'on parle du patron Observateur. Dès qu'un changement apparaît sur un observé, il notifie l'observateur qui s'est abonné aux observés.
On retrouve ce comportement au niveau du couple Emplacement / Tâche où l'emplacement, sous certaines conditions, va notifier la tâche qui devra se mettre à jour.

Politique passive

La politique passive consiste pour l'observé à simplement changer son état indiquant qu'il a changé et préparer le message qui doit être transmis à l'observateur. C'est donc l'observateur qui est actif et va explicitement demander la notification d'état et récuperer les messages des observables qui ont indiqués qu'ils étaient modifiés.

Ce comportement apparaît par exemple au niveau de l'observation de l'environnement par l'IA. L'IA va périodiquement choisir de regarder l'état de l'environnement.

2.5 Diagramme de classes

2.5.1 Algorithmes

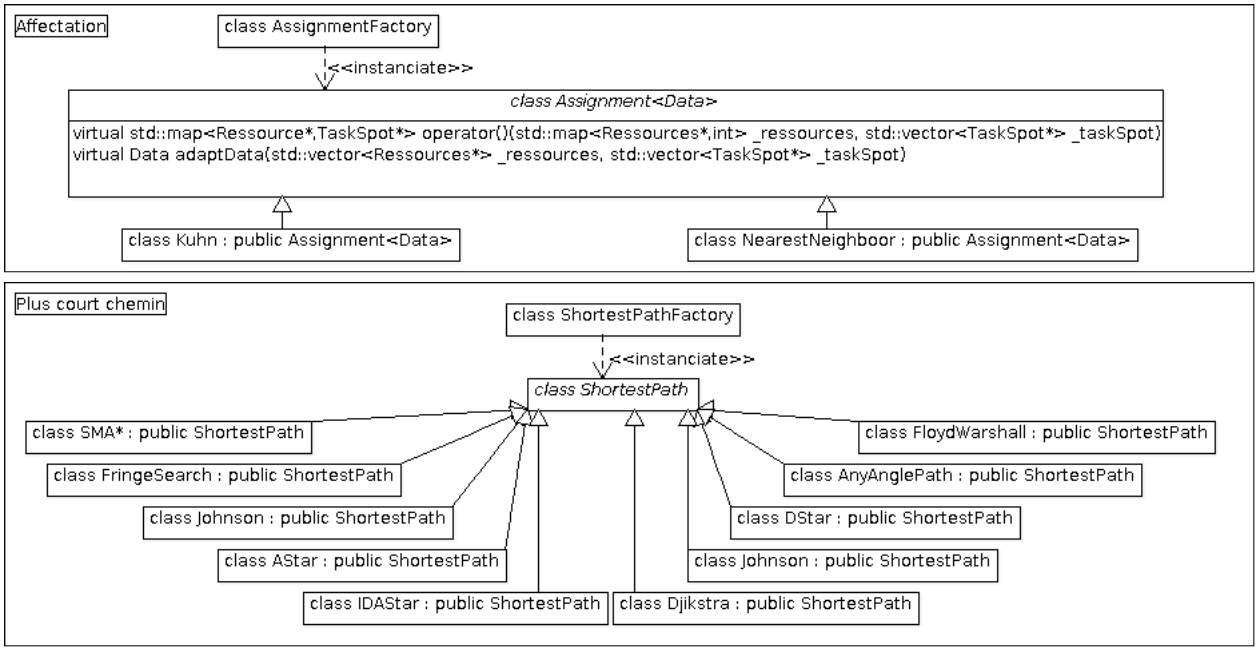


FIGURE 2.5 – Diagramme de classes des algorithmes

Pour faciliter l'instanciation des algorithmes de plus court chemin et d'affectation, un Design Pattern Factory est prévu. Il instanciera un objet dérivant de la classe abstraite Assigment ou ShortestPath qui représentent l'interface de manipulation des algorithmes.

2.5.2 Arbres

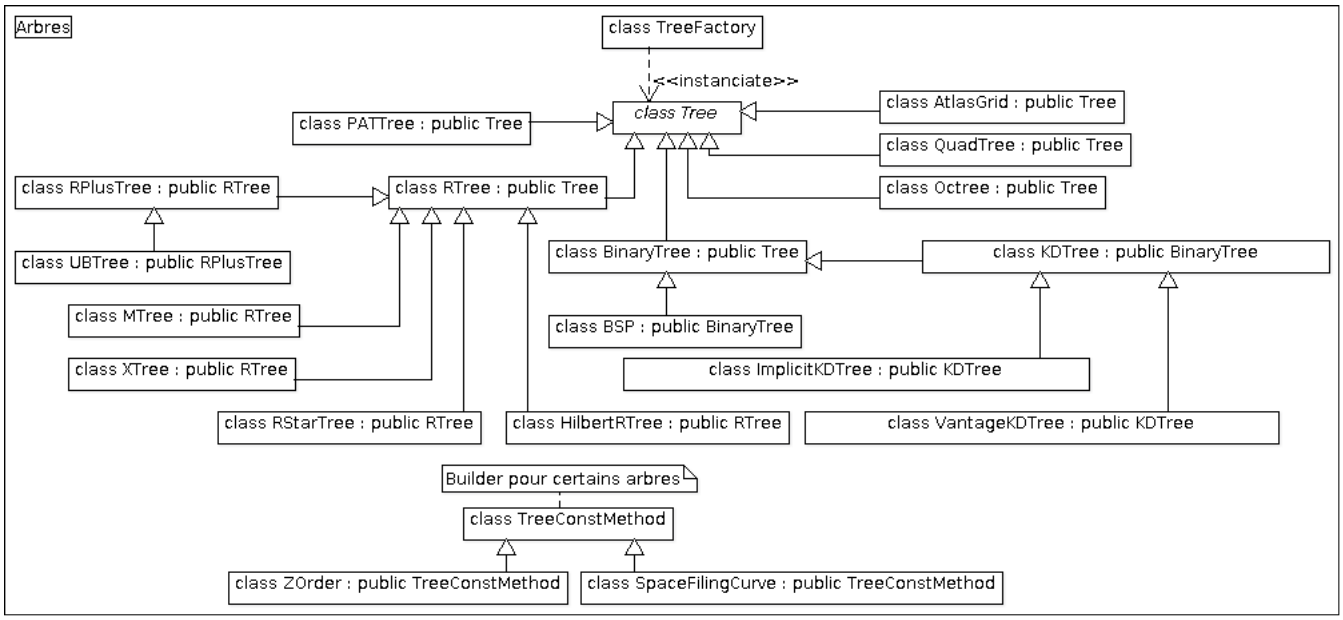


FIGURE 2.6 – Diagramme de classes des arbres

De la même manière qu'avec les algorithmes, une Factory permettra d'instancier un arbre concret.
Certains arbres peuvent se constuire de plusieurs manières, c'est pourquoi on prévoira un Design Pattern Builder pour permettre de monter différemment ces arbres.

2.5.3 Framework

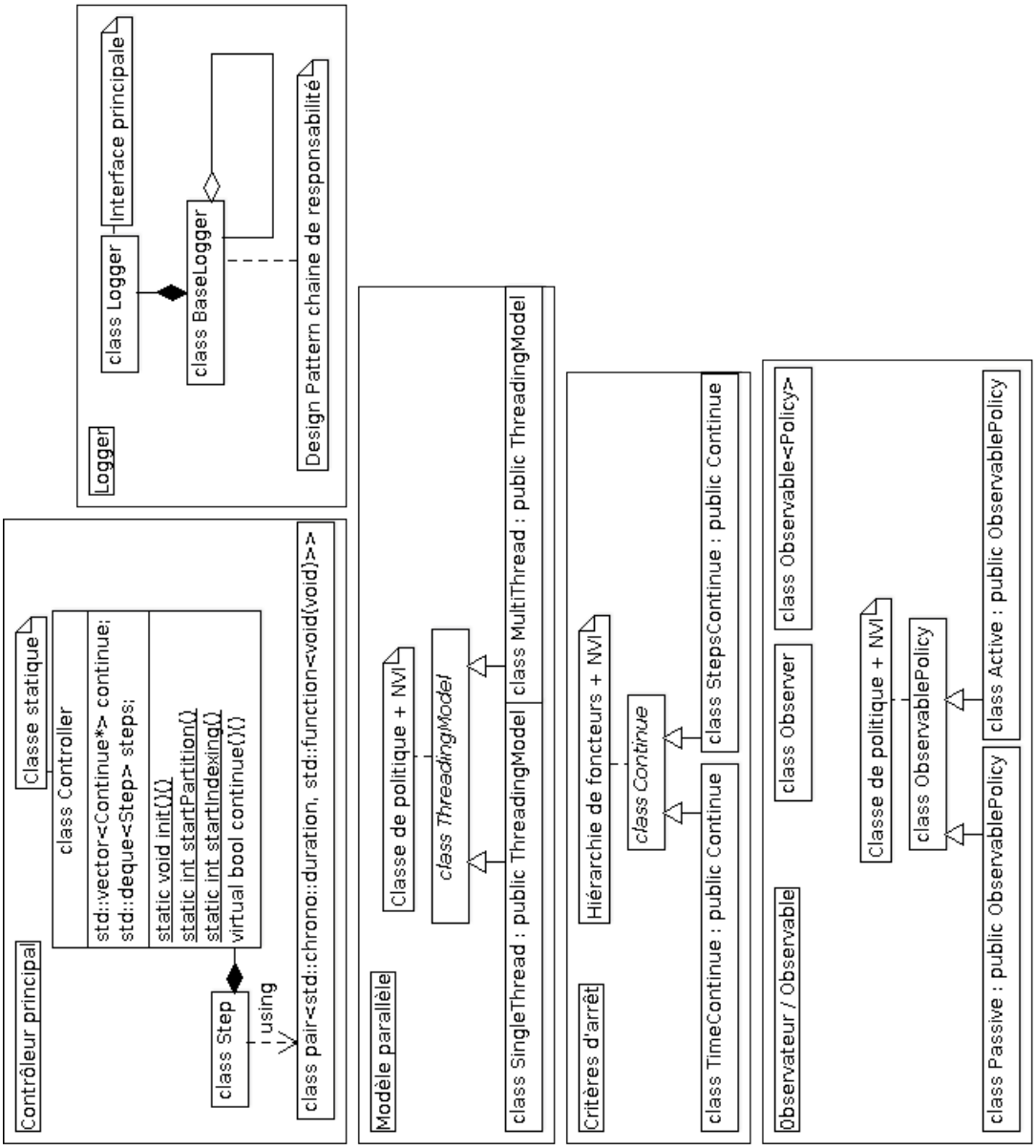


FIGURE 2.7 – Diagramme de classes du framework

2.5.4 IA & Contraintes

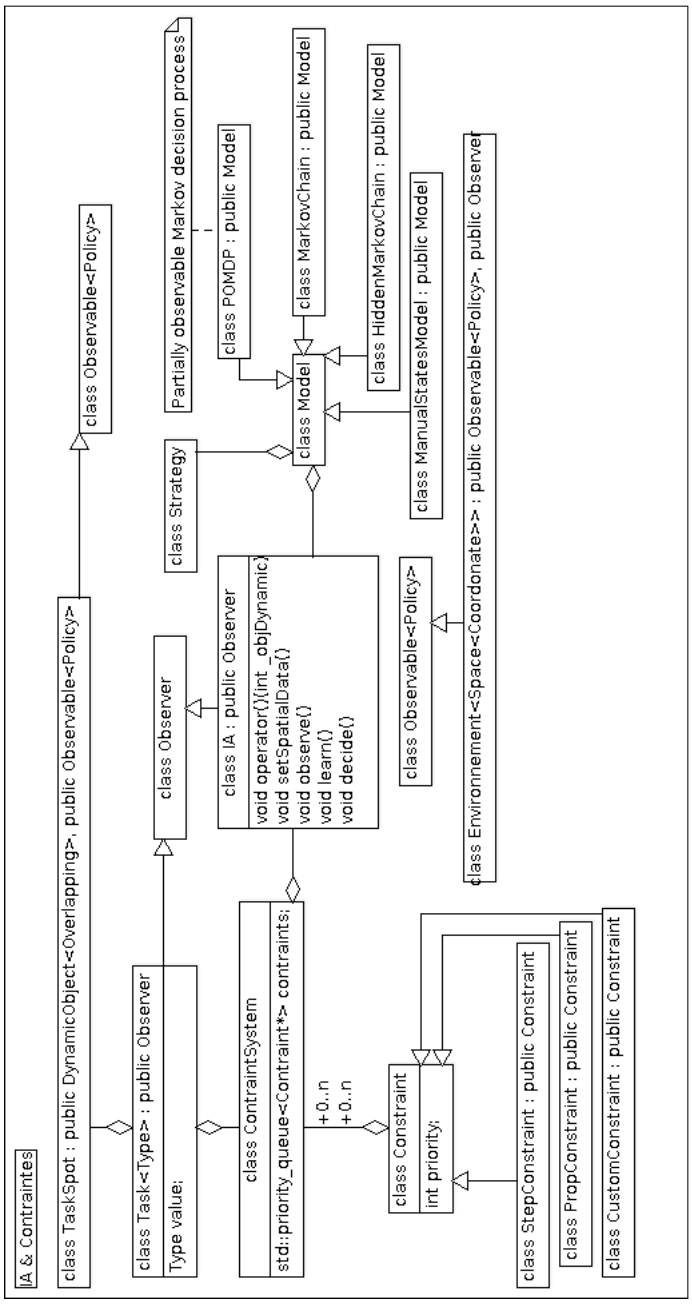


FIGURE 2.8 – Diagramme de classes d'IA et contraintes

2.5.5 Environnement

Une erreur de copié / collé avait fait disparaitre le diagramme de classe du rapport sur la modélisation...

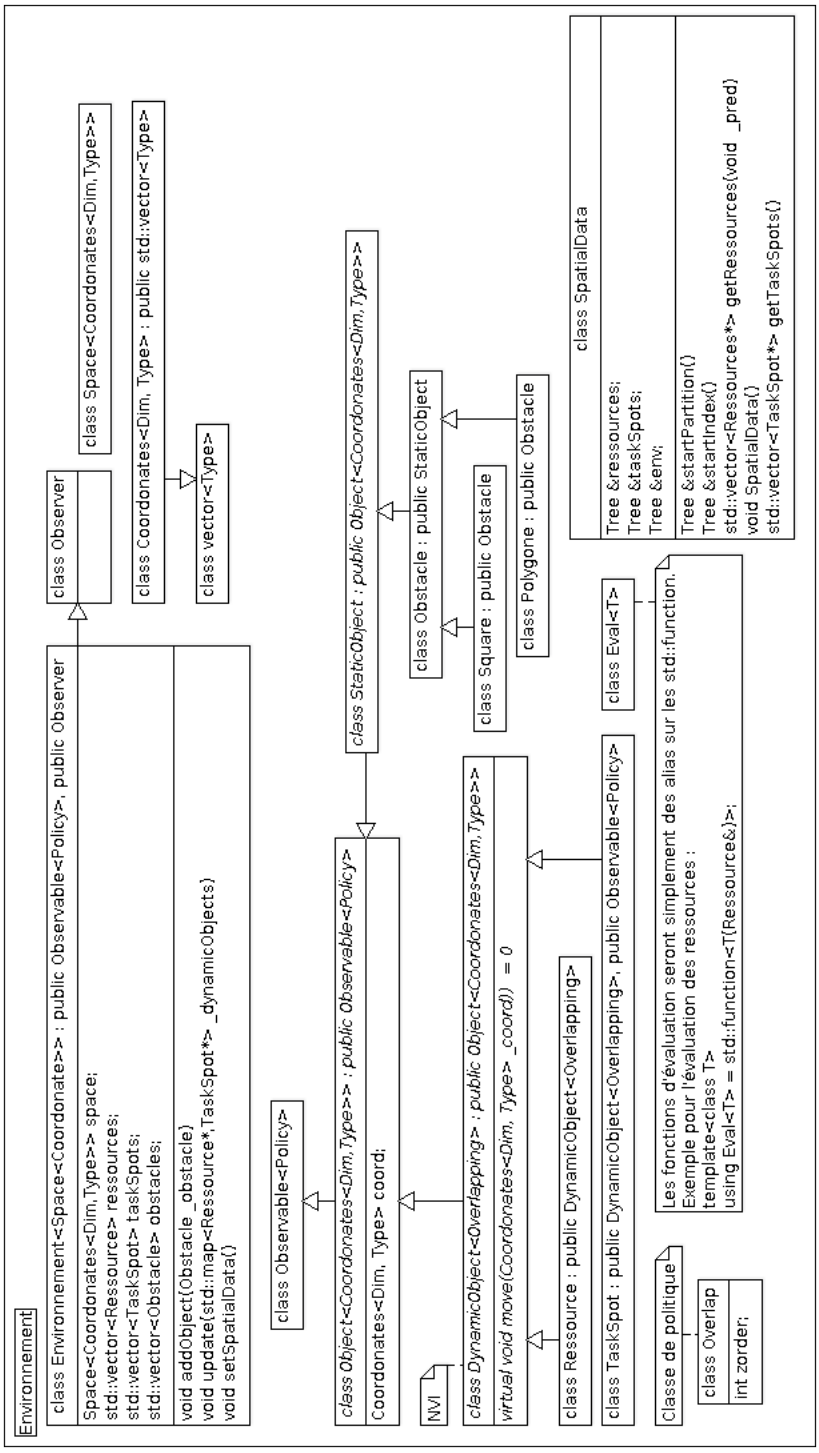


FIGURE 2.9 – Diagramme de classes de l'environnement

2.6 Scénario et diagrammes de séquences

On peut distinguer plusieurs scénarios pour notre application. Le scénario utilisateur où l'acteur principal est l'utilisateur final, c'est à dire celui qui va utiliser le framework pour résoudre un problème en fonction de son contexte.

Des scénarios annexes existent, relatifs aux sous-systèmes du framework et à l'interaction entre ses composants. Le principal est celui qui intervient durant toute la simulation / résolution et dont l'acteur principal est le contrôleur du framework.

2.6.1 Scénario principal

- L'utilisateur définit ses tâches
  ↪ Définition des contraintes
- L'utilisateur définit l'environnement
  ↪ Définition de l'espace (dimensions, frontières)
  ↪ Définition des objets statiques (obstacles)
  ↪ Définition des objets dynamiques
    - Définition des ressources
      - Définition d'une fonction de coût
    - Définition des emplacements de travail
      - Observation de l'emplacement par un type de tâche
  ↪ Paramétrage de l'environnement
    - Délais de mise à jour
- Définition de l'algorithme de partitionnement de l'espace
- Définition de l'algorithme d'indexation des objets dynamiques
- Création des stratégies
  ↪ Définition des fonctions d'évaluation
  ↪ Définition de l'algorithme d'affectation
- Création de l'IA
- Définition du modèle de décision et d'observation
- Ajout des stratégies
- Paramétrage de l'IA
  ↪ Délais entre chaque cycle « observation – apprentissage – prise de décision - affectation »
- Lancement de la simulation / résolution par le contrôleur du framework.

2.6.2 Scénarios annexes

Initialisation

L'initialisation est la première étape effectuée lors de l'exécution. Elle se déroule comme suit :

- Création de l'objet contenant les données spatiales
- Partionnement de l'espace
- Indexation :
  ↪ Des ressources
  ↪ Des emplacements de travail
  ↪ Des obstacles
- Donne un accès en lecture aux données spatiales

(Voir Diagramme de séquence d'initialisation 2.10, page 25).

Contrôleur principal

- Le contrôleur principal appelle le contrôleur de l'IA si la contrainte de temps est respectée
- Le contrôleur de l'IA :
  ↪ Observation du monde par la stratégie courante
    - Récupération des changements
    - Evaluation de la situation
  ↪ Apprentissage (non étudié ici)
  ↪ Prise de décision (non étudié ici - peut être effectué manuellement par l'utilisateur)
  ↪ Affectation par la stratégie courante
  ↪ Notification aux ressources des nouvelles affectations
- Le contrôleur principal appelle le contrôleur de l'environnement si la contrainte de temps est respectée
- Le contrôleur de l'environnement :
  ↪ Mise à jour des ressources
    - Si pas d'affectation : ne rien faire
    - Si affectation : calcul du chemin le plus court
    - Si chemin connu : déplacement et calcul de collisions
    - ...
- La boucle recommence tant que les conditions sont respectées

(Voir Diagramme de séquence de simulation 2.11, page 26).

Contrôleur IA

Contrôleur Environnement

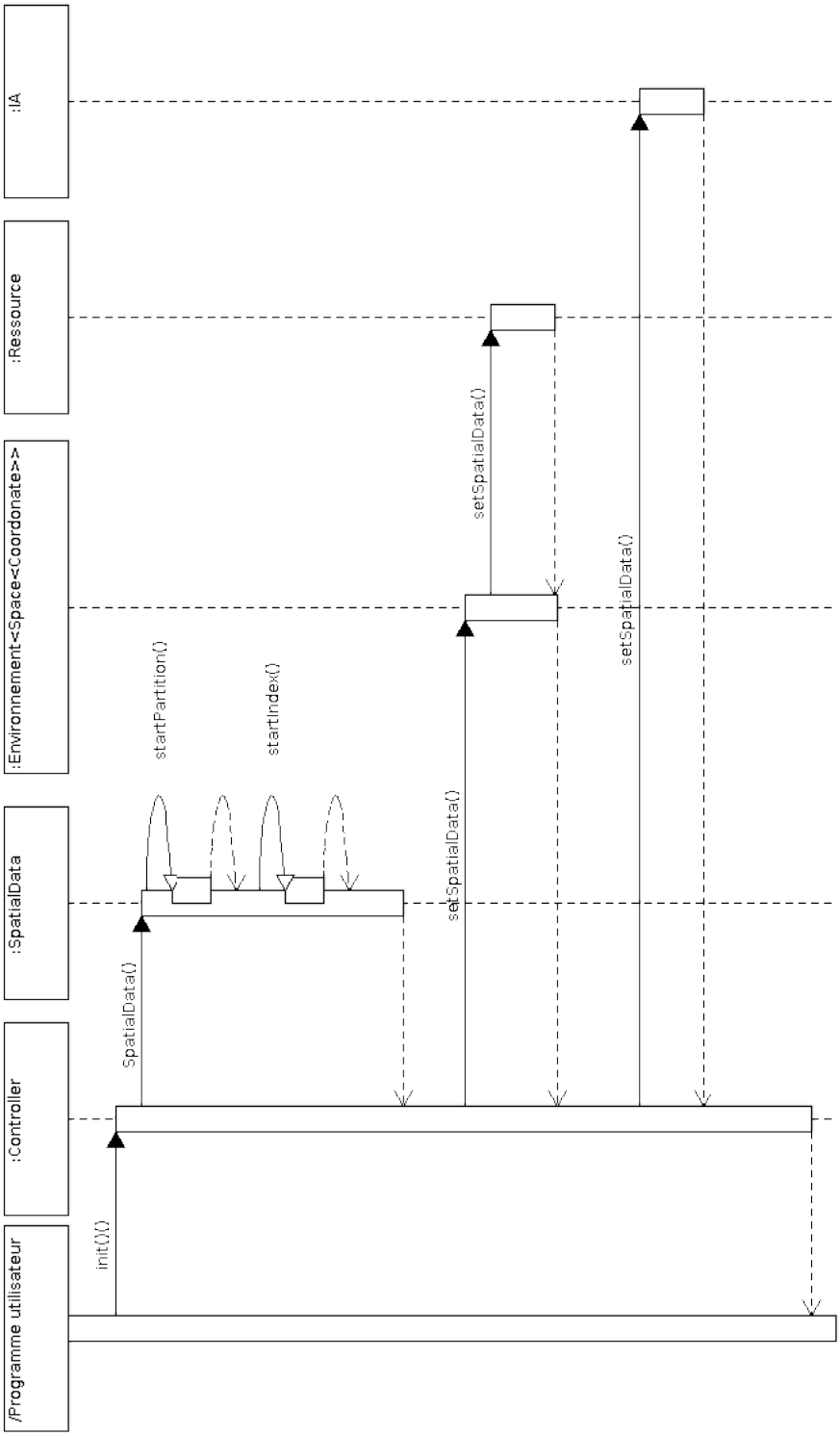


FIGURE 2.10 – Séquence d'initialisation

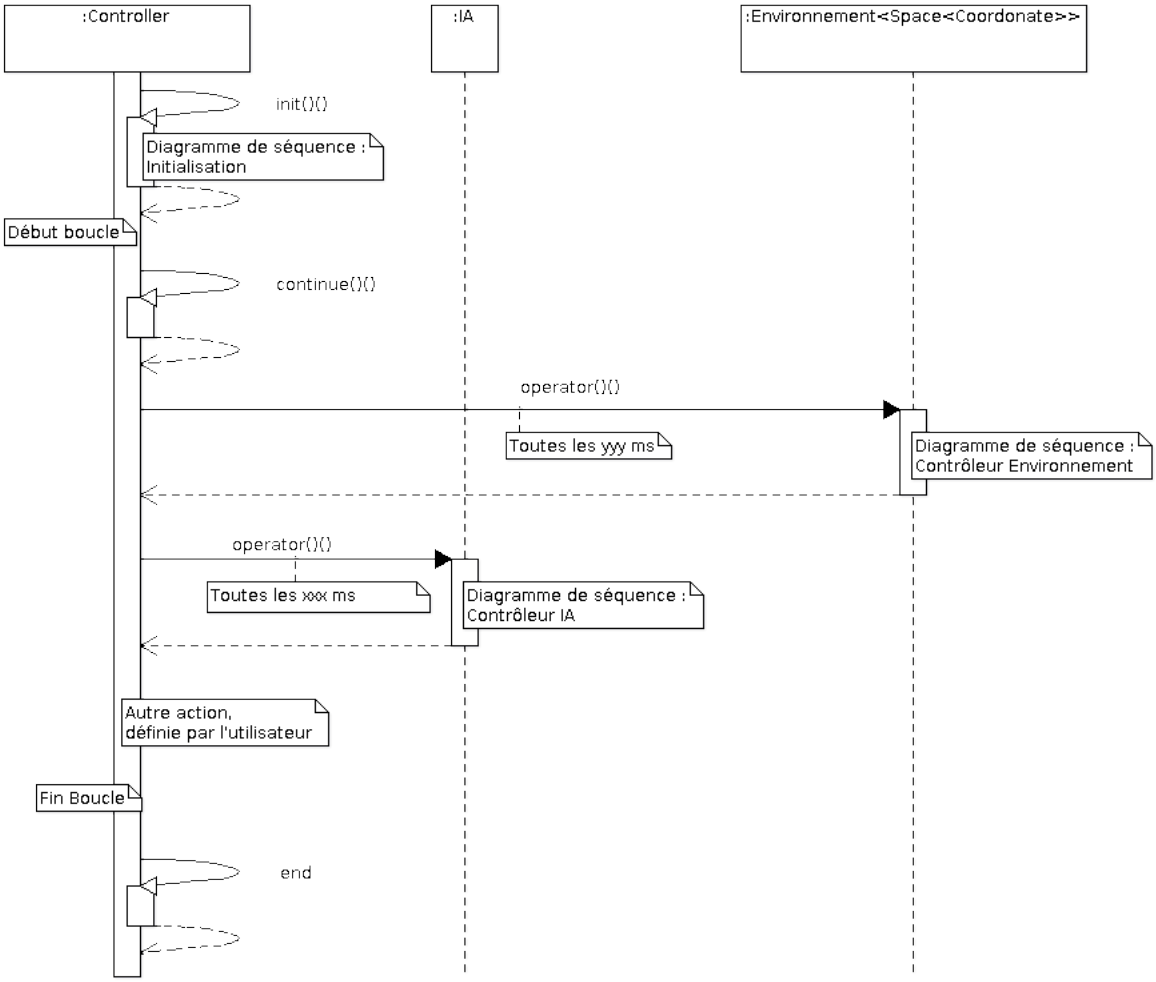


FIGURE 2.11 – Séquence de simulation

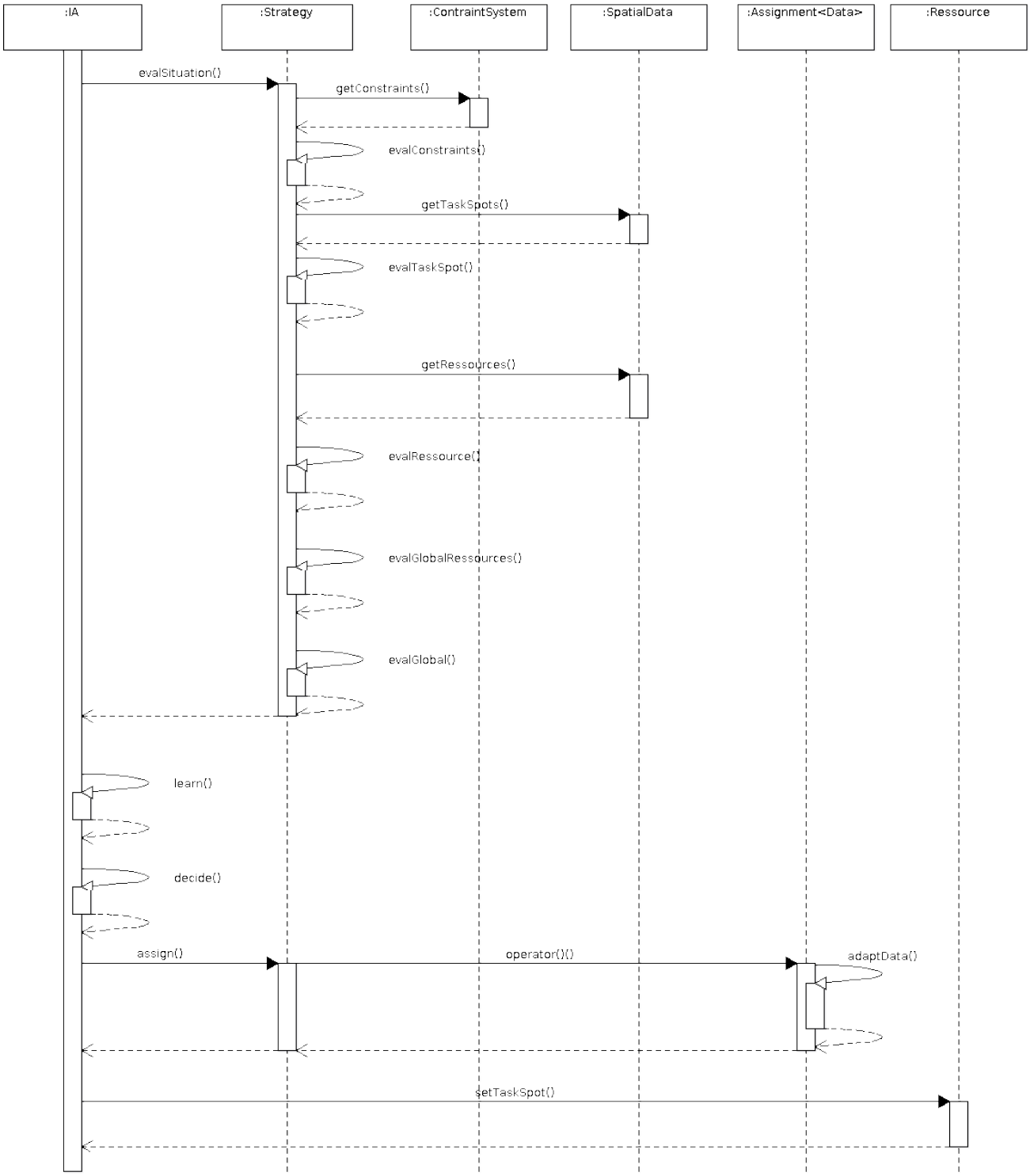


FIGURE 2.12 – Séquence du contrôleur de l'IA : cycle « observation – apprentissage – prise de décision - affectation »

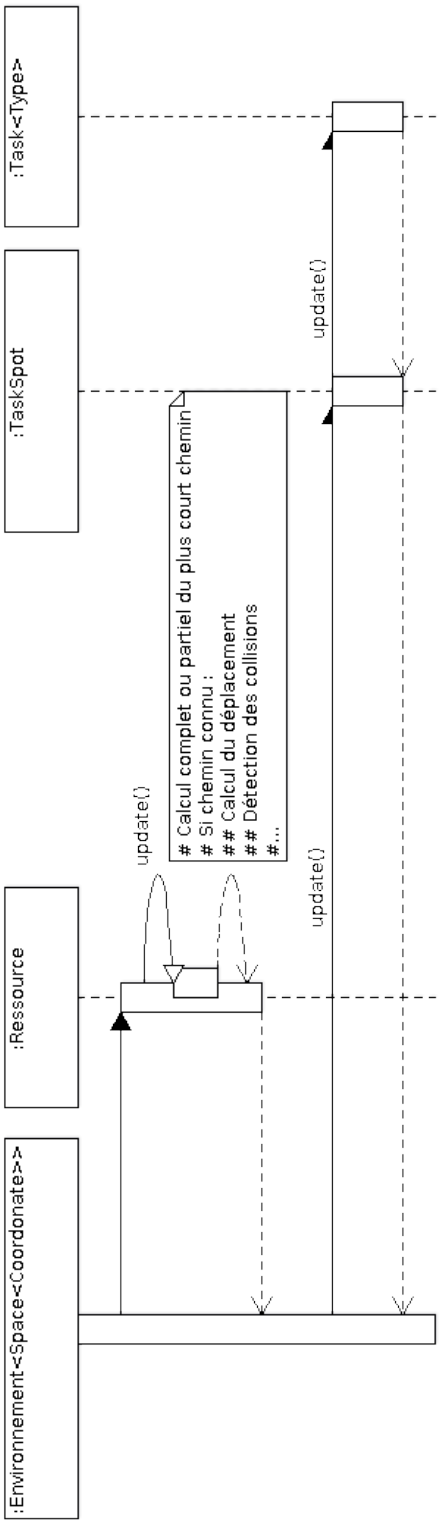


FIGURE 2.13 – Séquence de mise à jour de l'environnement

2.7 Annexes

2.8 Présentation des pattern récurrents

2.8.1 Idiome NVI

Présentation

L'idiome NVI, pour Non-Virtual Interface, est la réalisation en C++ du design pattern Template Method. Il possède plusieurs avantages notamment dans le développement d'un framework.
Il consiste à maintenir la consistance de certains comportements en un point de contrôle du code défini par le développeur. Cela permet d'une inversion de contrôle bénéfique dans de nombreux cas, dont le cas d'un framework puisqu'il déresponsabilise l'utilisateur final de certaines contraintes qui ne pourraient être exprimées que par de la documentation et dont le non respect entraineraient des comportements inattendus difficiles à localiser.
NVI est guidé par 4 règles décrites par Herb Sutter dans son article sur la virtualité :

- Prefer to make interfaces nonvirtual, using Template Method design pattern.
- Prefer to make virtual functions private.
- Only if derived classes need to invoke the base implementation of a virtual function, make the virtual function protected.
- A base class destructor should be either public and virtual, or protected and nonvirtual.

Exemple

Imaginons une bibliothèque de manipulation de matrices. La bibliothèque propose une classe abstraite Matrice avec l'API globale de toute matrice. On considérera ici la calcul du déterminant. Découle de cette classe une hiérarchie de matrices pour les cas particuliers : matrice triangulaire, matrice diagonale, etc.

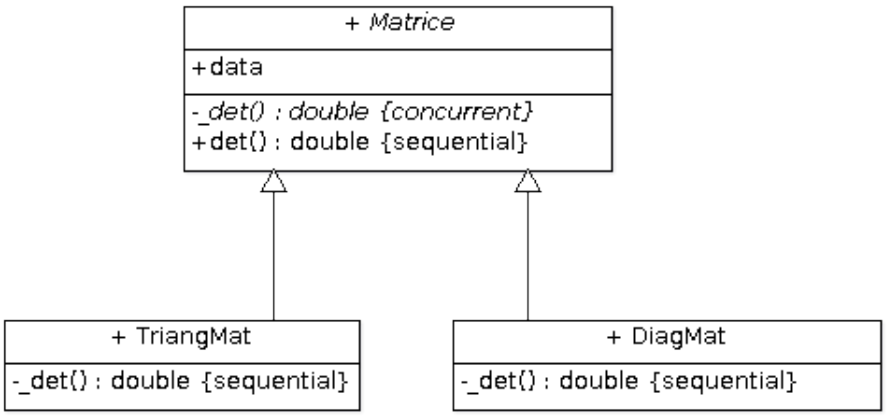


FIGURE 2.14 – Illustration du NVI.

```
class Matrice
{
public :
    double det() const
    {
        // Pre-traitement : verification d'invariants, lock multithread,
           etc.
        m.lock();
        assert(data_.check_invariants() == true);
        // Appel de l'implementation
        return _det();
        // Post-traitement
        assert(data_.check_invariants() == true);
        m.unlock();
    }

private :
    virtual double _det() const = 0;

protected :
    Data data;
    mutex m;
};

class TriangMat : public Matrice
{
private :
    virtual double _det() const
    {
        // Retourne le produit de la diagonale
    }
}

class DiagMat : public Matrice
{
private :
    virtual double _det() const
    {
        // Retourne le produit de la diagonale
    }
}
```

Listing 2.1– NVI : principe.

Ainsi, lorsque l'utilisateur voudra ajouter une nouvelle matrice, disons de forme Hessenberg, il aura simplement à implémenter le calcul du déterminant, les invariants étant toujours vérifiés dans le code de la classe de base qui va appeler l'implémentation.
Il y a donc une réelle inversion de contrôle et c'est le développeur du framework qui dirige le client à l'utiliser correctement.

Nous utiliserons par la suite du projet l'idiome NVI de manière intensive pour permettre de guider le flux d'exécution.

2.8.2 Paramétrage par politique

Présentation

Le paramétrage par politique est une technique de programmation développée et démocratisée par Andrei Alexandrescu dans son livre Modern C++ Design : Generic Programming and Design Patterns Applied et dans la bibliothèque Loki dédiée à la métaprogrammation en C++ dont beaucoup d'éléments ont été repris dans Boost puis dans le standard 2011.

Concrètement, il s'agit de profiter de l'héritage multiple et de la métaprogrammation pour permettre de séparer les différents comportements d'une classe ou de plusieurs classes et de créer sur mesure des comportements en combinant plusieurs politiques.

Fonctionnement

La clef d'un paramétrage par politique efficace réside dans l'analyse des différents comportements d'une classe ou d'un ensemble de classes. Imaginons que nous ayons à créer une bibliothèque de gestion de graphes. Un graphe peut être représenté sous différentes formes : matrice d'adjacence, matrice d'incidence , ou liste de successeurs. Chaque représentation à ses avantages et ses inconvénients en fonction des applications. On pourrait aisément créer 3 classes différentes mais cela ne serait pas très pertinent. On pourrait simplement templater la classe de graphe mais chaque type donnée n'a pas la même API. De plus, imaginons qu'un graphe puisse être partagé entre plusieurs thread ou non selon les applications.
Sans paramétrage par politique, il faudrait un nombre de classes égales au nombre de facteurs multiplié par le nombre de modes par facteur. Cela apporterait évidemment du code redondant et une maintenabilité moindre puisque dans le cas d'un paramétrage par politique on peut isoler complètement un comportement. Pour modifier l'intégralité du modèle multithread d'une application, la modification d'une seule classe de politique est nécessaire.

Chaque facteur est représenté par une classe abstraite permettant de définir l'API commune à tous les modes et qui pourra être utilisée par les classes paramétrées avec ce facteur. Les classes concrètes implémentent chacun des modes de la politique.
Enfin, la classe de services qui doit être paramétrée par politique va être templaté avec chacune des politique et en hériter de manière publique ou privée selon les besoins.

Exemple

On définit la première hiérarchie de classes correspondant à la première politique :

```
class ThreadingModel
{
protected :
    virtual void Lock() = 0;
};
```

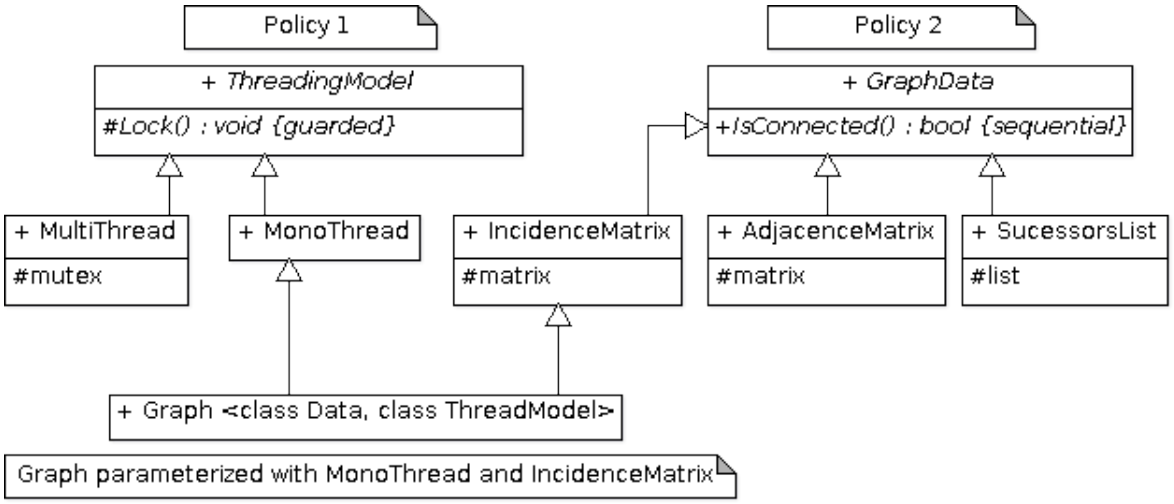


FIGURE 2.15 – Illustration du paramétrage par politique.

```
class MultiThread : public ThreadingModel
{
protected :
    virtual void Lock()
    {
        m.lock();
    }

    std::mutex m;
};

class MonoThread : public ThreadingModel
{
protected :
    virtual void Lock() = default
};

// ... Autres modeles ...
```

Listing 2.2– Définition de la première politique

On définit la seconde politique :

```
class IncidenceMatrix : public GraphRepresentation
{
public :
    virtual bool IsConnected() const
    {
        // Determine si un graphe est connexe
    }
protected :
    std::vector<std::vector<int>> data;
};
```

```
class AdjacenceMatrix : public GraphRepresentation
{
public :
    virtual bool IsConnected() const
    {
        // Determine si un graphe est connexe
    }
protected :
    std::vector<std::vector<bool>> data;
};

// ... Autres representations ...
```

Listing 2.3– Définition de la seconde politique

On définit la classe principale de graphe et on la paramétrise à l'instanciation selon les besoins :

```
template <class Rep = AdjacenceMatrix, class ThreadModel = MonoThread>
class Graph : public Rep, public ThreadModel
{
public :
    bool IsConnected() const
    {
        ThreadModel::Lock();
        return Rep::IsConnected();
    }
};

using GraphInciMT = Graph<IncidenceMatrix, MultiThread>;
using GraphAdjMT = Graph<AdjacenceMatrix, MultiThread>;

// Exemples
Graph a; // Adjacence, MonoThread
GraphInciMT b; // Incidence, MultiThread
GraphAdjMT c; // Adjacence, MultiThread
```

Listing 2.4– Illustration de la paramétrisation

La classe Graph fait appelle à l'aveugle à sa politique de thread ainsi qu'à sa politique de représentation. La bonne écriture d'une politique est guidée par l'API de la classe abstraite en haut de la hiérarchie mais aucune vérification de type n'est effectuée par la classe Graph. Ainsi, un utilisateur pourrait écrire sa propre politique qui ne serait pas basée sur une des classes abstraites.
La partie `using` n'est pas du simple sucre synthaxique puisqu'il contribue également à la maintenabilité de l'application. L'utilisateur final utilise des types en « dur », sans template, ce qui permet, si le besoin s'en fait sentir, de ne changer le template qu'à un unique endroit.

2.9 Notes sur la modélisation

Quelques petites choses ont changé lors de l'implémentation. Notamment, la fabrique pour les algorithmes de plus court chemin a été supprimé puisqu'elle n'avait pas d'intérêt.

En règle générale, des interfaces, sous la forme de classe abstraite (ie. avec des méthodes virtuelles pures) ont été rajoutées pour la plupart des entitées templatées, afin de pouvoir les manipuler de manière plus générique (et éviter la multiplication des templates dans des classes qui n'en ont pas besoin).

Les modélisations des chemins, directions et mouvements n'avaient pas été faites.

2.9.1 Direction

Une direction est une simple paire comprenant l'indice du vecteur de la base canonique de l'environnement (qui peut être à n dimensions) et un booléen indiquant si le vecteur est dans le sens direct ou indirect.
On pourrait imaginer un système plus général où une direction est une combinaison linéaire de vecteurs de la base canonique ce qui permettrait des déplacements autre des déplacements en norme 1.

2.9.2 Mouvement

Un mouvement représente un vecteur. C'est donc une paire comprenant une direction et sa norme.

2.9.3 Chemin

Il s'agit d'une liste de mouvements. Ils seront parcourus dans l'ordre par une unité qui se déplace tout simplement parce le chemin calculé peut dépendre de l'ordre des mouvements (peut être qu'un autre ordre de parcours poserait problème à cause d'obstacles).

Chapitre 3

Plan de tests

3.1 Assignment

3.1.1 Kuhn

On va tester la fonction `operator()`, deux cas sont à tester.
Cas simple où l'on a une ressource pour une tâche, voir deux ressources et deux tâches. 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 ressources pour 3 tâches ; 3 ressources pour 2 tâches...

3.2 Constraint

3.2.1 Constraint

On va tester la fonction `update()` des différentes contraintes existantes. Selon les paramètres de la tâche, on vérifiera si on obtient les bons résultats.

3.3 Environnment

3.3.1 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.

On va également tester les différentes fonctions `addObject()`. Pour cela on va ajouter différents éléments et vérifier que l'on récupère bien ce qu'on a ajouté à l'aide des `getter`.

3.3.2 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.

3.3.3 Resource

Quatre fonctions sont à évaluer.

La première est la fonction `update(double _time)`. Pour cette focntion, 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 resource doivent rester inchangées. Si le temps est positif, les coordonnées de la resource 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`).

3.3.4 Space

Une fonction à tester : `inSpace(Coordonate<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 `faux`.
- Deuxième cas : Les coordonées sont dans l'espace. On va vérifier sur chaque extrémité de l'espace (aux frontières) que la fonction renvoie bien vraie.

3.3.5 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.

3.3.6 TaskSpot

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

3.4 Graph

3.4.1 simpleIndex

On va vérifier que les données fournies aux constructeurs sont bien récupérer dans le vecteur de données.

3.5 IA

3.5.1 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.

3.5.2 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.

Chapitre 4

Implémentation

4.1 Etat du livrable

4.1.1 Licence

Le code source a été placé sous licence CECILL, équivalente française de la GPL. Il interdit l'utilisation commercial du code.

4.1.2 Gestionnaire de version

L'ensemble de l'avancement du projet a été versionné grâce au système Git et un dépôt crée sur Bitbucket.

L'adresse du dépôt est la suivante : https ://bitbucket.org/aquemy/taskassignment

Architecture du dépôt

Le dépôt est constitué des répertoires suivants :
- app : Applications utilisant le framework
- cmake : Fichiers de construction du framework
- include : Fichiers d'entête du framework
- lessons : Leçons d'apprentissage sur le framework
- src : Sources du framework
- test : Ensemble de tests

Quelques fichiers sont présents à la racine, écrit pour la plupart en Markdown : auteurs du projet, guide d'installation, licence, notes et liste des tâches à effectuer.

4.1.3 Systeme de construction

Un système de construction a été mis en place pour le livrable grâce à CMake et les outils gravitants autour. CMake permet de gérer l'ensemble des dépendances, générer les exécutables pour les tests, les binaires de librairies, etc.

Le système de build mis en place est principalement localisé dans le dossier cmake de la racine du projet.

Il comprend les fichiers suivant :
- Config.cmake, qui gère les différentes options de compilation
- Files.cmake, qui liste l'ensemble des fichiers nécessaires à la création de la bibliothèque. Notamment, il liste à la fois les fichiers .cpp ne contenant que l'implémentation mais également les fichiers .hpp des classes templatées qui appellent eux même l'implémentation.
- Macro.cmake, qui comprends les commandes personnalisées. La seule commande disponible permet d'ajouter facilement de nouvelles lessons.
- Package.cmake, qui gère le packaging basique.
- Target.cmake, comprenant des cibles personnalisées, notamment quelques raccourcis d'utilisation.

Le livrable n'étant pas complètement implémenté, le processus d'installation n'a pas été ajouté.

4.1.4 Documentation

La documentation est générée par doxygen grâce à un fichier de configuration présent dans le dossier `doc`.

L'ensemble des rapports et des fichiers de modélisation (sous argoUML) a également été versionné et est présent dans le dossier `doc`.

4.2 Détail de l'implémentation

4.2.1 Les principales fonctionnalités

L'implémentation est très incomplète par rapport à l'ensemble de la modélisation effectuée. Cependant, un certain nombre d'éléments essentiels au principal cycle d'affectation sont présents, permettant de fournir un framework fonctionnel avec un exemple d'application graphique l'utilisant.

Le logger

Le logger est mis en place afin de garder une trace de l'ensemble des étapes de la simulation. Différents niveaux de verbosités sont présents afin de permettre à l'utilisation de gérer le degré d'information qu'il souhaite. De même, un système de sérialisation par fichier permet de régler séparemment les informations à afficher dans le terminal et celles à directement enregistrer dans un fichier.

Les niveaux de logs sont les suivants :
- PROGRESS : Indique une étape de progression de la simulation
- ERROR : Indique une erreur qui va arrêter la simulation
- DEBUG : Affiche des informations supplémentaires utiles pour le débugage
- WARNING : Affiche un avertissement ou une modification de comportement pour éviter une erreur. Par exemple dans le cas où une fonction d'update reçoit une valeur

de temps négative, celle-ci est mise à 0 avant tout traitement et un message de niveau WARNING est envoyé afin de ne pas interrompre la simulation.
- INFO : Informations supplémentaires, généralement liées à une étape de progression. Ces informations ne sont pas liées directements aux résultats de la simulation qui sont plutôt affichées sur PROGRESS.

Quelques exemples d'utilisation :

```
    // Desactivation du mode DEBUG
2   logger.startSerialize("log.txt"); // On sérialize tout dans log.txt
    logger.setQuiet(Logger::PROGRESS); // Pas de message de progression
       affiché
    logger.startSerialize(Logger::ERROR, "error.txt"); // On sérialize
       les erreurs à part
5
    // Exemples
    logger(Logger::PROGRESS) << "Step1 completed.";
8   logger << "Step2 completed";
    logger(Logger::ERROR) << "An error has occurred.";
    logger(Logger::DEBUG) << "Entering function y.";
11  logger(Logger::WARNING) << "Size invalid. Set to 0.";
    logger(Logger::INFO) << "Init IA.";
```

Listing 4.1– Utilisation du logger

Contrôleur

Le contrôleur de l'application est la classe centrale du framework. C'est le point d'entrée de toute simulation.
Il s'agit d'une classe dont toutes les méthodes et membres sont statiques. L'utilisateur définit ses critères d'arrêts de la simulation (en temps ou en étapes pour les critères implémentés), ainsi que les étapes qui doivent être effectuées à chaque tour de boucle. Chaque étape doit être paramétrée par un temps de rafraîchissement. Ainsi, on peut rafraîchir l'environnement toutes les 100ms alors qu'on ne souhaite effectuer un cycle d'affectation toutes les 500ms et un affichage d'informations toutes les secondes.
De plus, chaque étape, lorsqu'elle est appelée, reçoit le temps qui s'est écoulé, en millisecondes, depuis le dernier appel, permettant de gérer correctement le temps réel (par exemple, le déplacement d'une unité ne dépendra pas du temps de rafraichissement de sa méthode d'update, mais uniquement de sa vitesse, le déplacement étant pondéré par la durée depuis le dernier appel de la fonction d'update.

Une fonction d'initialisation qui prend en paramètre l'IA et l'environment de la simulation, permet d'effectuer l'étape de partitionnement et d'indexation et de créer un objet SpatialData invisible à l'utilisateur, qui est passé à tous les objets qui doivent accéder aux informations de l'environment (resources et IA principalement).

Voici un exemple de définition de critères et d'étapes :

```
// Stop criterion
TimeContinue cont(5); // Arrêt de la simulation après 5 minutes
```

```
3 Controller::addContinue(cont);

  // Step Environment
6 unsigned envDelay = 500; // Rafraichissement tout les 500 ms
  // La fonction appelée sera la fonction d'update de l'environnement
  auto envController = bind(&Environment<2,int,int>::update, std::ref(env),
      placeholders::_1);
9 // L'étape est constituée d'une fonction et du temps de rafraichissement
  Step envStep(envController, envDelay);
  Controller::addStep(envStep);
12
  // Step DUMP : On crée une étape pour afficher en détail les informations
      de l'environnement
  // toutes les 2 secondes
15 unsigned dumpDelay = 2000; // ms
  auto dump = bind(&Environment<2,int,int>::dump, std::ref(env));
  Step dumpStep(dump, dumpDelay);
```

Listing 4.2– Paramètres du contrôleur

IA & Modèle

La classe principale représentant l'IA est implémentée. Il ne s'agit que d'un receptacle prenant un modèle.
La classe abstraite représentant un modèle est implémentée ainsi qu'un modèle très basique : simple conteneur de stratégies.

Seule l'étape d'évaluation de la situation et d'affectation est mise en place et ce travail est effectuée par la stratégie courante du modèle. Les étapes d'apprentissage et de décision ne sont pas implémentées, tout simplement par manque de temps et parce que l'objectif du modèle implémenté est d'être manuel (les stratégies peuvent être changées uniquement par demande de l'utilisateur).
Cependant, ces étapes sont indiquées dans le logger pour montrer qu'il ne reste plus qu'à implémenter ces étapes pour d'autres modèles pour lesquels elles sont nécessaires.

Stratégie

La classe abstraite des stratégies est implémentée ainsi qu'une stratégie dite simple.
Cette stratégie recherche la première contrainte non satisfaite et repère la tâche associée. Si cette contrainte existe, elle cherche les emplacements de travails relatifs à cette tâche.
Elle va ensuite chercher toutes les ressources libres et créer tous les couples possibles (ressources libres / emplacements) qu'elle va évaluer en utilisant la distance de manhattan entre les deux entitées de chaque couple.

La liste de couples et leur coût evalué est ensuite envoyé au modèle qui va lui même la transmettre à l'algorithme d'affectation choisi. L'évaluation de la situation est représentée par un entier qui est la somme des coûts de tous les couples. Ainsi, au fur et à mesure que les unités se déplacent vers leur affectation, la stratégie renverra une valeur d'autant plus

faible. C'est assez basique : on pourrait imaginer un estimateur des moindres carrés avec correction par rapport au nombre de couples, etc.

Dimension et politique multithread

Il s'agit plus d'une astuce permettant de faciliter la vie de l'utilisateur qu'une réelle fonctionnalité, d'autant que celle-ci est partiellement implémentée.
La plupart des classes sont templatées avec deux attributs : l'un entier, représentant la dimension de l'espace utilisé pour la simulation, et le second le type des coordonnées utilisées. De même, une politique multithread était envisagée dans la modélisation, ce qui rajouterait un template.

Pour une utilisation donnée, tous ces templates vont être identiques et il est possible grâce à CMake de définir la valeur de ces templates par défault.
Une constante du préprocesseur définie une dimension et un type par défaut, ainsi qu'une politique multithread par défaut. Ces constantes sont ensuite utlisées pour créer un alias sur une classe qui viendra remplacer l'ensemble des arguments par défaut des templates.

L'intérêt est que ces constantes peuvent être redéfinie par l'utilisateur dans son code mais aussi et surtout via CMake au moment de la construction logicielle. Ainsi, si l'on est sur une machine multi-thread on pourra par défault l'activer au moment de la compilation, et si l'on veut principalement travailler en dimension 4 sur des coordonnées entières, on peut compiler l'ensemble du framework pour que ce soit le comportement par défaut et ainsi s'affranchir de l'écriture explicite de tous les paramètres des templates.

Algorithme d'affectation

L'algorithme d'affectation implémenté est la méthode de Khun. Il s'agit d'une méthode en $O(n^4)$ dans sa version implémentée, mais des implémentations existent en $O(n^3)$, notamment en utilisant des graphes bipartis.

L'algorithme va transformer les couples de (ressources / emplacements) en une matrice de coûts et affecter à un emplacement une seule ressource. S'il y a plus de resources que de tâches, elles devront attendre le prochain passage afin d'être affectées. Ce comportement pourrait être amélioré en utilisant l'algorithme plusieurs fois sur des sous-matrices de coûts mais la contrepartie est que s'il y a un nombre de ressources trop important, l'application peut être fortement ralenti puisque l'algorithme a une complexité tout de même élevée. La version de base est donc un bon compris entre la fluidité et l'efficacité.
S'il y a plus d'emplacements que de ressources, les emplacements en trop sont ignorés.

Algorithme d'indexation

Le seul algorithme d'indexation utilisé est l'identité : l'environnement fourni ses conteneurs comportants les différents objets et l'algorithme va les stocker sous forme de conteneurs identiques (et non d'un arbre intéressant pour l'accès à telle ou telle donnée). Il s'agit clairement d'un manque de temps mais l'interface globale des algorithmes est implémentée.

Comme précisé dans les modifications de la modélisation, la fabrique est supprimée et les algorithmes demandés se retrouve être de simples std : :function avec le bon type de retour et les bons arguments.

Cela permet à l'utilisateur de définir ses propres classes qui n'ont rien à voir avec celles de base du framework et de ne pas avoir de hiérarchie de foncteurs héritant d'un foncteur abstrait (ce qui peut poser des problèmes à l'utilisation).

Déplacements basiques

Toujours par manque de temps, seuls des déplacements basiques ont été implémentés, ne tenant pas compte des collisions, d'une part au moment du déplacement des ressources et d'autre part lors du calcul du plus court chemin.

Le plus court chemin ne tient pas compte des obstacles de l'environment, et va simplement créer un chemin en utilisant des mouvements en norme 1. Pour rappel, le mouvement a été ajouté à la modélisation par la suite et est constitué d'une direction et d'une norme. Le chemin étant alors une liste de mouvement.

Les unités se déplacent alors le long du chemin en calculant la distance qu'elles peuvent effectuer à cette itération (vitesse pondérée par le temps depuis la dernière mise à jour) grâce au mouvement courant. Une fois le point indiqué par le mouvement atteint, il est retiré de la liste.

4.2.2 Les défauts de l'implémentation

Beaucoup de choses n'ont pas pu être implémenté par manque de temps.
Certaines choses sont codées en « dur » dans certaines classes (notamment les foncteurs de distance, la structure d'indexation dans SpatialData qui est SimpleIndex, etc.). A chaque fois, une note TODO est présente en commentaire pour rappeler qu'il faudra généraliser le comportement ou le compléter.

Voici une liste des principales fonctionnalités manquantes :

Les obstacles et collisions

Les obstacles n'ont pas été créés, ainsi que les collisions entre les objets dynamiques. Ainsi, les resources, modélisées par un simple pixel, peuvent se chevaucher.
Le manque d'obstacle fait que l'algorithme de plus court chemin est très basique.

Espace

L'espace de l'environnement n'est pas implémenté (en tout cas partiellement mais pas utilisé). Son utilité première est de définir des bornes à l'espace.
Dans la version actuelle, l'espace est infini dans toutes ses dimensions et donc aucune vérification lors de l'ajout d'un objet dans l'environnement n'est effectuée pour savoir s'il est bien dans l'espace.

Partionnement

Comme il n'y a pas d'obstacle, il n'y a pas non plus d'algorithme de partitionnement. Il s'agit là encore d'un manque de temps et la fonctionnalité n'est pas essentielle pour avoir une simulation fonctionnelle. Elle permettrait simplement de réduire le coût de calcul des heuristiques de plus court chemin.

Observateur / Observé

Les classes du patron de conception observateur ne sont pas mises en place malgré que la plupart des classes possèdent une méthode update. La raison est que pour créer un système efficace et générique il aurait fallu plus de temps et donc les méthodes update et de notification sont présentes dans les interfaces des objets directement et pas héritées par des classes spécifiquement dédiées à cela.

Multithreading

Bien que facile à mettre en place dans le cas présent, le parallélisme n'a pas été mis en place. Les interfaces sont présentes mais utilisées nul part. Il ne s'agit pas d'une priorité d'implémentation pour le moment.

Chapitre 5

Application

5.1 Présentation

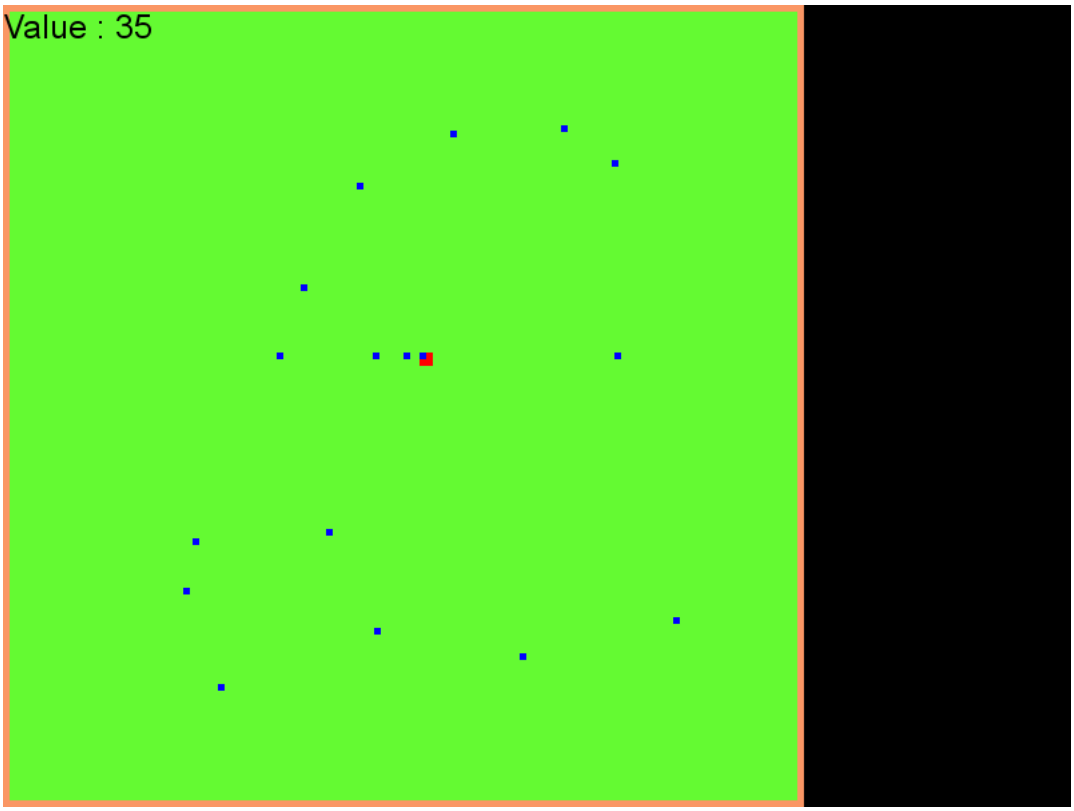
Une application de démonstration a été développée.
Elle se trouve dans le répertoire app/SFML de la racine du projet. Il s'agit gobalement de la même simulation que celle présente dans la leçon 1. Seuls les paramètres changent, et surtout, une interface graphique basique est ajoutée afin de visualiser le processus d'affectation en temps réel.

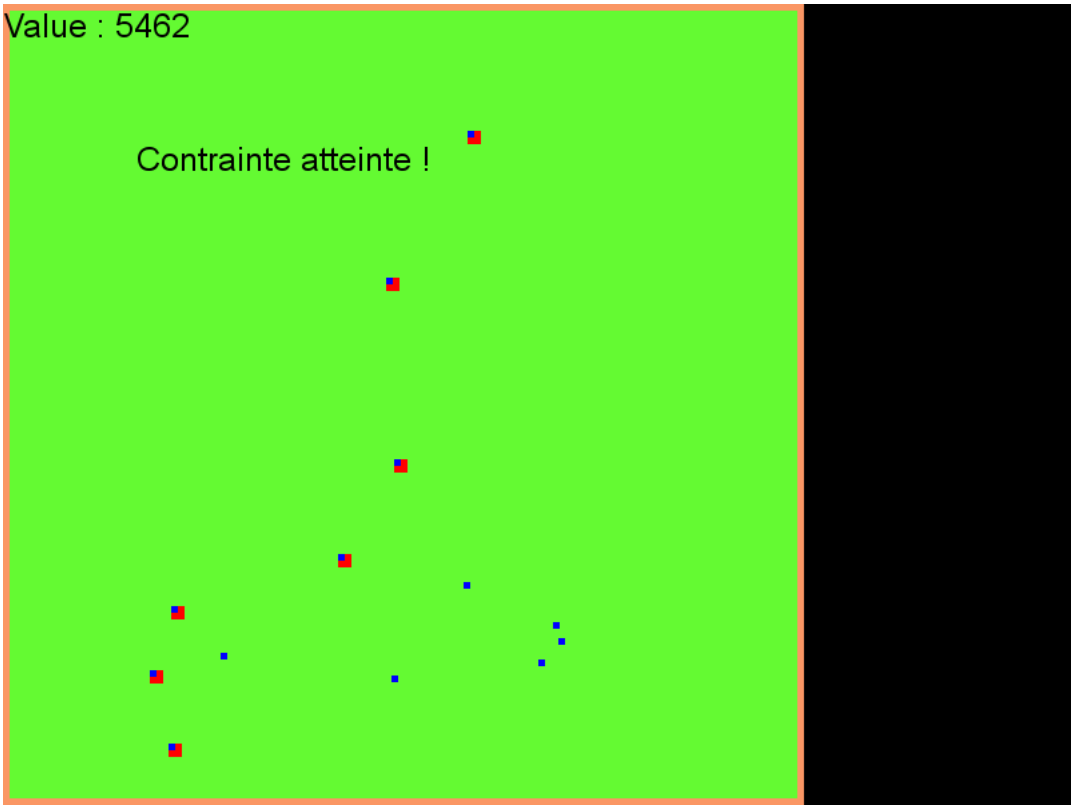
La bibliothèque multimédia utilisée est la SFML, qui est donc requise comme dépendance pour compiler l'application. Pour compiler et utiliser l'application, reportez-vous à la section Guide utilisateur.

La encore, aucun obstacle ne se trouve sur le monde, et il n'existe pas de frontière à l'espace. Il est modélisé à l'écran par un rectangle. Les emplacements de travail sont représentés par des carrés rouges et les carrés bleus représentent les ressources.
En haut à gauche un compte indique la valeur de la tâche définie. Une contrainte est définie sur cette tâche : elle doit être supérieure à 5000.

L'affichage et la gestion des commandes sont gérées par une seule fonction qui est ajoutée au contrôleur principal du framework et a un taux de rafraichissement identique à celui de l'environnement.
Pour faire l'affichage, 3 classes sont partiellement wrappées : environnement, resource et taskspot.

5.2 Captures d'écran


Value : 35


Value : 5462
Contrainte atteinte !

Chapitre 6

Guide utilisateur

6.1 Framework

6.1.1 Compilation

Pour construire et compiler le framework il faut se placer dans un dossier build créé à la racine :

```
> mkdir build
> cd build
```

Il suffit ensuite de lancer CMake puis make. L'option -DINSTALL_TYPE=full permettra de générer également les tests.

```
> cmake .. -DINSTALL_TYPE=full
> make
```

6.1.2 Lancer les tests

Les tests peuvent être lancés tous ensemble via la commande ctest :

```
> ctest
```

Ou indépendamment, en se placant dans le répertoire /build/test et en exécutant chaque test.

6.1.3 Utilisation dans un projet

Comme l'installation n'est pas prévue par CMake, il faudra donner directement le chemin du dossier des includes au compilateur ainsi que linker explicitement à à la librairie générée dans le dossier build.

L'inclusion des headers complet du framework se faisant à l'aide de la commande suivante :

```
#include <sif.hpp>
```

6.1.4 Leçon

Les leçons permettent d'apprendre un aspect du framework. Nous n'avons créée qu'une leçon, qui présente tout le processus de création d'une simulation basique. Celle-ci donne un bon exemple d'utilisation des mécanismes basiques (et d'ailleurs les seuls implémentés !).

Leçon 1 : simulation basique

Tout d'abord nous créons un générateur aléatoire avec une distribution uniforme sur $[0, 10]$ de sorte à placer aléatoirement des objets dans l'environnement.

```
1       // Random engine
        std::random_device rd;
        std::mt19937 gen(rd());
4       std::uniform_int_distribution<> dis(0, 10);
```

Listing 6.1– Générateur aléatoire

Nous créons un simple environnement 2D dont les coordonées seront des `int`. Le dernier template n'est pas utilisé pour l'implémentation à l'heure actuelle.

```
        // Create simple 2D environment
2       Environment<2, int, int> env;
```

Listing 6.2– Environnement

Nous ajoutons quelques ressources de manière aléatoire. Chaque ressource a besoin d'un algorithme de plus court chemin. Nous choisissons l'algorithme le plus simple, qui ne tient pas compte des obstacles.
La valeur 100 correspond à la vitesse de la ressource et nous lui donnons un statut non-occupé par défaut. En dernier lieu nous ajoutons les ressources ainsi créées à l'environnement.

```
1       // Add some resources randomly
        SimpleSP spa;
        Coordonate<2,int> coord;
4
        std::vector<AResource*> res;
        for(unsigned i = 0; i < 5; i++)
7       {
            coord[0] = dis(gen);
            coord[1] = dis(gen);
10          res.push_back(new Resource<2,int,int>(coord, 100, false, spa)
                );
        }

13      env.addObject(res);
```

Listing 6.3– Création de ressources

Nous faisons de même avec des emplacements de travail que nous générons aléatoirement. Pour cela, il nous faut créer une tâche, et tant qu'à faire, une contrainte de seuil dessus qui sera ajoutée dans le système de contraintes.
La contrainte est initialisée avec une priorité 0, utilisant la tâche t qui doit être supérieure à 500.

```
        // Create some tasks and the constraintSystem
        Task t;
        StepConstraint sc(0, t, ConstraintComp::GREATER, 500);
        ConstraintSystem constraintSystem;
        constraintSystem.push(&sc);

        // Add some taskSpots
        std::vector<ATaskSpot*> taskSpots;
        for(unsigned i = 0; i < 5; i++)
        {
            coord[0] = dis(gen);
            coord[1] = dis(gen);
            taskSpots.push_back(new TaskSpot<2,int>(coord, std::ref(t),
                [](int& i, double _time){ return i+0.001*_time; }));
        }

        env.addObject(taskSpots);
```

Listing 6.4– Création des tâches / contraintes et emplacements

Nous créons alors notre IA. Dans un premier temps nous choisisson un algorithme d'affectation pour la seule stratégie que nous donnerons à l'IA. Entre temps il faut créer un modèle qui sert, en temps normal, à agencer les stratégies.

```
        // Assignment
        Kuhn assignment;

        // Strategies creation
        SimpleStrategy<2,int> s1(assignment);

        // Model creation
        ManualModel<2,int> model(s1);

        // IA Creation
        IA<2,int> ia(model, constraintSystem);
```

Listing 6.5– Déclaration de l'IA

La dernière étape consiste à définir les paramètres de la simulation : critères d'arrêts et étapes.

```
        // Stop criterion
        TimeContinue cont(5);
        Controller::addContinue(cont);

        // Step Environment
        unsigned envDelay = 500; // ms
        auto envController = bind(&Environment<2,int,int>::update, std::
            ref(env), placeholders::_1);
        Step envStep(envController, envDelay);
        Controller::addStep(envStep);

        // Step DUMP
```

```
        unsigned dumpDelay = 500; // ms
13      auto dump = bind(&Environment<2,int,int>::dump, std::ref(env));
        Step dumpStep(dump, dumpDelay);
        Controller::addStep(dumpStep);
16
        // Step IA
        unsigned iaDelay = 1000; // ms
19      auto iaController = bind(&IA<2,int>::update, std::ref(ia),
           placeholders::_1);
        Step iaStep(iaController, iaDelay);
        Controller::addStep(iaStep);
```

Listing 6.6– Paramètrage de la simulation

Nous devons initialiser le contrôleur qui se chargera alors de lancer l'indexation et le partitionnement et transmettre les informations récoltées aux entités en ayant besoin.

```
        // Init and launch the simulation
        Controller::init(ia, env);
3       Controller::run();
```

Listing 6.7– Initialisation et lancement du contrôleur

En dernier lieu nous libérons la mémoire liée à la création dynamique des différents objets.

```
        // Free Memory
        for(auto e : taskSpots)
3           delete e;
        for(auto e : res)
            delete e;
```

Listing 6.8– Libération des resources

6.2 Application

6.2.1 Compilation

La compilation de l'application requiert que la SFML 2.0 soit installée. Si c'est le cas et qu'elle se trouve dans les chemins standards, elle peut être trouvé grâce à CMake.

Le site de la SFML est le suivant : http ://www.sfml-dev.org/index-fr.php

Elle peut être compilée via CMake mais se trouve normalement dans la plupart des dépôts des distributions Linux (attention à prendre la version 2.0 et pas 1.6). L'installation n'a pas été vérifiée sous Mac OS X.

Pour construire et compiler l'application en même temps que le framwork, il faut rajouter l'option -DSFML_EXAMPLE lors de l'appel à cmake :

```
> cmake .. -DSFML_EXAMPLE=true
> make
```

6.2.2 Lancement & utilisation

L'application requiert une police d'écriture qui est présente et copiée dans la racine du dossier de construction. Ainsi pour lancer l'application il faut se trouver à la racine du build (et pas dans un dossier parent ou fils) :

```
> ./app/SFML/sifExample <time>
```

La simulation dure <time>secondes et se terminera quoiqu'il arrive après ce temps. Si aucun argument n'est spécifié, la durée est de 25 secondes.
On peut ajouter des emplacements de travail en cliquant avec le bouton droit de la souris et des ressources avec le bouton gauche.
Comme la seule contrainte définie est sur la tâche 1 et requiert que la tâche soit à plus de 5000, une fois la constrainte satisfaite, la stratégie implémentée n'affectera pas les nouvelles ressources créées, même si elles sont libres (un message s'affiche alors à l'écran).

Le contrôleur graphique étant indépendant du contrôleur principal de la simulation, il est possible de faire l'interface graphique sans interrompre la simulation, en appuyant sur la touche echap. On peut également réouvrir l'interface en appuyant sur cette même touche.

Conclusion

Le projet, bien que très incomplet en terme d'implémentation a été très intéressant et enrichissant d'un point de vue génie logiciel. Malgré quelques difficultés sur la modélisation et quelques modifications à effectuer lors de l'implémentation, le projet nous a permis de renforcer une méthodologie efficace de développement.
On regrettera peut-être que la méthode utilisée n'ait pas été incrémentale (mais le temps dédié au projet est certainement trop court pour cela).

La différence de niveau technique en C++ n'a pas du tout été un problème et les échanges techniques ont été intéressant et enrichissant, permettant à l'un d'en apprendre plus en C++ et à l'autre membre du projet de tester de nouvelles choses en terme d'architecture et de techniques.

Le regret lié au manque de temps provient du fait que le projet était à la base calibré pour 4 personnes mais que les effectifs ont été réduit à 2 sans changer le sujet. Ceci dit, il semble que la modélisation tienne globalement la route et pose les bases d'un logiciel correct, malgré une implémentation partielle.
Cependant, chaque module pourrait largement être raffiné, que ce soit la gestion de l'environnement, de l'IA, des contraintes, etc. L'objectif étant plutôt d'amorcer le projet et de fournir un prototype fonctionnel de ce que pourrait être un tel framework, pour le continuer par la suite, en dehors du cours C++.

Spatial Intelligence Framework

Generated by Doxygen 1.8.3.1

Thu May 23 2013 21:46:29

Contents

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

Namespace Documentation

4.1 sif Namespace Reference

Logger : Logger class.

Classes

- class **Assignment**

  Assignment *(*p. **??***) : General class for the assignment.*
- class **AssignmentFactory**

  AssignmentFactory *(*p. **??***) : Instanciate assignment algorithm.*
- class **Kuhn**

  Kuhn *(*p. **??***) : Algorithm which solves the assignment problem.*
- class **Constraint**

  Constraint *(*p. **??***) : General class for constraints.*
- class **ConstraintCompare**

  ConstraintCompare *(*p. **??***) : Functor to compare constraints on priority level.*
- class **ConstraintSystem**

  ConstraintSystem *(*p. **??***) : List of constraints.*
- class **CustomConstraint**

  CustomConstraint *(*p. **??***) : Special constraint.*
- class **StepConstraint**

  StepConstraint *(*p. **??***) : Special constraint.*
- class **ActivePolicy**

  ActivePolicy *(*p. **??***) : Directly notify observer when state change occured.*
- class **Continue**

  Continue *(*p. **??***) : Criterion for ending the calculation / simulation.*
- class **StepsContinue**

  StepsContinue *(*p. **??***) : Criterion for ending the calculation / simulation after a user-defined number of steps.*
- class **TimeContinue**

  TimeContinue *(*p. **??***) : Criterion for ending the calculation / simulation after a user-defined time.*
- class **Controller**

  Controller *(*p. **??***) : Main controller of the application.*
- class **MultiThread**

  MultiThread *(*p. **??***) : Multithread policy.*
- class **Observable**

  Observable *(*p. **??***) : Notify observer when state change occured.*

- class **ObservablePolicy**

  ObservablePolicy *(*p. **??***) : Abstract class to determine observable behavior.*
- class **Observer**

  Observer *(*p. **??***) : Get notification from observable.*
- class **PassivePolicy**

  PassivePolicy *(*p. **??***) : Change the internal state of the observable.*
- class **SingleThread**

  SingleThread *(*p. **??***) : Singlethread policy.*
- class **ThreadingModel**

  ThreadingModel *(*p. **??***) : Threading Policy.*
- class **ACoordonate**

  ACoordonate *(*p. **??***) : Abstract coordonate.*
- class **AEnvironment**

  AEnvironment *(*p. **??***) : Abstract class for environment.*
- class **AObject**

  AObject *(*p. **??***) :*
- class **AResource**

  ARessource : Abstract resource.
- class **ATaskSpot**

  TaskSpot *(*p. **??***) : Physical object that can modify a task.*
- class **Coordonate**

  Coordonate *(*p. **??***) : Cartesian coordonates for n-dimensional space.*
- class **Direction**

  Direction *(*p. **??***) : Vector of an orthonormal base of the space.*
- class **DynamicObject**

  DynamicObject *(*p. **??***) : Objects that can evolve in time.*
- class **Environment**

  Environment *(*p. **??***) : Modelize the environment of the problem.*
- class **Object**

  Object *(*p. **??***) : Abstract class for objects contained in the environment.*
- class **Obstacle**

  Obstacle *(*p. **??***) :* ***Obstacle*** *(*p. **??***) in the environment.*
- class **Resource**

  Ressource : Ressource that can be affected to a task.
- class **Space**

  Space *(*p. **??***) : Defines the space.*
- class **SpatialData**

  SpatialData *(*p. **??***) : Contains the index of the objets and the partition of the space.*
- class **Square**

  Square *(*p. **??***) : A square shaped obstacle.*
- class **StaticObject**

  StaticObject *(*p. **??***) : Objects that cannot evolve in time.*
- class **Task**

  Task *(*p. **??***) : Abstract object.*
- class **TaskSpot**

  TaskSpot *(*p. **??***) : Physical object that can modify a task.*
- class **APath**

  APath *(*p. **??***) : Abstract path.*
- class **AtlasGrid**

  AtlasGrid *(*p. **??***) :* ***Tree*** *(*p. **??***) data structure.*
- class **Path**

Path *(p. **??**) : **Path** (p. **??**) data structure.*

- class **QuadTree**

  QuadTree *(p. **??**) : **Tree** (p. **??**) data structure.*
- class **RTree**

  RTree *(p. **??**) : **Tree** (p. **??**) data structure for indexing multi-dimensional information.*
- class **SimpleIndex**

  Simplexe : Stock information in vector.
- class **Tree**

  Tree *(p. **??**) : General API for tree.*
- class **TreeFactory**

  TreeFactory *(p. **??**) : General API for tree.*
- class **IA**

  IA *(p. **??**) :*
- class **ManualModel**

  ManualModel *(p. **??**) : **Model** (p. **??**) without any decision or learning process.*
- class **Model**

  Model *(p. **??**) : Defines relationship between strategies and how the observation is done.*
- class **SimpleStrategy**

  SimpleStrategy *(p. **??**) : Defines a simple strategy.*
- class **Strategy**

  Strategy *(p. **??**) : Defines the behavior of the **IA** (p. **??**) at a precise moment.*
- class **AStar**

  AStar *(p. **??**) : Computer algorithm that is widely used in pathfinding.*
- class **ShortestPathFactory**

  ShortestPathFactory *(p. **??**) :*
- class **SimpleSP**

  SimplePath.
- class **BaseLogger**

  BaseLogger *(p. **??**) : Element of the reponsability chain of logging.*

Typedefs

- template<class Data >
  using **EvalTable** = std::map< Data *, int >

  Eval : Different types of eval function.
- template<class Data >
  using **Eval** = std::function< int(Data &)>

  Eval : Base for eval function.
- using **ConstraintEval** = **Eval**< **Constraint** >

  ConstraintEval : Typedef for the constraint evaluation.
- using **TaskSpotEval** = **Eval**< **ATaskSpot** >

  *TaskSpotEval : Typedef for the **TaskSpot** (p. **??**) evaluation.*
- using **ResourceEval** = **Eval**< **AResource** >

  *ResourceEval : Typedef for the **Resource** (p. **??**) evaluation.*
- template<int Dim>
  using **Movement** = std::pair< **Direction**< Dim >, unsigned >

  *Movement : Define a movement for a **DynamicObject** (p. **??**).*
- template<int Dim, class Type >
  using **ShortestPath** = std::function< **Path**< Dim > *(const **Coordonate**< Dim, Type > &_from, const **Coordonate**< Dim, Type > &_to)>

  *Shortest **Path** (p. **??**).*

Functions

- template<class Data >
  EvalTable< Data > **EvalLoop** (**Eval**< Data > _eval, std::vector< Data * > &_data)

  EvalLoop : Evaluate a vector of Data.
- const std::string **currentTime** ()

  Get the current time.
- Logger & **operator**<< (Logger &_logger, const std::string _msg)

4.1.1 Detailed Description

Logger : Logger class. The Logger class is based on a Chain of Responsability pattern using logging level. It defines 5 levels of logging that can be manipulated separatly through its interface.

See Also

sif::BaseLogger (p. **??**)

4.1.2 Typedef Documentation

4.1.2.1 template<class Data > using sif::EvalTable = typedef std::map<Data*,int>

Eval : Different types of eval function.

EvalTable : A map of cost indexed by Data pointers

Definition at line 51 of file eval.hpp.

4.1.2.2 template<int Dim> using sif::Movement = typedef std::pair<Direction<Dim>, unsigned>

Movement : Define a movement for a **DynamicObject** (p. **??**).

See Also

sif::Direction (p. **??**)

Definition at line 45 of file movement.hpp.

4.1.2.3 template<int Dim, class Type > using sif::ShortestPath = typedef std::function<Path<Dim>*(const Coordonate<Dim, Type>& _from, const Coordonate<Dim, Type>& _to)>

Shortest **Path** (p. **??**).

Parameters

_from	Origine
_to	Destination

Returns

A **Path** (p. **??**) between _from and _to

Definition at line 49 of file shortestPath.hpp.

4.1.3 Function Documentation

4.1.3.1 template<class Data > EvalTable< Data > sif::EvalLoop (Eval< Data > _eval, std::vector< Data * > & _data)

EvalLoop : Evaluate a vector of Data.

Parameters

_eval	Evaluation function
_data	Vector of data to evaluate

Returns

EvalTable

Definition at line 33 of file eval.cpp.

Referenced by sif::SimpleStrategy< Dim, Type >::evalSituation().

4.1.3.2 const std::string sif::currentTime ( )

Get the current time.

Returns

String according to the current time

Definition at line 34 of file timeManagement.cpp.

Referenced by sif::BaseLogger::print().

4.1.3.3 Logger& sif::operator<< (Logger & _logger, const std::string _msg)

Parameters

_logger	Reference on the Logger
_msg	Message to show

Returns

Reference on the Logger

Definition at line 112 of file logger.cpp.

References sif::BaseLogger::handle().

Chapter 5

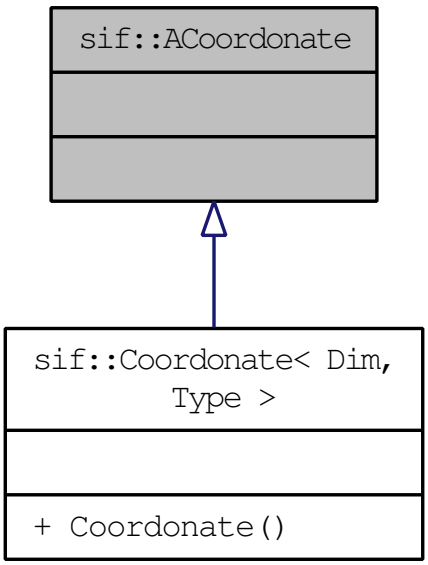
Class Documentation

5.1 sif::ACoordonate Class Reference

ACoordonate (p. **??**) : Abstract coordonate.

`#include <aCoordonate.hpp>`

Inheritance diagram for sif::ACoordonate:



5.1.1 Detailed Description

ACoordonate (p. **??**) : Abstract coordonate.

See Also

sif::Environment (p. **??**), **sif::Space** (p. **??**)

Definition at line 44 of file aCoordonate.hpp.

The documentation for this class was generated from the following file:

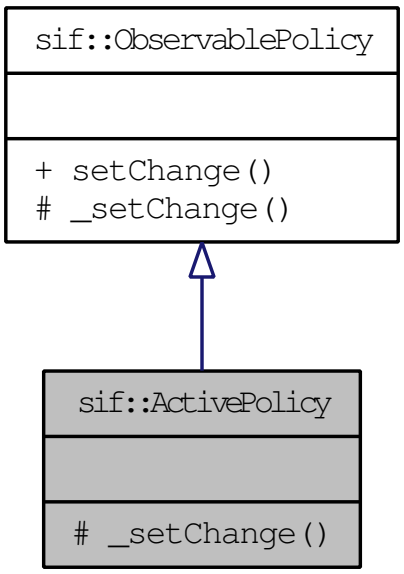
- aCoordonate.hpp

5.2 sif::ActivePolicy Class Reference

ActivePolicy (p. **??**) : Directly notify observer when state change occured.

`#include <activePolicy.hpp>`

Inheritance diagram for sif::ActivePolicy:



Public Member Functions

- virtual void **setChange** () final

  Set change : the meaning of this function depends on the policy.

Protected Member Functions

- virtual void **_setChange** ()

  Implemention of the setChange method.

5.2.1 Detailed Description

ActivePolicy (p. **??**) : Directly notify observer when state change occured.

This policy will notify directly the observer when a state change occured.

See Also

sif::Observer (p. **??**), **sif::ObservablePolicy** (p. **??**), **sif::PassivePolicy** (p. **??**)

Definition at line 44 of file activePolicy.hpp.

The documentation for this class was generated from the following files:

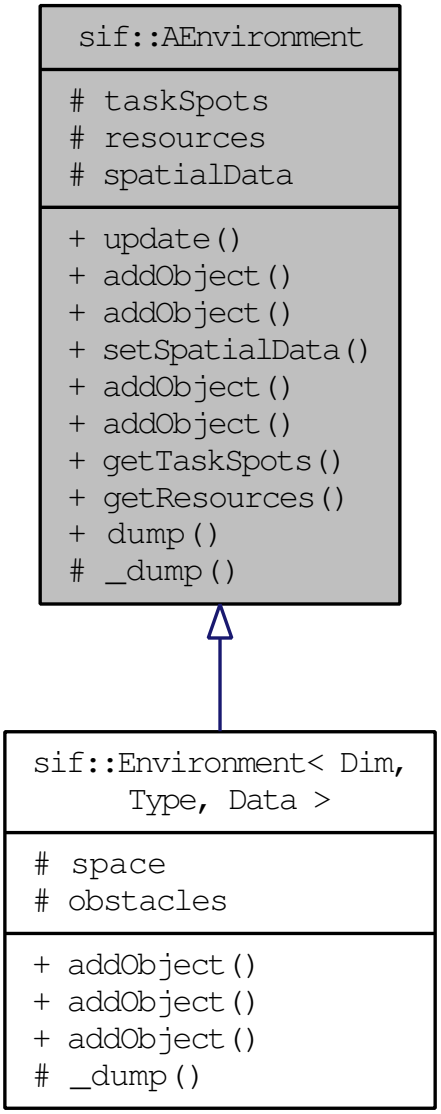
- activePolicy.hpp
- activePolicy.cpp

5.3 sif::AEnvironment Class Reference

AEnvironment (p. **??**) : Abstract class for environment.

`#include <aEnvironment.hpp>`

Inheritance diagram for sif::AEnvironment:



Public Member Functions

- void **update** (double _time)

  Update environment by updating all its components.
- void **addObject** (**AResource** &_resource)

  Add resource to the environment.
- void **addObject** (std::vector< **AResource** * > &_resources)

  Add resources to the environment.
- void **setSpatialData** (**SpatialData** &_spatialData)

  setSpatialData
- void **addObject** (**ATaskSpot** &_taskSpot)

Add taskSpot to the environment.

- void **addObject** (std::vector< **ATaskSpot** * > &_taskSpots)

  Add taskSpots to the environment.
- std::vector< **ATaskSpot** * > & **getTaskSpots** ()

  Get TaskSpots.
- std::vector< **AResource** * > & **getResources** ()

  Get Resources.
- void **dump** ()

  Debug function to show all informations about the class.

Protected Member Functions

- virtual void **_dump** ()=0

  Implementation for dump function.

Protected Attributes

- std::vector< **ATaskSpot** * > **taskSpots**

  TaskSpots of the environment.
- std::vector< **AResource** * > **resources**

  Resources of the environment.
- **SpatialData** * **spatialData**

  SpatialData *(p. **??**).*

5.3.1 Detailed Description

AEnvironment (p. **??**) : Abstract class for environment.

See Also

sif::Environment (p. **??**)

Definition at line 48 of file aEnvironment.hpp.

5.3.2 Member Function Documentation

5.3.2.1 void sif::AEnvironment::update (double *_time*)

Update environment by updating all its components.

Parameters

_time	Ellapsed time since the last update

Definition at line 37 of file aEnvironment.cpp.

References resources.

5.3.2.2 void sif::AEnvironment::addObject (AResource & *_resource*)

Add resource to the environment.

Parameters

_resource	New resource

Definition at line 56 of file aEnvironment.cpp.

References resources.

Referenced by sif::Environment< Dim, Type, Data >::addObject().

5.3.2.3 void sif::AEnvironment::addObject (std::vector< AResource * > & *_resources*)

Add resources to the environment.

Parameters

_resources	Vector of resources

Definition at line 50 of file aEnvironment.cpp.

References resources.

5.3.2.4 void sif::AEnvironment::setSpatialData (SpatialData & *_spatialData*)

setSpatialData

Parameters

_spatialData	New spatialData

Definition at line 62 of file aEnvironment.cpp.

References resources, and spatialData.

5.3.2.5 void sif::AEnvironment::addObject (ATaskSpot & *_taskSpot*)

Add taskSpot to the environment.

Parameters

_taskSpot	New taskSpot

Definition at line 69 of file aEnvironment.cpp.

References taskSpots.

5.3.2.6 void sif::AEnvironment::addObject (std::vector< ATaskSpot * > & *_taskSpots*)

Add taskSpots to the environment.

Parameters

_taskSpots	Vector of taskSpots

Definition at line 74 of file aEnvironment.cpp.

References taskSpots.

5.3.2.7 std::vector< ATaskSpot * > & sif::AEnvironment::getTaskSpots ( )

Get TaskSpots.

Returns

Vector of TaskSpots

Definition at line 80 of file aEnvironment.cpp.

References taskSpots.

Referenced by sif::SpatialData::startIndexing().

5.3.2.8 std::vector< AResource * > & sif::AEnvironment::getResources ( )

Get Resources.

Returns

Vector of Resources

Definition at line 85 of file aEnvironment.cpp.

References resources.

Referenced by sif::SpatialData::startIndexing().

The documentation for this class was generated from the following files:

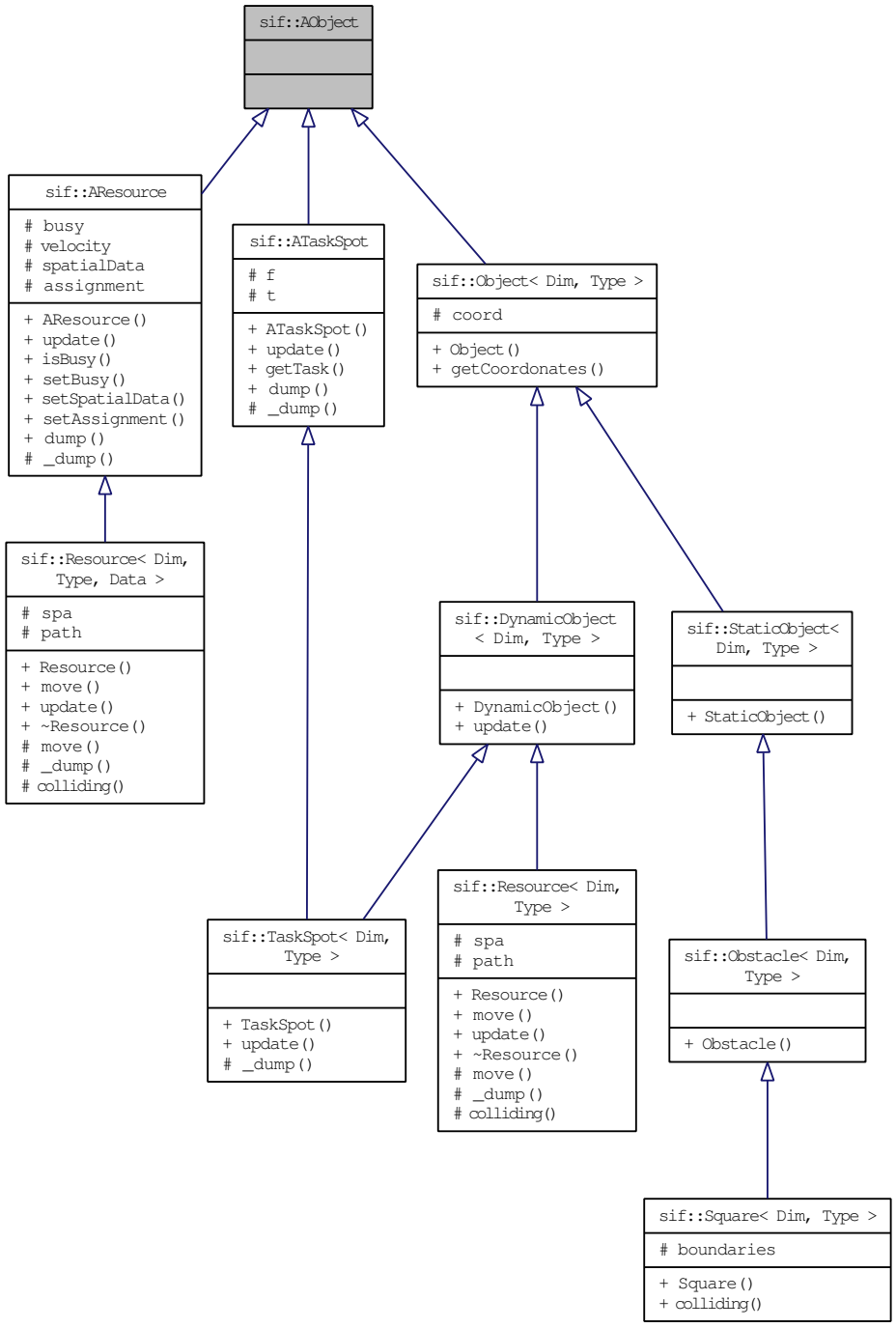
- aEnvironment.hpp
- aEnvironment.cpp

5.4 sif::AObject Class Reference

AObject (p. **??**) :

```
#include <aObject.hpp>
```

Inheritance diagram for sif::AObject:



5.4.1 Detailed Description

AObject (p. **??**) :

See Also

sif::Environment (p. **??**), **sif::Space** (p. **??**), **sif::DynamicObject** (p. **??**), **sif::StaticObject** (p. **??**)

Definition at line 41 of file aObject.hpp.

The documentation for this class was generated from the following file:

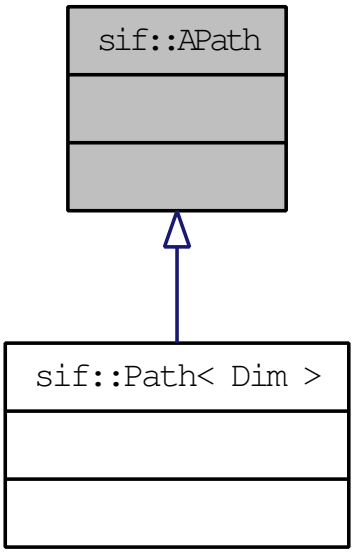
- aObject.hpp

5.5 sif::APath Class Reference

APath (p. **??**) : Abstract path.

`#include <aPath.hpp>`

Inheritance diagram for sif::APath:



5.5.1 Detailed Description

APath (p. **??**) : Abstract path.

See Also

sif::Tree (p. **??**)

Definition at line 42 of file aPath.hpp.

The documentation for this class was generated from the following file:

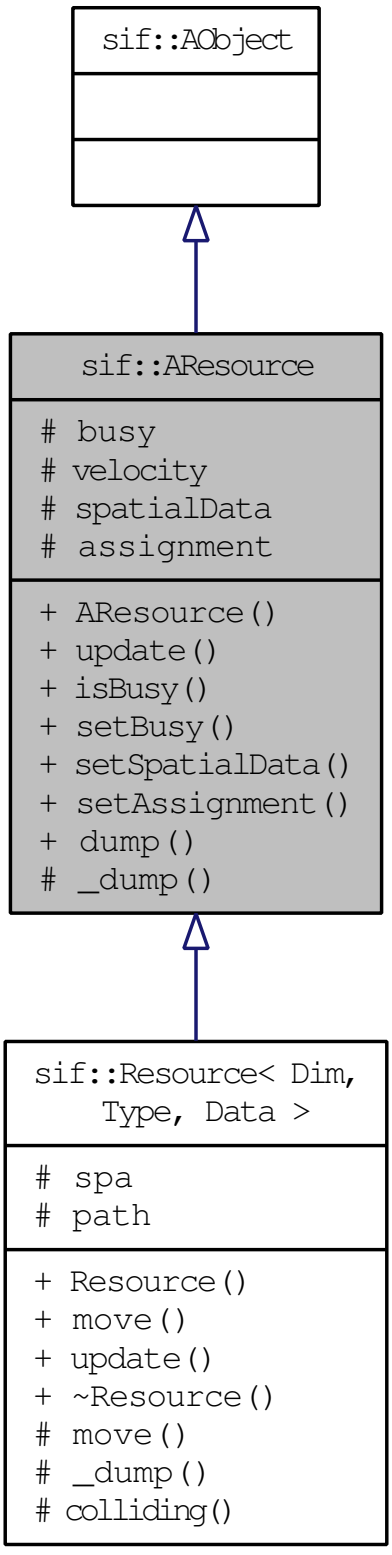
- aPath.hpp

5.6 sif::AResource Class Reference

AResource : Abstract resource.

`#include <aResource.hpp>`

Inheritance diagram for sif::AResource:



Public Member Functions

- **AResource** (double _velocity, bool _busy)

  Constructor.

- virtual void **update** (double _time)=0

  *Update **Resource** (p. **??**).*
- bool **isBusy** () const

  Check if the resource is busy (it cannot be assigned)
- void **setBusy** (bool _status)

  Set the status of the resource.
- void **setSpatialData** (**SpatialData** &_spatialData)

  Set spatialData.
- void **setAssignment** (**ATaskSpot** &_assignment)

  Set a new assignement if it is possible.
- void **dump** ()

  Debug function to show all informations about the class.

Protected Member Functions

- virtual void **_dump** ()=0

  Implementation for dump function.

Protected Attributes

- bool **busy**

  Status of the resource.
- double **velocity**

  Velocity of the ressource.
- **SpatialData** * **spatialData**

  ***SpatialData** (p. **??**).*
- **ATaskSpot** * **assignment**

  ***Assignment** (p. **??**).*

5.6.1 Detailed Description

AResource : Abstract resource.

See Also

sif::Resource (p. **??**)

Definition at line 50 of file aResource.hpp.

5.6.2 Constructor & Destructor Documentation

5.6.2.1 sif::AResource::AResource (double *_velocity*, bool *_busy*)

Constructor.

Parameters

_velocity	Velocity of the resource
_busy	Status of the resource

Definition at line 35 of file aResource.cpp.

5.6.3 Member Function Documentation

5.6.3.1 virtual void sif::AResource::update (double *_time*) [pure virtual]

Update **Resource** (p. **??**).

Parameters

_time	Ellapsed time since the last update

Implemented in **sif::Resource**< **Dim, Type, Data** > (p. **??**).

5.6.3.2 bool sif::AResource::isBusy ( ) const

Check if the resource is busy (it cannot be assigned)

Returns

boolean

Definition at line 41 of file aResource.cpp.

References busy.

5.6.3.3 void sif::AResource::setBusy (bool *_status*)

Set the status of the resource.

Parameters

_status	Bool

Definition at line 46 of file aResource.cpp.

References busy.

Referenced by setAssignment().

5.6.3.4 void sif::AResource::setSpatialData (**SpatialData** & *_spatialData*)

Set spatialData.

Parameters

_spatialData	New **SpatialData** (p. **??**)

Definition at line 51 of file aResource.cpp.

References spatialData.

5.6.3.5 void sif::AResource::setAssignment (**ATaskSpot** & *_assignment*)

Set a new assignement if it is possible.

Parameters

_assignment	New assignment

Definition at line 56 of file aResource.cpp.

References assignment, and setBusy().

The documentation for this class was generated from the following files:

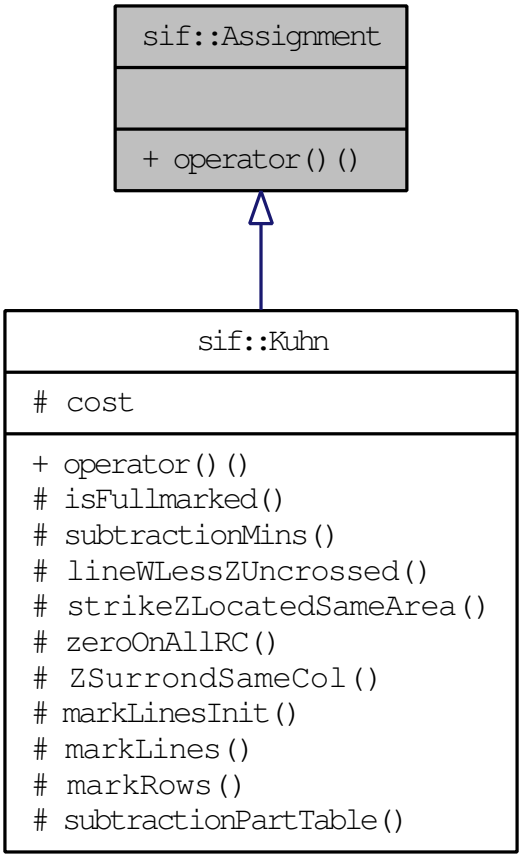
- aResource.hpp
- aResource.cpp

5.7 sif::Assignment Class Reference

Assignment (p. **??**) : General class for the assignment.

`#include <assignment.hpp>`

Inheritance diagram for sif::Assignment:



Public Member Functions

- virtual std::map< **AResource** *, **ATaskSpot** * > **operator()** (std::map< std::pair< **AResource** *, **ATaskSpot** * >, int >)=0

  Start the assignment problem resolution.

5.7.1 Detailed Description

Assignment (p. **??**) : General class for the assignment.

This class enabled to reassign the different units.

See Also

sif::Assignment (p. **??**), **sif::Kuhn** (p. **??**), sif::NearestNeighboor

Definition at line 47 of file assignment.hpp.

5.7.2 Member Function Documentation

5.7.2.1 virtual std::map<AResource*,ATaskSpot*> sif::Assignment::operator() (std::map< std::pair< AResource *, ATaskSpot * >, int >) [pure virtual]

Start the assignment problem resolution.

Returns

A map from resource to **TaskSpot** (p. **??**)

Implemented in **sif::Kuhn** (p. **??**).

The documentation for this class was generated from the following file:

- assignment.hpp

5.8 sif::AssignmentFactory Class Reference

AssignmentFactory (p. **??**) : Instanciate assignment algorithm.

`#include <assignmentFactory.hpp>`

Static Public Member Functions

- static **Kuhn** && **KuhnInstance** (std::map< **AResource** *, int > _resource, std::vector< **ATaskSpot** * > _taskSpot)

  Kuhn *(*p. **??***) instanciation.*

5.8.1 Detailed Description

AssignmentFactory (p. **??**) : Instanciate assignment algorithm.

This factory converts general data to specific data used for a specific assignment algorithm.

See Also

sif::Assignment (p. **??**)

Definition at line 44 of file assignmentFactory.hpp.

5.8.2 Member Function Documentation

5.8.2.1 Kuhn && sif::AssignmentFactory::KuhnInstance (std::map< AResource *, int > *_resource,* std::vector< ATaskSpot * > *_taskSpot*) [static]

Kuhn (p. **??**) instanciation.

Parameters

_resource	Resources to assign
_taskSpot	taskSpot to be affected

Returns

Kuhn (p. **??**) instance (right-value)

Definition at line 34 of file assignmentFactory.cpp.

The documentation for this class was generated from the following files:

- assignmentFactory.hpp
- assignmentFactory.cpp

5.9 sif::AStar Class Reference

AStar (p. **??**) : Computer algorithm that is widely used in pathfinding.

`#include <aStar.hpp>`

Public Member Functions

- **AStar** ()=default

  Default constructor.
- template<int Dim, class Type >
  Path< Dim > * **operator()** (const **Coordonate**< Dim, Type > &_from, const **Coordonate**< Dim, Type > &_to)

  Start the shortest path computation.

5.9.1 Detailed Description

AStar (p. **??**) : Computer algorithm that is widely used in pathfinding.

AStar (p. **??**) is a computer algorithm that is widely used in pathfinding and graph traversal.

See Also

sif::ShortestPath (p. **??**)

Definition at line 44 of file aStar.hpp.

5.9.2 Member Function Documentation

5.9.2.1 template<int Dim, class Type > Path< Dim > * sif::AStar::operator() (const Coordonate< Dim, Type > & *_from*, const Coordonate< Dim, Type > & *_to*)

Start the shortest path computation.

Parameters

_from	Origin
_to	Goal

Returns

A path

Definition at line 35 of file aStar.cpp.

The documentation for this class was generated from the following files:

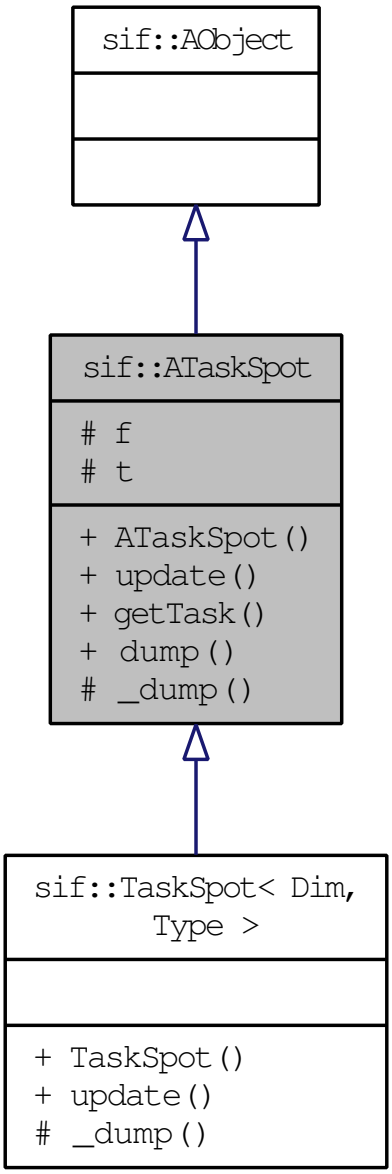
- aStar.hpp
- aStar.cpp

5.10 sif::ATaskSpot Class Reference

TaskSpot (p. **??**) : Physical object that can modify a task.

`#include <aTaskSpot.hpp>`

Inheritance diagram for sif::ATaskSpot:



Public Member Functions

- **ATaskSpot** (**Task** &_t, std::function< int(int &, double _time)> _f)

  Constructor.
- virtual void **update** (double _time)

  Update Ressource.
- const **Task** & **getTask** () const

  *Return **Task** (p. **??**).*
- void **dump** ()

  Debug function to show all informations about the class.

Protected Member Functions

- virtual void **_dump** ()=0

  Implementation for dump function.

Protected Attributes

- std::function< int(int &, double)> **f**

  Function in order to update task.
- **Task** & **t**

  Attached task.

5.10.1 Detailed Description

TaskSpot (p. **??**) : Physical object that can modify a task.

The **TaskSpot** (p. **??**) is an object that can modify a task when a specific action occured. Different types of **TaskSpot** (p. **??**) are provided such as periodic ones.

See Also

sif::Task (p. **??**), sif::PeriodicTaskSpot, **sif::Observable** (p. **??**)

Definition at line 50 of file aTaskSpot.hpp.

5.10.2 Constructor & Destructor Documentation

5.10.2.1 sif::ATaskSpot::ATaskSpot (Task & _t, std::function< int(int &, double _time)> _f)

Constructor.

Parameters

_t	The associated task
_f	Function which serves to update the task

Definition at line 35 of file aTaskSpot.cpp.

The documentation for this class was generated from the following files:

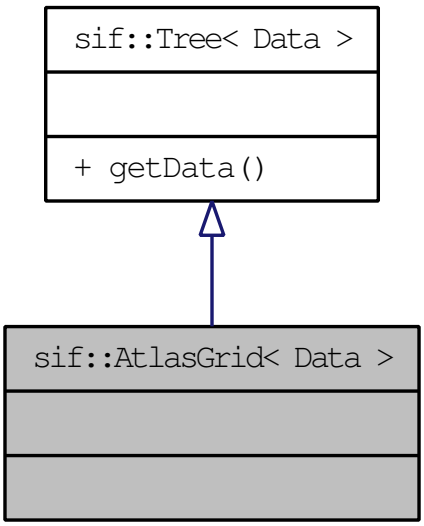
- aTaskSpot.hpp
- aTaskSpot.cpp

5.11 sif::AtlasGrid< Data > Class Template Reference

AtlasGrid (p. **??**) : **Tree** (p. **??**) data structure.

`#include <atlasGrid.hpp>`

Inheritance diagram for sif::AtlasGrid< Data >:



Public Member Functions

- virtual std::vector< Data * > **getData** ()=0

  Return a vector of all objects of the tree.

5.11.1 Detailed Description

template<class Data>class sif::AtlasGrid< Data >

AtlasGrid (p. **??**) : **Tree** (p. **??**) data structure.

AtlasGrid (p. **??**) is a simple data structure for partitionning..

See Also

sif::Tree (p. **??**)

Definition at line 48 of file atlasGrid.hpp.

5.11.2 Member Function Documentation

5.11.2.1 template<class Data> virtual std::vector<Data*> sif::Tree< Data >::getData ( ) [pure virtual], [inherited]

Return a vector of all objects of the tree.

Returns

Vector of objets

Implemented in **sif::SimpleIndex< Data >** (p. **??**), **sif::SimpleIndex< sif::ATaskSpot >** (p. **??**), and **sif::SimpleIndex< sif::AResource >** (p. **??**).

The documentation for this class was generated from the following file:

- atlasGrid.hpp

5.12 sif::BaseLogger Class Reference

BaseLogger (p. **??**) : Element of the reponsability chain of logging.

`#include <baseLogger.hpp>`

Public Member Functions

- **BaseLogger** (int _level, std::string _label, std::ostream &_os=std::clog, bool _quiet=false)

  Constructor.
- **BaseLogger** (int _level, std::string _label, std::string _logFile, std::ostream &_os=std::clog, bool _quiet=false)

  Constructor.
- ∼**BaseLogger** ()

  Destructor.
- **BaseLogger** ∗ **setNext** (**BaseLogger** ∗_next)

  Set next element in reponsability chain.
- **BaseLogger** ∗ **getNext** () const

  Get next element in reponsability chain.
- void **handle** (const std::string _msg, int _priority)

  Try to handle the message.
- void **startSerialize** (int _level, std::string _logFile)

  Start to serialize.
- void **startSerialize** (std::string _logFile)

  Start to serialize (all levels)
- void **stopSerialize** (int _level)

  Stop to serialize.
- void **stopSerialize** ()

  Start to serialize (all levels)
- void **setQuiet** ()

  Stop writting on printing stream (all levels)
- void **setQuiet** (int _level)

  Stop writting on printing stream.
- void **setVerbose** ()

  Start writting on printing stream (all levels)
- void **setVerbose** (int _level)

  Start writting on printing stream.
- void **print** (const std::string _msg, std::ostream ∗_stream)

  Print message on the specified stream.
- void **log** (std::string _msg)

  Log the message and dispatch on streams.

5.12.1 Detailed Description

BaseLogger (p. **??**) : Element of the reponsability chain of logging.

The **BaseLogger** (p. **??**) class define an element of the logger. It defines a level of logging and have two streams, one for printing and one for serialize logs in a specified file.

See Also

sif::Logger

Definition at line 46 of file baseLogger.hpp.

5.12.2 Constructor & Destructor Documentation

5.12.2.1 sif::BaseLogger::BaseLogger (int *_level,* std::string *_label,* std::ostream & *_os* = `std::clog`, bool *_quiet* = `false`)

Constructor.

Parameters

_level	Level of logging
_label	Label of the level
_os	Printing stream
_quiet	Quiet mode

Definition at line 39 of file baseLogger.cpp.

5.12.2.2 sif::BaseLogger::BaseLogger (int *_level,* std::string *_label,* std::string *_logFile,* std::ostream & *_os* = `std::clog`, bool *_quiet* = `false`)

Constructor.

Parameters

_level	Level of logging
_label	Label of the level
_logFile	Log file for serialization
_os	Printing stream
_quiet	Quiet mode

Definition at line 52 of file baseLogger.cpp.

5.12.3 Member Function Documentation

5.12.3.1 BaseLogger * sif::BaseLogger::setNext (BaseLogger * *_next*)

Set next element in reponsability chain.

Parameters

_next	Pointer on next element

Returns

This element

Definition at line 66 of file baseLogger.cpp.

References getNext().

5.12.3.2 BaseLogger * sif::BaseLogger::getNext ( ) const

Get next element in reponsability chain.

Returns

Pointer on next element

Definition at line 72 of file baseLogger.cpp.

Referenced by setNext().

5.12.3.3 void sif::BaseLogger::handle (const std::string *_msg*, int *_priority*)

Try to handle the message.

Parameters

_msg	Message to handle
_priority	Level of the message

Definition at line 77 of file baseLogger.cpp.

References handle().

Referenced by handle(), and sif::operator<<().

5.12.3.4 void sif::BaseLogger::startSerialize (int *_level*, std::string *_logFile*)

Start to serialize.

Parameters

_level	Level of logging
_logFile	Log file for serialization

Definition at line 85 of file baseLogger.cpp.

References startSerialize().

Referenced by startSerialize().

5.12.3.5 void sif::BaseLogger::startSerialize (std::string *_logFile*)

Start to serialize (all levels)

Parameters

_logFile	Log file for serialization

Definition at line 106 of file baseLogger.cpp.

References startSerialize().

5.12.3.6 void sif::BaseLogger::stopSerialize (int _level)

Stop to serialize.

Parameters

_level	Level of logging

Definition at line 124 of file baseLogger.cpp.

References stopSerialize().

Referenced by stopSerialize().

5.12.3.7 void sif::BaseLogger::setQuiet (int _level)

Stop writting on printing stream.

Parameters

_level	Level of logging

Definition at line 146 of file baseLogger.cpp.

References setQuiet().

5.12.3.8 void sif::BaseLogger::setVerbose (int _level)

Start writting on printing stream.

Parameters

_level	Level of logging

Definition at line 161 of file baseLogger.cpp.

References setVerbose().

5.12.3.9 void sif::BaseLogger::print (const std::string _msg, std::ostream * _stream)

Print message on the specified stream.

Parameters

_msg	Message to write
_stream	Stream

Definition at line 169 of file baseLogger.cpp.

References sif::currentTime().

Referenced by log().

5.12.3.10 void sif::BaseLogger::log (std::string _msg)

Log the message and dispatch on streams.

Parameters

_msg	Message to log

Definition at line 181 of file baseLogger.cpp.

References print().

The documentation for this class was generated from the following files:

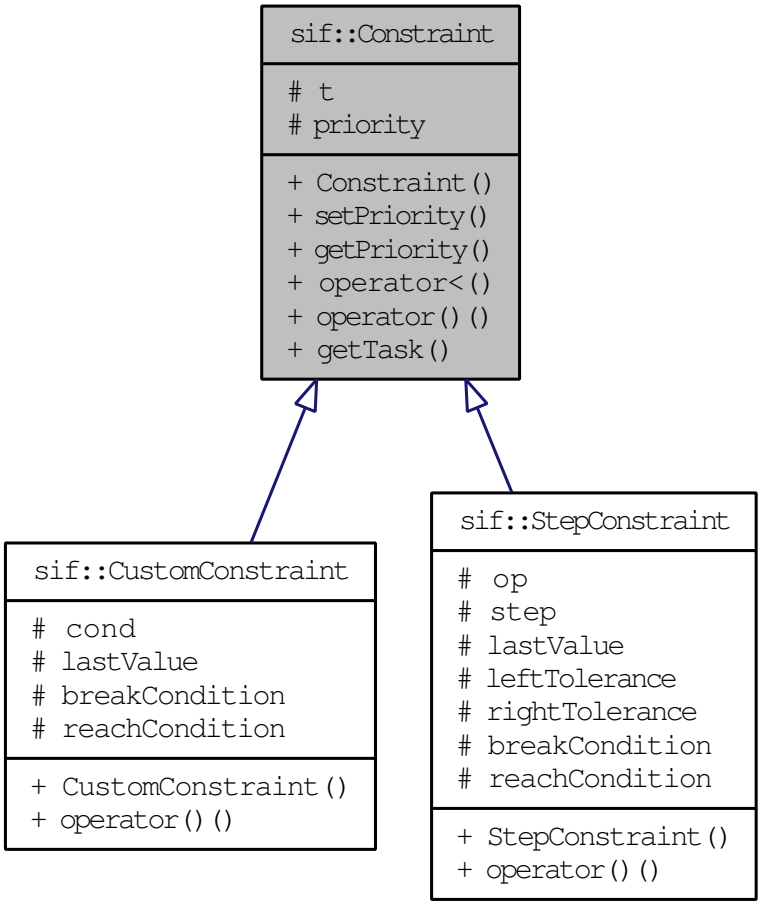
- baseLogger.hpp
- baseLogger.cpp

5.13 sif::Constraint Class Reference

Constraint (p. **??**) : General class for constraints.

`#include <constraint.hpp>`

Inheritance diagram for sif::Constraint:



Public Member Functions

- **Constraint** (**Task** &_t, unsigned _priority)

  Default constructor.
- void **setPriority** (unsigned _priority)

  Change the priority.

- unsigned **getPriority** () const

  Get the priority.
- bool **operator**< (**Constraint** &_const)

  Comparaison operator based on priority.
- virtual bool **operator()** ()=0

  Check the constraint satisfaction and launch callback function if needed.
- const **Task** & **getTask** () const

  Get task.

Protected Attributes

- **Task** & **t**

  Task *(*p. **??***) concerned by the constraint.*
- unsigned **priority**

  Priority of the constraint.

5.13.1 Detailed Description

Constraint (p. **??**) : General class for constraints.

General class for constraints, has the level of the priority (int). Several types of constraints exist.

See Also

sif::ConstraintSystem (p. **??**), **sif::StepConstraint** (p. **??**), sif::PropConstraint, **sif::CustomConstraint** (p. **??**)

Definition at line 44 of file constraint.hpp.

5.13.2 Constructor & Destructor Documentation

5.13.2.1 sif::Constraint::Constraint (Task & *_t*, unsigned *_priority*)

Default constructor.

Parameters

_t	**Task** (p. **??**) which is concerned by the constraint
_priority	**Constraint** (p. **??**) priority

Definition at line 34 of file constraint.cpp.

5.13.3 Member Function Documentation

5.13.3.1 void sif::Constraint::setPriority (unsigned *_priority*)

Change the priority.

Parameters

_priority	**Constraint** (p. **??**) priority

Definition at line 39 of file constraint.cpp.

References priority.

5.13.3.2 unsigned sif::Constraint::getPriority ( ) const

Get the priority.

Returns

Constraint (p. **??**) priority

Definition at line 44 of file constraint.cpp.

References priority.

Referenced by operator<().

5.13.3.3 bool sif::Constraint::operator< (Constraint & *_const*)

Comparaison operator based on priority.

Parameters

_const	**Constraint** (p. **??**) to compare

Returns

boolean

Definition at line 49 of file constraint.cpp.

References getPriority(), and priority.

5.13.3.4 virtual bool sif::Constraint::operator() ( ) [pure virtual]

Check the constraint satisfaction and launch callback function if needed.

Returns

boolean

Implemented in **sif::StepConstraint** (p. **??**), and **sif::CustomConstraint** (p. **??**).

5.13.3.5 const Task & sif::Constraint::getTask ( ) const

Get task.

Returns

Associated task

Definition at line 59 of file constraint.cpp.

References t.

Referenced by sif::SimpleStrategy< Dim, Type >::SimpleStrategy().

The documentation for this class was generated from the following files:

- constraint.hpp
- constraint.cpp

5.14 sif::ConstraintCompare Class Reference

ConstraintCompare (p. **??**) : Functor to compare constraints on priority level.

`#include <constraint.hpp>`

5.14.1 Detailed Description

ConstraintCompare (p. **??**) : Functor to compare constraints on priority level.

Functor to compare constraints on priority level

See Also

sif::ConstraintSystem (p. **??**), **sif::StepConstraint** (p. **??**), sif::PropConstraint, **sif::CustomConstraint** (p. **??**)

Definition at line 100 of file constraint.hpp.

The documentation for this class was generated from the following files:

- constraint.hpp
- constraint.cpp

5.15 sif::ConstraintSystem Class Reference

ConstraintSystem (p. **??**) : List of constraints.

`#include <constraintSystem.hpp>`

Inherits std::priority_queue< T >.

Public Attributes

- T **elements**

  STL member.

5.15.1 Detailed Description

ConstraintSystem (p. **??**) : List of constraints.

ConstraintSystem (p. **??**) is a container of constraints. This list is sorted by priority level. At the moment it is a simple priority queue.

See Also

sif::Constraint (p. **??**)

Definition at line 48 of file constraintSystem.hpp.

The documentation for this class was generated from the following file:

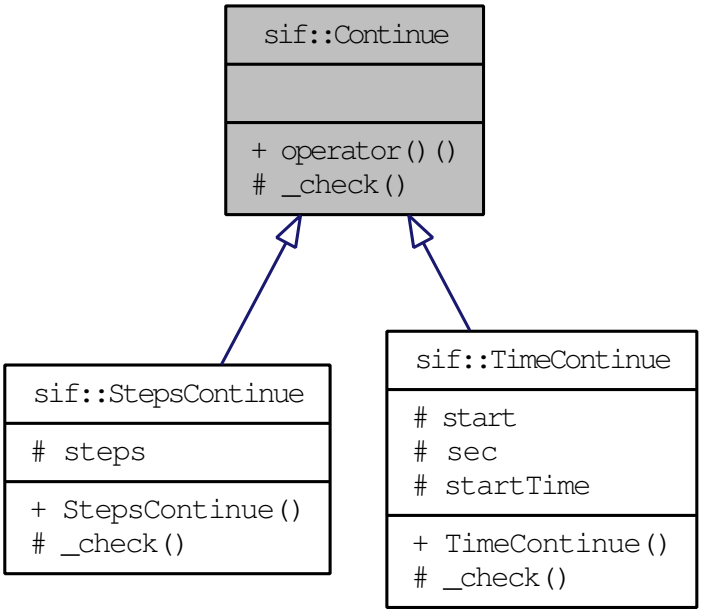
- constraintSystem.hpp

5.16 sif::Continue Class Reference

Continue (p. **??**) : Criterion for ending the calculation / simulation.

`#include <continue.hpp>`

Inheritance diagram for sif::Continue:



Public Member Functions

- virtual bool **operator()** ()

  Check the criterion.

Protected Member Functions

- virtual bool **_check** ()=0

  Implementation of the test.

5.16.1 Detailed Description

Continue (p. **??**) : Criterion for ending the calculation / simulation.

Hierarchie of functors that defines criteria in order to continue the calculation / simulation. The implementation is based on NVI idiom (C++ implementation of the Design Pattern Template Method).

See Also

sif::StepsContinue (p. **??**), **sif::TimeContinue** (p. **??**), **sif::Controller** (p. **??**)

Definition at line 42 of file continue.hpp.

5.16.2 Member Function Documentation

5.16.2.1 bool sif::Continue::operator() ( ) [virtual]

Check the criterion.

Returns

boolean

Definition at line 34 of file continue.cpp.

References _check().

5.16.2.2 virtual bool sif::Continue::_check ( ) [protected], [pure virtual]

Implementation of the test.

Returns

boolean

Implemented in **sif::TimeContinue** (p. **??**), and **sif::StepsContinue** (p. **??**).

Referenced by operator()().

The documentation for this class was generated from the following files:

- continue.hpp
- continue.cpp

5.17 sif::Controller Class Reference

Controller (p. **??**) : Main controller of the application.

`#include <controller.hpp>`

Public Member Functions

- **Controller** ()
  Default contructor.
- ∼**Controller** ()
  Default destructor.

Static Public Member Functions

- static void **setEnvironment** (**AEnvironment** &_env)
  Set the environment.
- static void **addContinue** (**Continue** &_cont)
  Add a stop criterion.
- static void **addStep** (Step _step)
  Add a step to the controller, at the end of the list.
- static void **addStep** (Step _step, unsigned _pos)
  Add a step to the controller.
- template<int Dim, class Type , class Data >
  static void **init** (**IA**< Dim, Type > &_ia, **Environment**< Dim, Type, Data > &_env)
  Static initialization.

- static void **startPartitioning** ()

  Start the space partitioning.
- static void **startIndexing** ()

  Start the space indexing.
- static void **run** ()

  Run the simulation / computation.

Static Protected Member Functions

- static bool **checkContinue** ()

  Check if a stop criterion is reached.

Static Protected Attributes

- static std::vector< **Continue** * > **cont**

  Continuators.
- static std::list< std::pair
  < Step, unsigned > > **steps**

  Functions to call during the main loop.
- static **SpatialData** * **spatialData**

  SpatialData *(*p. **??***) information.*
- static bool **initialized** = false

  True if the controller has been initialized.
- static **AEnvironment** * **env**

  Environment *(*p. **??***) of the simulation.*

5.17.1 Detailed Description

Controller (p. **??**) : Main controller of the application.

The controller is a static class that is the entry point of the execution flux.

Definition at line 51 of file controller.hpp.

5.17.2 Member Function Documentation

5.17.2.1 void sif::Controller::setEnvironment (AEnvironment & *_env*) [static]

Set the environment.

Parameters

_env	The environment of the simulation

Definition at line 55 of file controller.cpp.

References env.

5.17.2.2 void sif::Controller::addContinue (Continue & *_cont*) [static]

Add a stop criterion.

Parameters

_cont	Criterion

Definition at line 60 of file controller.cpp.

References cont.

5.17.2.3 void sif::Controller::addStep (Step *_step*) [static]

Add a step to the controller, at the end of the list.

Parameters

_step	Step to add

Definition at line 65 of file controller.cpp.

References steps.

5.17.2.4 void sif::Controller::addStep (Step *_step,* unsigned *_pos*) [static]

Add a step to the controller.

Parameters

_step	Step to add
_pos	Position of the step in the list

Definition at line 70 of file controller.cpp.

References steps.

5.17.2.5 template<int Dim, class Type , class Data > static void sif::Controller::init (IA< Dim, Type > & *_ia,* Environment< Dim, Type, Data > & *_env*) [static]

Static initialization.

Parameters

_ia	**IA** (p. **??**) for the simulation
_env	**Environment** (p. **??**) for the simulation

5.17.2.6 bool sif::Controller::checkContinue ( ) [static], [protected]

Check if a stop criterion is reached.

Returns

boolean

Definition at line 149 of file controller.cpp.

References cont.

Referenced by run().

The documentation for this class was generated from the following files:

- controller.hpp

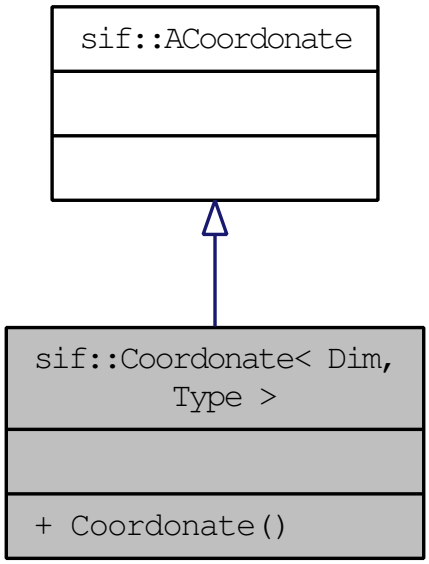
- controller.cpp

5.18 sif::Coordonate< Dim, Type > Class Template Reference

Coordonate (p. **??**) : Cartesian coordonates for n-dimensional space.

`#include <coordonate.hpp>`

Inheritance diagram for sif::Coordonate< Dim, Type >:



Public Member Functions

- **Coordonate** ()

  Default constructor Check the Type template is a numeric type with type_trait.

Public Attributes

- T **elements**

  STL member.

5.18.1 Detailed Description

template<int Dim = DEF_DIM, class Type = DEF_COORD_TYPE>class sif::Coordonate< Dim, Type >

Coordonate (p. **??**) : Cartesian coordonates for n-dimensional space.

The coordonate object is designed to describe the space. First template represents the number of dimensions and the second one represents the type of each coordonate. This allows only cartesian system and uniform type representation at the moment. Some inherited functions from std::vector have been deleted such as push_back and other size-related functions.

See Also

sif::Environment (p. **??**), **sif::Space** (p. **??**)

Definition at line 51 of file coordonate.hpp.

The documentation for this class was generated from the following files:

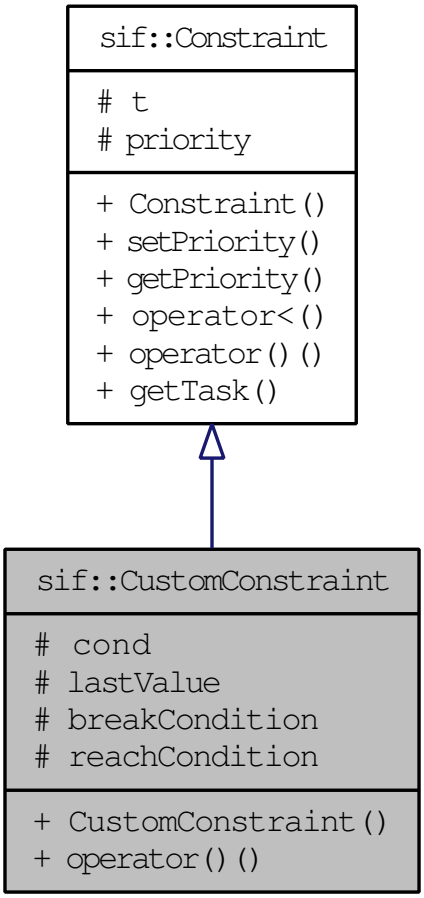
- coordonate.hpp
- coordonate.cpp

5.19 sif::CustomConstraint Class Reference

CustomConstraint (p. **??**) : Special constraint.

`#include <customConstraint.hpp>`

Inheritance diagram for sif::CustomConstraint:



Public Member Functions

- **CustomConstraint** (**Task** &_t, unsigned _priority, std::function< bool()> _cond, std::function< void()> _reachCondition=[]{}, std::function< void()> _breakCondition=[]{})

  Default constructor.
- virtual bool **operator()** ()

  Check the constraint satisfaction and launch callback function if needed.

- void **setPriority** (unsigned _priority)

  Change the priority.
- unsigned **getPriority** () const

  Get the priority.
- bool **operator**< (**Constraint** &_const)

  Comparaison operator based on priority.
- const **Task** & **getTask** () const

  Get task.

Protected Attributes

- std::function< bool()> **cond**

  Constraint *(*p. **??***) satisfaction.*
- bool **lastValue**

  Value at the last check.
- std::function< void()> **breakCondition**

  Called when the constraint is not satisfied.
- std::function< void()> **reachCondition**

  Called when the constraint is satisfied.
- **Task** & **t**

  Task *(*p. **??***) concerned by the constraint.*
- unsigned **priority**

  Priority of the constraint.

5.19.1 Detailed Description

CustomConstraint (p. **??**) : Special constraint.

CustomConstraint (p. **??**) is a special constraint defined by the user.

See Also

sif::Constraint (p. **??**)

Definition at line 46 of file customConstraint.hpp.

5.19.2 Constructor & Destructor Documentation

5.19.2.1 sif::CustomConstraint::CustomConstraint (Task & *_t*, unsigned *_priority*, std::function< bool()> *_cond*, std::function< void()> *_reachCondition* = [ ]{}, std::function< void()> *_breakCondition* = [ ]{})

Default constructor.

Parameters

_t	**Task** (p. **??**) which is concerned by the constraint
_priority	Priority of the constraint
_cond	Condition to satisfy the constraint
_reachCondition	Callback function called when the constraint is satisfied
_breakCondition	Callback function called when the constraint is not satisfied

Definition at line 34 of file customConstraint.cpp.

5.19.3 Member Function Documentation

5.19.3.1 bool sif::CustomConstraint::operator() ( ) [virtual]

Check the constraint satisfaction and launch callback function if needed.

Returns

boolean

Implements **sif::Constraint** (p. **??**).

Definition at line 47 of file customConstraint.cpp.

References breakCondition, cond, lastValue, and reachCondition.

5.19.3.2 void sif::Constraint::setPriority (unsigned *_priority*) [inherited]

Change the priority.

Parameters

_priority	**Constraint** (p. **??**) priority

Definition at line 39 of file constraint.cpp.

References sif::Constraint::priority.

5.19.3.3 unsigned sif::Constraint::getPriority ( ) const [inherited]

Get the priority.

Returns

Constraint (p. **??**) priority

Definition at line 44 of file constraint.cpp.

References sif::Constraint::priority.

Referenced by sif::Constraint::operator<().

5.19.3.4 bool sif::Constraint::operator< (Constraint & *_const*) [inherited]

Comparaison operator based on priority.

Parameters

_const	**Constraint** (p. **??**) to compare

Returns

boolean

Definition at line 49 of file constraint.cpp.

References sif::Constraint::getPriority(), and sif::Constraint::priority.

5.19.3.5 const Task & sif::Constraint::getTask ( ) const [inherited]

Get task.

Returns

Associated task

Definition at line 59 of file constraint.cpp.

References sif::Constraint::t.

Referenced by sif::SimpleStrategy< Dim, Type >::SimpleStrategy().

The documentation for this class was generated from the following files:

- customConstraint.hpp
- customConstraint.cpp

5.20 sif::Direction< Dim > Class Template Reference

Direction (p. **??**) : Vector of an orthonormal base of the space.

`#include <direction.hpp>`

Public Member Functions

- **Direction** (unsigned _pos, bool _way)
  Direction *(*p. **??***).*
- std::pair< int, bool > **getValue** () const
  getValue

Protected Attributes

- std::pair< int, bool > **dir**
  Direction *(*p. **??***) information.*

5.20.1 Detailed Description

template<int Dim>class sif::Direction< Dim >

Direction (p. **??**) : Vector of an orthonormal base of the space.

See Also

sif::Environment (p. **??**), **sif::Object** (p. **??**), sif::Ressource

Definition at line 43 of file direction.hpp.

5.20.2 Constructor & Destructor Documentation

5.20.2.1 template<int Dim> sif::Direction< Dim >::Direction (unsigned _pos, bool _way)

Direction (p. **??**).

Parameters

_pos	Position of the vector in the base
_way	Indicate the way

Definition at line 37 of file direction.cpp.

5.20.3 Member Function Documentation

5.20.3.1 template<int Dim> std::pair< int, bool > sif::Direction< Dim >::getValue ( ) const

getValue

Returns

Direction (p. **??**) information

Definition at line 47 of file direction.cpp.

Referenced by sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

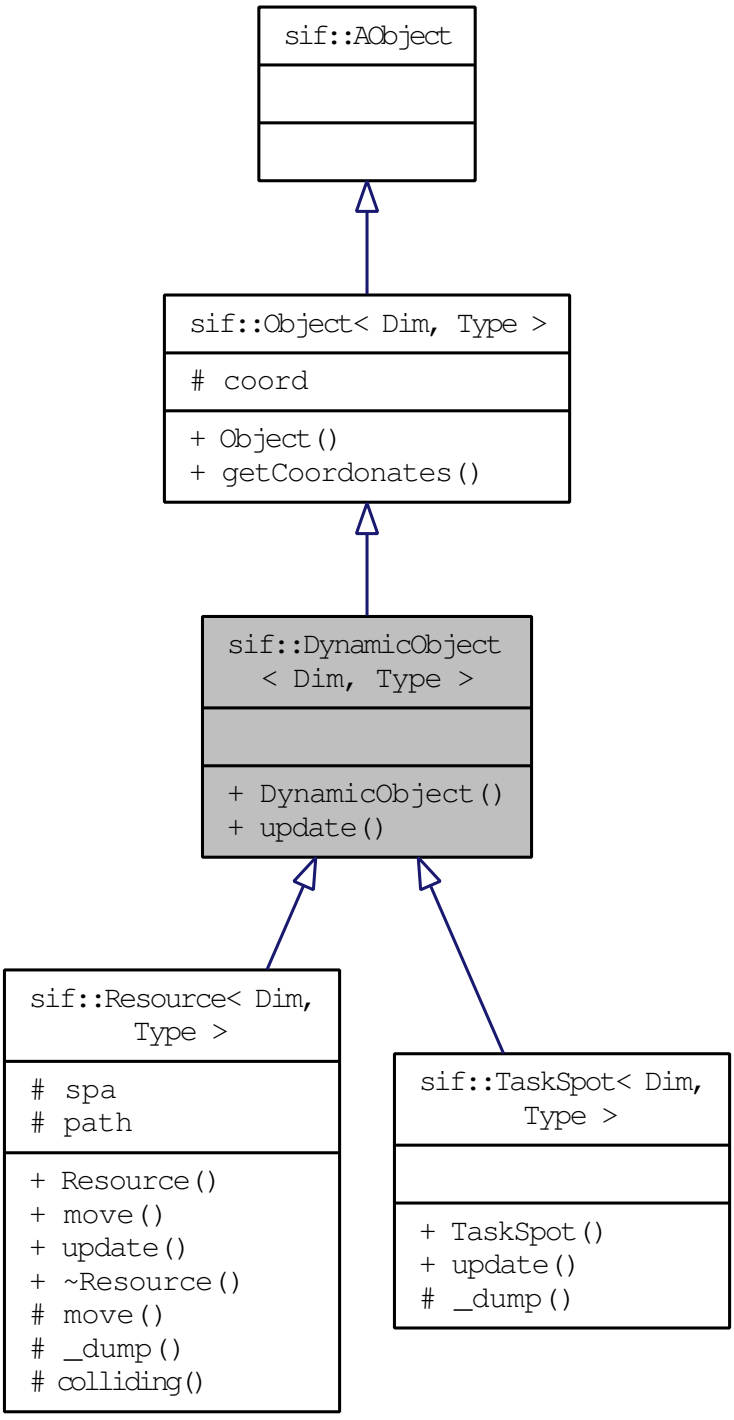
- direction.hpp
- direction.cpp

5.21 sif::DynamicObject< Dim, Type > Class Template Reference

DynamicObject (p. **??**) : Objects that can evolve in time.

`#include <dynamicObject.hpp>`

Inheritance diagram for sif::DynamicObject< Dim, Type >:



Public Member Functions

- **DynamicObject** (**Coordonate**< Dim, Type > _coord)

  Constructor.
- virtual void **update** (double _time)=0

  Update dynamic object.
- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Attributes

- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.21.1 Detailed Description

template<int Dim, class Type>class sif::DynamicObject< Dim, Type >

DynamicObject (p. **??**) : Objects that can evolve in time.

It defines an abstract API for object that can move during the simulation. By moving, we suggest physically or just changing its internal state.

See Also

sif::Environment (p. **??**), **sif::Object** (p. **??**), **sif::StaticObject** (p. **??**), sif::Ressource, **sif::TaskSpot** (p. **??**)

Definition at line 48 of file dynamicObject.hpp.

5.21.2 Constructor & Destructor Documentation

5.21.2.1 template<int Dim, class Type > sif::DynamicObject< Dim, Type >::DynamicObject (Coordonate< Dim, Type > *_coord*)

Constructor.

Parameters

_coord	Coordonates of the resource

Definition at line 33 of file dynamicObject.cpp.

5.21.3 Member Function Documentation

5.21.3.1 template<int Dim, class Type > virtual void sif::DynamicObject< Dim, Type >::update (double *_time*) [pure virtual]

Update dynamic object.

Parameters

_time	Ellapsed time since the last update

Implemented in **sif::Resource**< **Dim, Type, Data** > (p. **??**), and **sif::TaskSpot**< **Dim, Type** > (p. **??**).

5.21.3.2 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const [inherited]

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

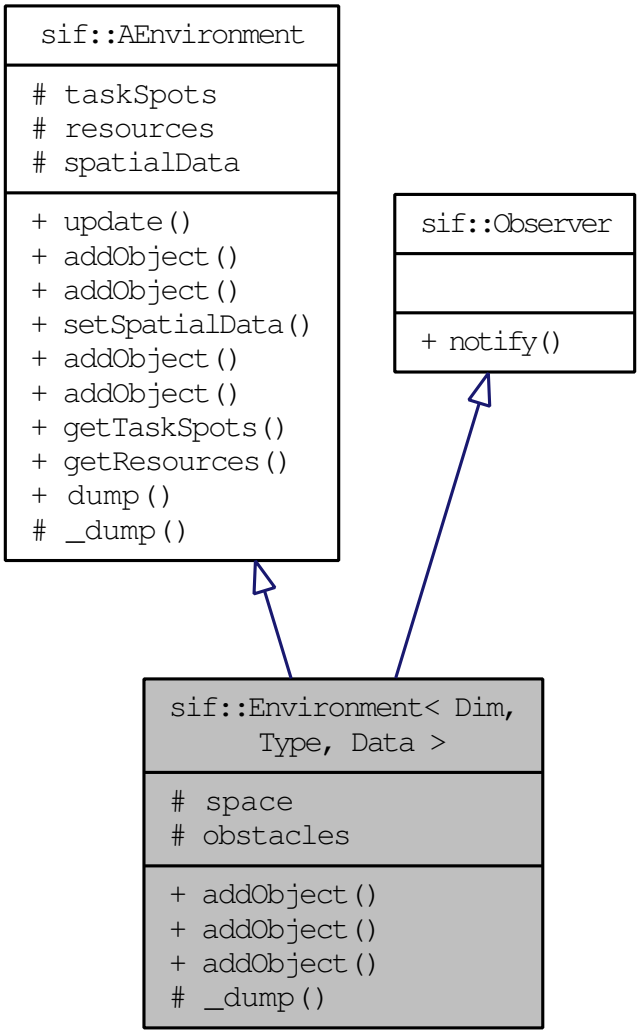
- dynamicObject.hpp
- dynamicObject.cpp

5.22 sif::Environment< Dim, Type, Data > Class Template Reference

Environment (p. **??**) : Modelize the environment of the problem.

`#include <environment.hpp>`

Inheritance diagram for sif::Environment< Dim, Type, Data >:



Public Member Functions

- void **addObject** (std::vector< **AResource** * > &_resources)

  Add resources to the environment.
- void **addObject** (**Obstacle**< Dim, Type > &_obstacle)

  Add obstacle to the environment.
- void **addObject** (std::vector< **ATaskSpot** * > &_taskSpots)

Add taskSpots to the environment.

- void **update** (double _time)

  Update environment by updating all its components.
- void **addObject** (**AResource** &_resource)

  Add resource to the environment.
- void **addObject** (**ATaskSpot** &_taskSpot)

  Add taskSpot to the environment.
- void **setSpatialData** (**SpatialData** &_spatialData)

  setSpatialData
- std::vector< **ATaskSpot** * > & **getTaskSpots** ()

  Get TaskSpots.
- std::vector< **AResource** * > & **getResources** ()

  Get Resources.
- void **dump** ()

  Debug function to show all informations about the class.
- void **notify** ()

  Notify observable.

Protected Member Functions

- void **_dump** ()

  Debug function to show all informations about the class.

Protected Attributes

- **Space**< Dim, Type > **space**

  Space *(p. **??**).*
- std::vector< **Obstacle**< Dim, Type > > **obstacles**

  Obstacle *(p. **??**) of the environment.*
- std::vector< **ATaskSpot** * > **taskSpots**

  TaskSpots of the environment.
- std::vector< **AResource** * > **resources**

  Resources of the environment.
- **SpatialData** * **spatialData**

  SpatialData *(p. **??**).*

5.22.1 Detailed Description

template<int Dim, class Type, class Data>class sif::Environment< Dim, Type, Data >

Environment (p. **??**) : Modelize the environment of the problem.

The environment is a description of the world used for the simulation or the calculation. It contains the different objets that can evolve in a specific space.

See Also

sif::Space (p. **??**), **sif::Object** (p. **??**)

Definition at line 53 of file environment.hpp.

5.22.2 Member Function Documentation

5.22.2.1 template<int Dim, class Type , class Data > void sif::Environment< Dim, Type, Data >::addObject (std::vector< AResource * > & *_resources*)

Add resources to the environment.

Parameters

_resources	Vector of resources

Definition at line 33 of file src/SIF/environment/environment.cpp.

References sif::AEnvironment::addObject().

5.22.2.2 template<int Dim, class Type , class Data > void sif::Environment< Dim, Type, Data >::addObject (Obstacle< Dim, Type > & *_obstacle*)

Add obstacle to the environment.

Parameters

_obstacle	New obstacle

Definition at line 39 of file src/SIF/environment/environment.cpp.

References sif::Object< Dim, Type >::getCoordonates().

5.22.2.3 template<int Dim, class Type , class Data > void sif::Environment< Dim, Type, Data >::addObject (std::vector< ATaskSpot * > & *_taskSpots*)

Add taskSpots to the environment.

Parameters

_taskSpots	Vector of taskSpots

Definition at line 53 of file src/SIF/environment/environment.cpp.

References sif::AEnvironment::addObject().

5.22.2.4 void sif::AEnvironment::update (double *_time*) [inherited]

Update environment by updating all its components.

Parameters

_time	Ellapsed time since the last update

Definition at line 37 of file aEnvironment.cpp.

References sif::AEnvironment::resources.

5.22.2.5 void sif::AEnvironment::addObject (AResource & *_resource*) [inherited]

Add resource to the environment.

Parameters

_resource	New resource

Definition at line 56 of file aEnvironment.cpp.

References sif::AEnvironment::resources.

Referenced by sif::Environment< Dim, Type, Data >::addObject().

5.22.2.6 void sif::AEnvironment::addObject (ATaskSpot & *_taskSpot*) [inherited]

Add taskSpot to the environment.

Parameters

_taskSpot	New taskSpot

Definition at line 69 of file aEnvironment.cpp.

References sif::AEnvironment::taskSpots.

5.22.2.7 void sif::AEnvironment::setSpatialData (SpatialData & *_spatialData*) [inherited]

setSpatialData

Parameters

_spatialData	New spatialData

Definition at line 62 of file aEnvironment.cpp.

References sif::AEnvironment::resources, and sif::AEnvironment::spatialData.

5.22.2.8 std::vector< ATaskSpot * > & sif::AEnvironment::getTaskSpots ( ) [inherited]

Get TaskSpots.

Returns

Vector of TaskSpots

Definition at line 80 of file aEnvironment.cpp.

References sif::AEnvironment::taskSpots.

Referenced by sif::SpatialData::startIndexing().

5.22.2.9 std::vector< AResource * > & sif::AEnvironment::getResources ( ) [inherited]

Get Resources.

Returns

Vector of Resources

Definition at line 85 of file aEnvironment.cpp.

References sif::AEnvironment::resources.

Referenced by sif::SpatialData::startIndexing().

The documentation for this class was generated from the following files:

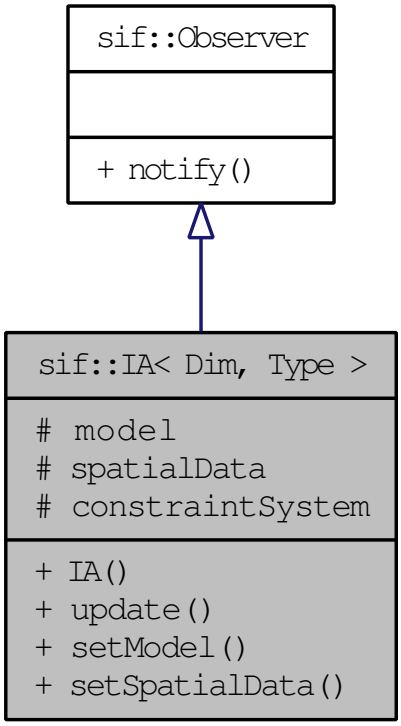
- environment.hpp
- src/SIF/environment/environment.cpp

5.23 sif::IA< Dim, Type > Class Template Reference

IA (p. **??**) :

`#include <ia.hpp>`

Inheritance diagram for sif::IA< Dim, Type >:



Public Member Functions

- **IA** (**Model**< Dim, Type > &_model, **ConstraintSystem** &_constraintSystem)

  Constructor.
- void **update** (double _time)

  *Update **IA** (p. **??**).*
- void **setModel** (**Model**< Dim, Type > &_model)

  Set the model.
- void **setSpatialData** (**SpatialData** &_spatialData)

  Set the spatial data.
- void **notify** ()

  Notify observable.

Protected Attributes

- **Model**< Dim, Type > & **model**

  ***Model** (p. **??**) of the **IA** (p. **??**).*
- **SpatialData** * **spatialData**

SpatialData *(*p. **??***).*

- **ConstraintSystem** & **constraintSystem**

  Constraint *(*p. **??***) system.*

5.23.1 Detailed Description

template<int Dim, class Type>class sif::IA< Dim, Type >

IA (p. **??**) :

IA (p. **??**)

See Also

sif::Observer (p. **??**),

Definition at line 45 of file ia.hpp.

5.23.2 Constructor & Destructor Documentation

5.23.2.1 template<int Dim, class Type > sif::IA< Dim, Type >::IA (Model< Dim, Type > & *_model,* ConstraintSystem & *_constraintSystem*)

Constructor.

Parameters

_model	**Model** (p. **??**) of **IA** (p. **??**)
_constraint-System	**Constraint** (p. **??**) system related to tasks

Definition at line 35 of file ia.cpp.

5.23.3 Member Function Documentation

5.23.3.1 template<int Dim, class Type > void sif::IA< Dim, Type >::update (double *_time*)

Update **IA** (p. **??**).

Parameters

_time	Ellapsed time since the last update

Definition at line 41 of file ia.cpp.

5.23.3.2 template<int Dim, class Type > void sif::IA< Dim, Type >::setModel (Model< Dim, Type > & *_model*)

Set the model.

Parameters

_model	New strategy model

Definition at line 50 of file ia.cpp.

5.23.3.3 template<int Dim, class Type > void sif::IA< Dim, Type >::setSpatialData (SpatialData & *_spatialData*)

Set the spatial data.

Parameters

_spatialData	New spatialData

Definition at line 56 of file ia.cpp.

The documentation for this class was generated from the following files:

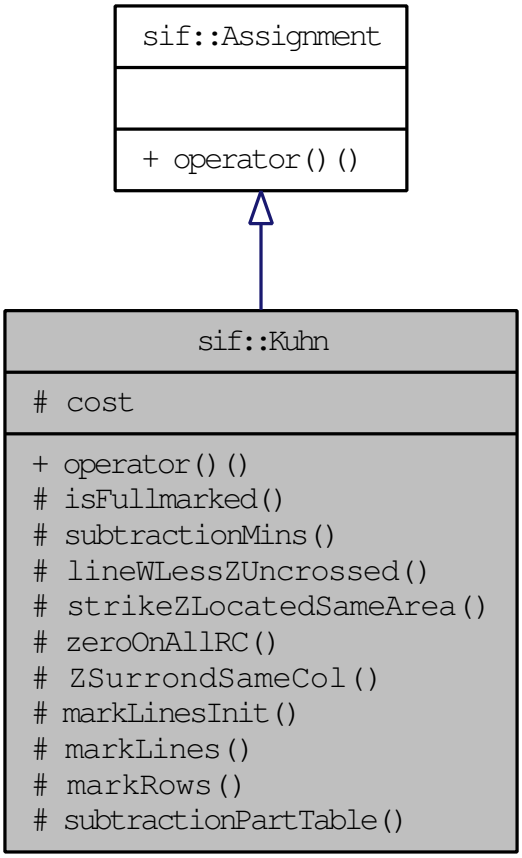
- ia.hpp
- ia.cpp

5.24 sif::Kuhn Class Reference

Kuhn (p. **??**) : Algorithm which solves the assignment problem.

`#include <kuhn.hpp>`

Inheritance diagram for sif::Kuhn:



Public Member Functions

- std::map< **AResource** ∗, **ATaskSpot** ∗ > **operator()** (std::map< std::pair< **AResource** ∗, **ATaskSpot** ∗ >, int > _mymap)

  *Start the **Kuhn** (p. **??**) method.*

Protected Member Functions

- bool **isFullmarked** (std::vector< bool > &mdL, std::vector< bool > &mdR)

  Checks if the matrix is totally marked.
- void **subtractionMins** (typeMat &mat)

  Subtracts minimum on each row & column.
- int **lineWLessZUncrossed** (typeMat &m)

  Looks for the line with less zero with Normal state.
- void **strikeZLocatedSameArea** (typeMat &m, int l, int c)

  Changes state of the zero on line l and row c to Crossed.
- bool **zeroOnAllRC** (typeMat &m)

  Checks if there is one zero surrounded on each line & column.
- bool **ZSurrondSameCol** (typeMat &m, int c)

  Looks if there is already a zero on the row c.
- void **markLinesInit** (typeMat &m, std::vector< bool > &mdL)

  Marked all lines containing no zero surrounded.
- bool **markLines** (typeMat &m, std::vector< bool > &mdL, std::vector< bool > &mdR)

  Marks all rows containing a zero crossed on a marked line.
- bool **markRows** (typeMat &m, std::vector< bool > &mdL, std::vector< bool > &mdR)

  Marks all lines containing a zero surrounded on a marked column.
- void **subtractionPartTable** (typeMat &m, std::vector< bool > &mdL, std::vector< bool > &mdR)

  Subtracts the minimum in the partial table.

Protected Attributes

- std::vector< std::vector< int > > **cost**

  Cost matrix.

5.24.1 Detailed Description

Kuhn (p. **??**) : Algorithm which solves the assignment problem.

Kuhn (p. **??**) is an algorithm wihch solves the asssignment problem. **Kuhn** (p. **??**) is also named the Hungarian method.

See Also

sif::Assignment (p. **??**)

Definition at line 49 of file kuhn.hpp.

5.24.2 Member Function Documentation

5.24.2.1 std::map< AResource *, ATaskSpot * > sif::Kuhn::operator() (std::map< std::pair< AResource *, ATaskSpot * >, int > *_mymap*) [virtual]

Start the **Kuhn** (p. **??**) method.

Parameters

_mymap	A map which associates a pair of resource / taskSpot with a cost

Returns

A map from resource to **TaskSpot** (p. **??**)

Implements **sif::Assignment** (p. **??**).

Definition at line 35 of file kuhn.cpp.

References cost, isFullmarked(), lineWLessZUncrossed(), markLines(), markLinesInit(), markRows(), strikeZLocatedSameArea(), subtractionMins(), subtractionPartTable(), zeroOnAllRC(), and ZSurrondSameCol().

5.24.2.2 bool sif::Kuhn::isFullmarked (std::vector< bool > & *mdL*, std::vector< bool > & *mdR*) [protected]

Checks if the matrix is totally marked.

Parameters

mdL	Vector of boolean which enable to know if a line is marked
mdR	Vector of boolean which enable to know if a row is marked

Returns

true if the 2 vectors are completely true (ie only filled with 1), false otherwise

Definition at line 251 of file kuhn.cpp.

Referenced by operator()().

5.24.2.3 void sif::Kuhn::subtractionMins (typeMat & *mat*) [protected]

Subtracts minimum on each row & column.

Parameters

mat	Cost matrix

Definition at line 267 of file kuhn.cpp.

Referenced by operator()().

5.24.2.4 int sif::Kuhn::lineWLessZUncrossed (typeMat & *m*) [protected]

Looks for the line with less zero with Normal state.

Parameters

mat	Cost matrix

Returns

Index of the line with less zero

Definition at line 301 of file kuhn.cpp.

Referenced by operator()().

5.24.2.5 void sif::Kuhn::strikeZLocatedSameArea (typeMat & *m*, int *l*, int *c*) [protected]

Changes state of the zero on line l and row c to Crossed.

Parameters

mat	Cost matrix
value	of the line to consider
value	of the row to consider

Definition at line 337 of file kuhn.cpp.

Referenced by operator()().

5.24.2.6 bool sif::Kuhn::zeroOnAllRC (typeMat & *m*) [protected]

Checks if there is one zero surrounded on each line & column.

Parameters

mat	Cost matrix

Returns

true if there is a zero on each line & column, false otherwise

Definition at line 350 of file kuhn.cpp.

Referenced by operator()().

5.24.2.7 bool sif::Kuhn::ZSurrondSameCol (typeMat & *m,* int *c*) [protected]

Looks if there is already a zero on the row c.

Parameters

mat	Cost matrix
c	Index of the row to consider

Returns

true if there is already a zero, false otherwise

Definition at line 324 of file kuhn.cpp.

Referenced by operator()().

5.24.2.8 void sif::Kuhn::markLinesInit (typeMat & *m,* std::vector< bool > & *mdL*) [protected]

Marked all lines containing no zero surrounded.

Parameters

mat	Cost matrix
mdL	Vector marked lines

Definition at line 372 of file kuhn.cpp.

Referenced by operator()().

5.24.2.9 bool sif::Kuhn::markLines (typeMat & *m*, std::vector< bool > & *mdL*, std::vector< bool > & *mdR*) [protected]

Marks all rows containing a zero crossed on a marked line.

Parameters

mat	Cost matrix
mdL	Vector marked lines
mdR	Vector marked rows

Returns

true if there have been changes, false otherwise

Definition at line 432 of file kuhn.cpp.

Referenced by operator()().

5.24.2.10 bool sif::Kuhn::markRows (typeMat & *m*, std::vector< bool > & *mdL*, std::vector< bool > & *mdR*) [protected]

Marks all lines containing a zero surrounded on a marked column.

Parameters

mat	Cost matrix
mdL	Vector marked lines
mdR	Vector marked rows

Returns

true if there have been changes, false otherwise

Definition at line 400 of file kuhn.cpp.

Referenced by operator()().

5.24.2.11 void sif::Kuhn::subtractionPartTable (typeMat & *m*, std::vector< bool > & *mdL*, std::vector< bool > & *mdR*) [protected]

Subtracts the minimum in the partial table.

Parameters

mat	Cost matrix
mdL	Vector marked lines
mdR	Vector marked rows

Definition at line 464 of file kuhn.cpp.

Referenced by operator()().

The documentation for this class was generated from the following files:

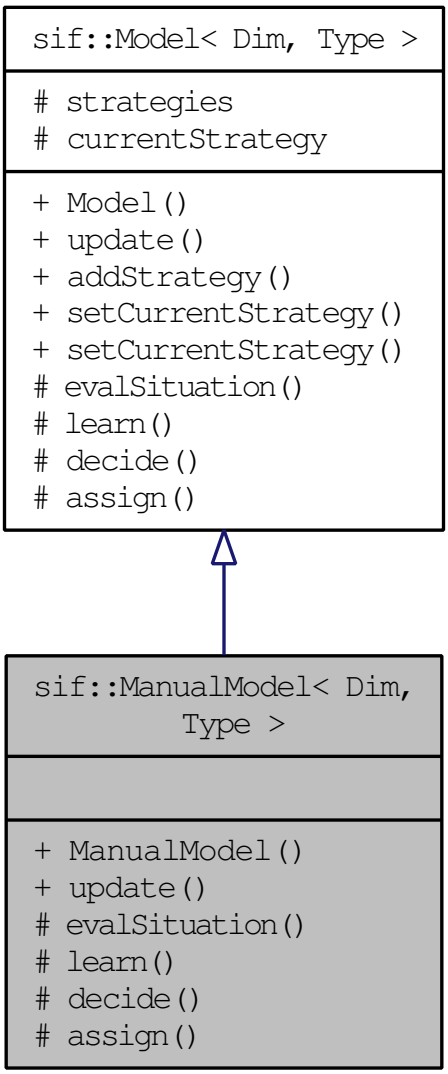
- kuhn.hpp
- kuhn.cpp

5.25 sif::ManualModel< Dim, Type > Class Template Reference

ManualModel (p. **??**) : **Model** (p. **??**) without any decision or learning process.

`#include <manualModel.hpp>`

Inheritance diagram for sif::ManualModel< Dim, Type >:



Public Member Functions

- **ManualModel** (**Strategy**< Dim, Type > &_currentStrategy)

  Constructor.
- virtual void **update** (double _time, **SpatialData** &_spatialData, **ConstraintSystem** &_constraintSystem)

  *Update **Model** (p. **??**).*
- void **addStrategy** (**Strategy**< Dim, Type > &_strategy)

  Add a strategy to the model.
- void **setCurrentStrategy** (unsigned _pos)

  Set current strategy.
- void **setCurrentStrategy** (**Strategy**< Dim, Type > &_strategy)

  Set current strategy, only if the strategy is in the model.

Protected Member Functions

- virtual int **evalSituation** (**SpatialData** &_spatialData, **ConstraintSystem** &_constraintSystem)

  Evaluation of the situation : define how the situation has to be evaluated, according to the current strategy.
- virtual void **learn** ()

  Learning process.
- virtual void **decide** ()

  Decision process : mainly dedicated to change the strategy according to evaluation.
- virtual std::map< **AResource** *, **ATaskSpot** * > **assign** ()

  Assignment *(*p. **??***) process : mainly dedicated to assign through the assignment algorithm of the main strategy.*

Protected Attributes

- std::vector< **Strategy**< Dim, Type > * > **strategies**

  Strategy *(*p. **??***) used by the model.*
- **Strategy**< Dim, Type > & **currentStrategy**

  Current strategy.

5.25.1 Detailed Description

template<int Dim, class Type>class sif::ManualModel< Dim, Type >

ManualModel (p. **??**) : **Model** (p. **??**) without any decision or learning process.

This model is the simpliest model that can be imagined. No relationship is defined between strategies and the current strategy is defined by the user.

See Also

sif::IA (p. **??**), **sif::Strategy** (p. **??**), **sif::Model** (p. **??**)

Definition at line 48 of file manualModel.hpp.

5.25.2 Constructor & Destructor Documentation

5.25.2.1 template<int Dim, class Type > sif::ManualModel< Dim, Type >::ManualModel (Strategy< Dim, Type > & *_currentStrategy*)

Constructor.

Parameters

_currentStrategy	Main strategy

Definition at line 33 of file manualModel.cpp.

5.25.3 Member Function Documentation

5.25.3.1 template<int Dim, class Type > void sif::ManualModel< Dim, Type >::update (double *_time,* SpatialData & *_spatialData,* ConstraintSystem & *_constraintSystem*) [virtual]

Update **Model** (p. **??**).

Parameters

_time	Ellapsed time since the last update
_spatialData	Data
_constraint-System	**Constraint** (p. **??**) system related to tasks

Implements **sif::Model< Dim, Type >** (p. **??**).

Definition at line 40 of file manualModel.cpp.

5.25.3.2 template<int Dim, class Type > int sif::ManualModel< Dim, Type >::evalSituation (SpatialData & *_spatialData*, ConstraintSystem & *_constraintSystem*) [protected], [virtual]

Evaluation of the situation : define how the situation has to be evaluated, according to the current strategy.

Returns

Value of the evaluation

Implements **sif::Model< Dim, Type >** (p. **??**).

Definition at line 50 of file manualModel.cpp.

References sif::Model< Dim, Type >::evalSituation().

5.25.3.3 template<int Dim, class Type > void sif::Model< Dim, Type >::addStrategy (Strategy< Dim, Type > & *_strategy*) [inherited]

Add a strategy to the model.

Parameters

_strategy	**Strategy** (p. **??**) to add to the model

Definition at line 42 of file model.cpp.

Referenced by sif::Model< Dim, Type >::Model().

5.25.3.4 template<int Dim, class Type > void sif::Model< Dim, Type >::setCurrentStrategy (unsigned *_pos*) [inherited]

Set current strategy.

Parameters

_pos	Position of the strategy in the vector

Definition at line 48 of file model.cpp.

5.25.3.5 template<int Dim, class Type > void sif::Model< Dim, Type >::setCurrentStrategy (Strategy< Dim, Type > & *_strategy*) [inherited]

Set current strategy, only if the strategy is in the model.

Parameters

_strategy	**Strategy** (p. **??**) to set as current strategy

Definition at line 55 of file model.cpp.

The documentation for this class was generated from the following files:

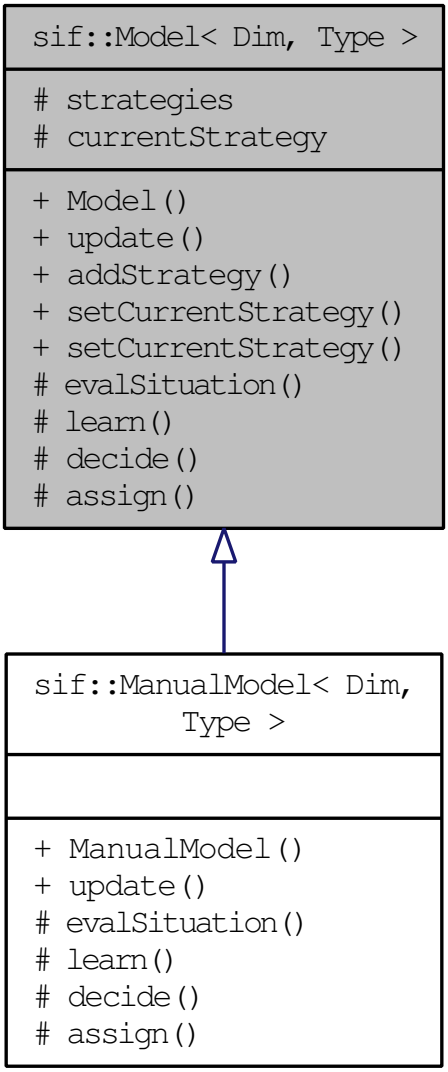
- manualModel.hpp
- manualModel.cpp

5.26 sif::Model< Dim, Type > Class Template Reference

Model (p. **??**) : Defines relationship between strategies and how the observation is done.

`#include <model.hpp>`

Inheritance diagram for sif::Model< Dim, Type >:



Public Member Functions

- **Model** (**Strategy**< Dim, Type > &_currentStrategy)

  Constructor.
- virtual void **update** (double _time, **SpatialData** &_spatialData, **ConstraintSystem** &_constraintSystem)=0

  *Update **Model** (p. **??**).*

- void **addStrategy** (**Strategy**< Dim, Type > &_strategy)

  Add a strategy to the model.
- void **setCurrentStrategy** (unsigned _pos)

  Set current strategy.
- void **setCurrentStrategy** (**Strategy**< Dim, Type > &_strategy)

  Set current strategy, only if the strategy is in the model.

Protected Member Functions

- virtual int **evalSituation** (**SpatialData** &_spatialData, **ConstraintSystem** &_constraintSystem)=0

  Evaluation of the situation : define how the situation has to be evaluated, according to the current strategy.
- virtual void **learn** ()=0

  Learning process.
- virtual void **decide** ()=0

  Decision process : mainly dedicated to change the strategy according to evaluation.
- virtual std::map< **AResource** *, **ATaskSpot** * > **assign** ()=0

  Assignment *(*p. **??***) process : mainly dedicated to assign through the assignment algorithm of the main strategy.*

Protected Attributes

- std::vector< **Strategy**< Dim, Type > * > **strategies**

  Strategy *(*p. **??***) used by the model.*
- **Strategy**< Dim, Type > & **currentStrategy**

  Current strategy.

5.26.1 Detailed Description

template<int Dim, class Type>class sif::Model< Dim, Type >

Model (p. **??**) : Defines relationship between strategies and how the observation is done.

Model (p. **??**) defines the arrangement bewteen strategies and how the observation and all the **IA** (p. **??**) process has to be carried out.

See Also

sif::IA (p. **??**), **sif::Strategy** (p. **??**)

Definition at line 45 of file model.hpp.

5.26.2 Constructor & Destructor Documentation

5.26.2.1 template<int Dim, class Type > sif::Model< Dim, Type >::Model (Strategy< Dim, Type > & *_currentStrategy*)

Constructor.

Parameters

_currentStrategy	Main strategy

Definition at line 35 of file model.cpp.

References sif::Model< Dim, Type >::addStrategy().

5.26.3 Member Function Documentation

5.26.3.1 template<int Dim, class Type> virtual void sif::Model< Dim, Type >::update (double *_time,* SpatialData & *_spatialData,* ConstraintSystem & *_constraintSystem*) [pure virtual]

Update **Model** (p. **??**).

Parameters

_time	Ellapsed time since the last update
_spatialData	Data
_constraint-System	**Constraint** (p. **??**) system related to tasks

Implemented in **sif::ManualModel< Dim, Type >** (p. **??**).

5.26.3.2 template<int Dim, class Type > void sif::Model< Dim, Type >::addStrategy (Strategy< Dim, Type > & *_strategy*)

Add a strategy to the model.

Parameters

_strategy	**Strategy** (p. **??**) to add to the model

Definition at line 42 of file model.cpp.

Referenced by sif::Model< Dim, Type >::Model().

5.26.3.3 template<int Dim, class Type > void sif::Model< Dim, Type >::setCurrentStrategy (unsigned *_pos*)

Set current strategy.

Parameters

_pos	Position of the strategy in the vector

Definition at line 48 of file model.cpp.

5.26.3.4 template<int Dim, class Type > void sif::Model< Dim, Type >::setCurrentStrategy (Strategy< Dim, Type > & *_strategy*)

Set current strategy, only if the strategy is in the model.

Parameters

_strategy	**Strategy** (p. **??**) to set as current strategy

Definition at line 55 of file model.cpp.

5.26.3.5 template<int Dim, class Type> virtual int sif::Model< Dim, Type >::evalSituation (SpatialData & *_spatialData,* ConstraintSystem & *_constraintSystem*) [protected], [pure virtual]

Evaluation of the situation : define how the situation has to be evaluated, according to the current strategy.

Returns

Value of the evaluation

Implemented in **sif::ManualModel< Dim, Type >** (p. **??**).

Referenced by sif::ManualModel< Dim, Type >::evalSituation().

The documentation for this class was generated from the following files:

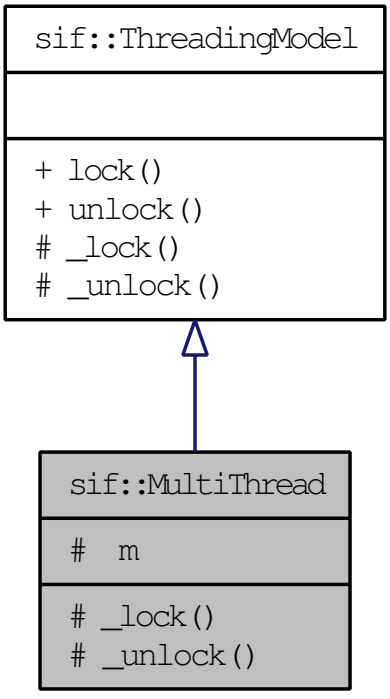
- model.hpp
- model.cpp

5.27 sif::MultiThread Class Reference

MultiThread (p. **??**) : Multithread policy.

`#include <multiThread.hpp>`

Inheritance diagram for sif::MultiThread:



Public Member Functions

- virtual void **lock** () final

  Lock the scope.
- virtual void **unlock** () final

  Unlock the scope.

Protected Member Functions

- virtual void **_lock** ()

  Implemention of the lock method.

- virtual void **_unlock** ()

  Implemention of the unlock method.

Protected Attributes

- std::mutex **m**

  Mutex to lock the scope.

5.27.1 Detailed Description

MultiThread (p. **??**) : Multithread policy.

Provide mecanisms for a thread-safe environnement in order to allow parallel computation.

See Also

sif::SingleThread (p. **??**), **sif::ThreadingModel** (p. **??**)

Definition at line 46 of file multiThread.hpp.

The documentation for this class was generated from the following files:

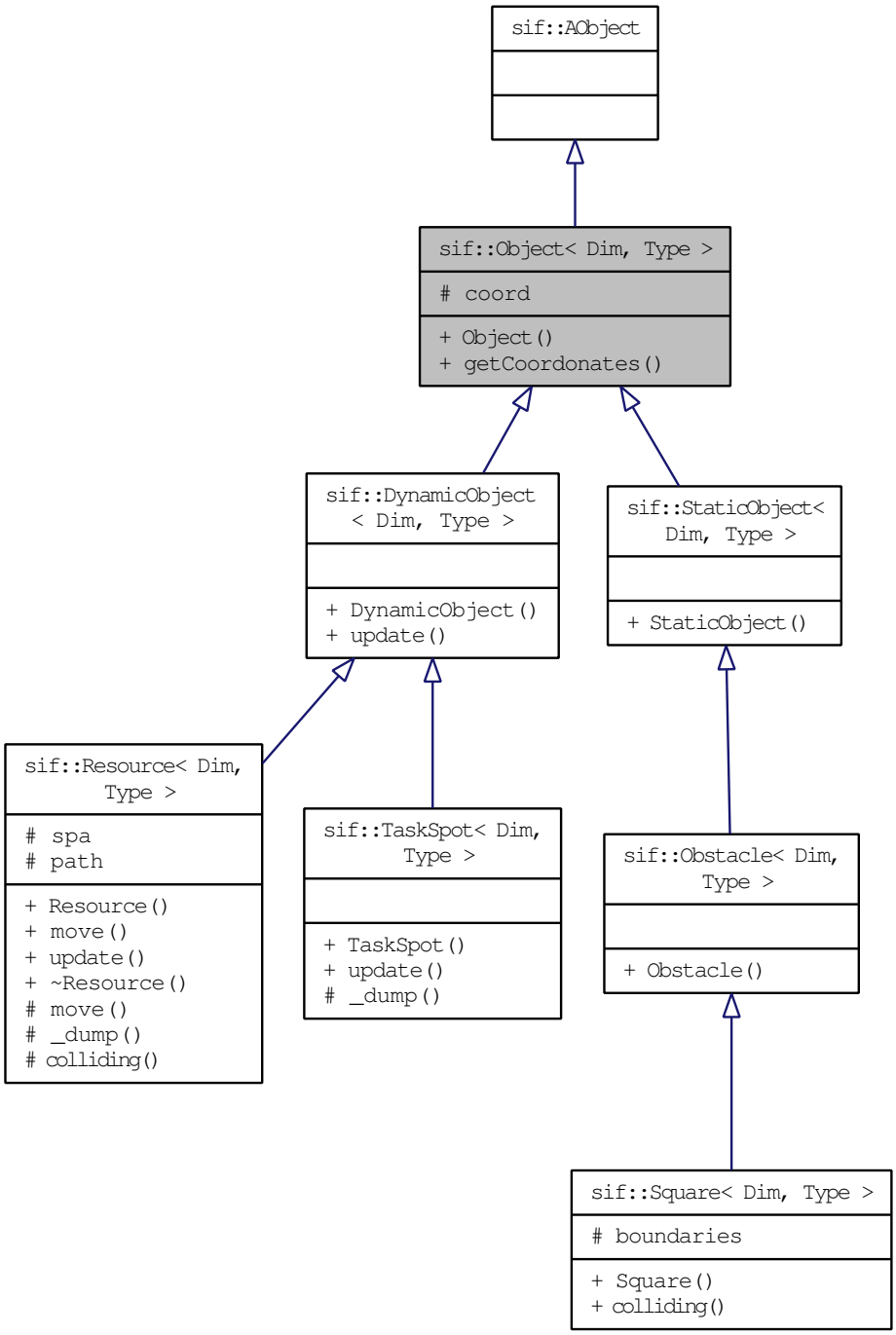
- multiThread.hpp
- multiThread.cpp

5.28 sif::Object< Dim, Type > Class Template Reference

Object (p. **??**) : Abstract class for objects contained in the environment.

`#include <object.hpp>`

Inheritance diagram for sif::Object< Dim, Type >:



Public Member Functions

- **Object** (**Coordonate**< Dim, Type > _coord)

  Constructor.

- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Attributes

- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.28.1 Detailed Description

template<int Dim, class Type>class sif::Object< Dim, Type >

Object (p. **??**) : Abstract class for objects contained in the environment.

Abstract class for objects contained in the environment.

See Also

sif::Environment (p. **??**), **sif::Space** (p. **??**), **sif::DynamicObject** (p. **??**), **sif::StaticObject** (p. **??**)

Definition at line 46 of file object.hpp.

5.28.2 Constructor & Destructor Documentation

5.28.2.1 template<int Dim, class Type > sif::Object< Dim, Type >::Object (Coordonate< Dim, Type > *_coord*)

Constructor.

Parameters

_coord	Coordonates of the resource

Definition at line 32 of file object.cpp.

5.28.3 Member Function Documentation

5.28.3.1 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

- object.hpp
- object.cpp

5.29 sif::Observable< NotifyPolicy > Class Template Reference

Observable (p. **??**) : Notify observer when state change occured.

`#include <observable.hpp>`

Inherits NotifyPolicy.

Public Member Functions

- void **addObserver** (**Observer** &_obs)

  Add observer.
- void **removeObserver** (**Observer** &_obs)

  Remove an observer.

Protected Member Functions

- virtual void **_setChange** ()=0

  Implemention of the setChange method.

Protected Attributes

- std::vector< **Observer** * > **observers**

  Observers list.

5.29.1 Detailed Description

template<class NotifyPolicy>class sif::Observable< NotifyPolicy >

Observable (p. **??**) : Notify observer when state change occured.

The **Observable** (p. **??**) part of the Design Pattern **Observer** (p. **??**). The notifying policy determines the behavior of the observable concerning the way of notifying the observer.

See Also

sif::Observer (p. **??**), **sif::ObservablePolicy** (p. **??**), **sif::ActivePolicy** (p. **??**), **sif::PassivePolicy** (p. **??**)

Definition at line 45 of file observable.hpp.

5.29.2 Member Function Documentation

5.29.2.1 template<class NotifyPolicy > void sif::Observable< NotifyPolicy >::addObserver (Observer & *_obs*)

Add observer.

Parameters

_obs	New observer

Definition at line 35 of file observable.cpp.

5.29.2.2 template<class NotifyPolicy > void sif::Observable< NotifyPolicy >::removeObserver (Observer & *_obs*)

Remove an observer.

Parameters

_obs	**Observer** (p. **??**) to remove

Definition at line 41 of file observable.cpp.

The documentation for this class was generated from the following files:

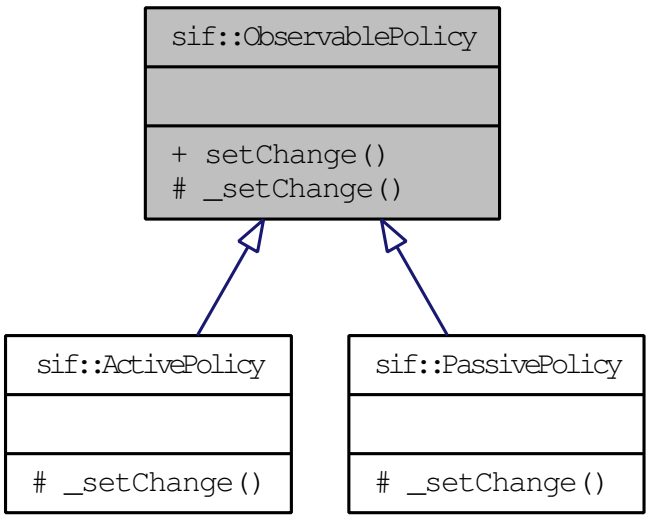
- observable.hpp
- observable.cpp

5.30 sif::ObservablePolicy Class Reference

ObservablePolicy (p. **??**) : Abstract class to determine observable behavior.

`#include <observablePolicy.hpp>`

Inheritance diagram for sif::ObservablePolicy:



Public Member Functions

- virtual void **setChange** () final

  Set change : the meaning of this function depends on the policy.

Protected Member Functions

- virtual void **_setChange** ()=0

  Implementation of setChange method.

5.30.1 Detailed Description

ObservablePolicy (p. **??**) : Abstract class to determine observable behavior.

Define different observable behaviors. The implementation is based on NVI idiom (C++ implementation of the Design Pattern Template Method).

See Also

sif::Observer (p. **??**), **sif::ActivePolicy** (p. **??**), **sif::PassivePolicy** (p. **??**)

Definition at line 42 of file observablePolicy.hpp.

The documentation for this class was generated from the following files:

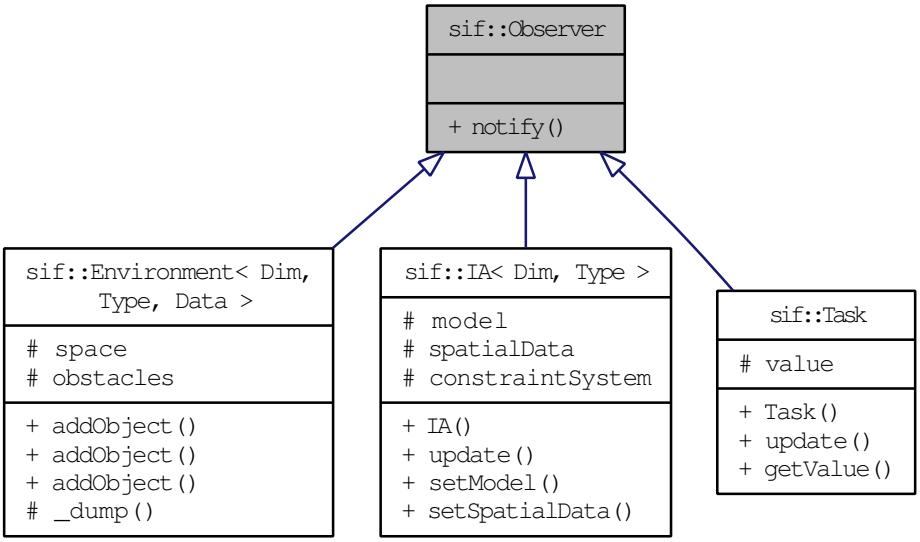
- observablePolicy.hpp
- observablePolicy.cpp

5.31 sif::Observer Class Reference

Observer (p. **??**) : Get notification from observable.

`#include <observer.hpp>`

Inheritance diagram for sif::Observer:



Public Member Functions

- void **notify** ()

  Notify observable.

5.31.1 Detailed Description

Observer (p. **??**) : Get notification from observable.

The **Observer** (p. **??**) part of the Design Pattern **Observer** (p. **??**). An observer can be notified directly by its observers or have to check them, depending on the policy of the observables.

See Also

sif::Observable (p. **??**), **sif::ObservablePolicy** (p. **??**)

Definition at line 43 of file observer.hpp.

The documentation for this class was generated from the following files:

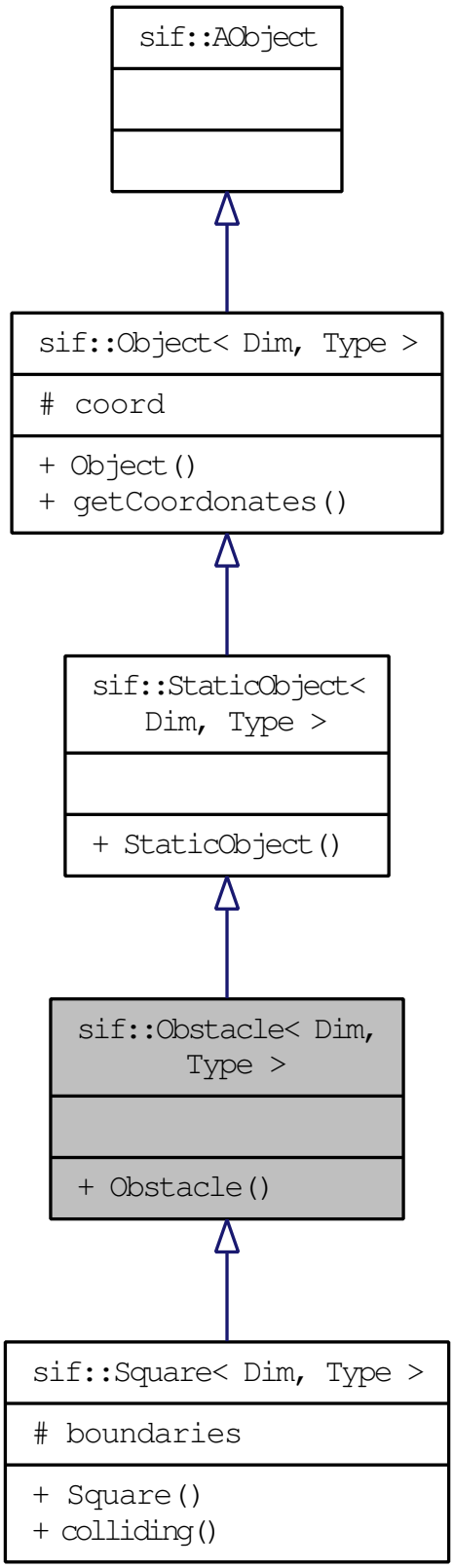
- observer.hpp
- observer.cpp

5.32 sif::Obstacle< Dim, Type > Class Template Reference

Obstacle (p. **??**) : **Obstacle** (p. **??**) in the environment.

`#include <obstacle.hpp>`

Inheritance diagram for sif::Obstacle< Dim, Type >:



Public Member Functions

- **Obstacle** (**Coordonate**< Dim, Type > _coord)

Constructor.

- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Attributes

- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.32.1 Detailed Description

template<**int Dim, class Type**>**class sif::Obstacle**< **Dim, Type** >

Obstacle (p. **??**) : **Obstacle** (p. **??**) in the environment.

It defines obstacle that cannot be overlapped by any objet.

See Also

sif::Environment (p. **??**), **sif::Object** (p. **??**), **sif::DynamicObject** (p. **??**), **sif::Square** (p. **??**)

Definition at line 45 of file obstacle.hpp.

5.32.2 Constructor & Destructor Documentation

5.32.2.1 template<int Dim, class Type > sif::Obstacle< Dim, Type >::Obstacle (Coordonate< Dim, Type > _coord)

Constructor.

Parameters

_coord	Coordonates of the resource

Definition at line 33 of file obstacle.cpp.

5.32.3 Member Function Documentation

5.32.3.1 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const [inherited]

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

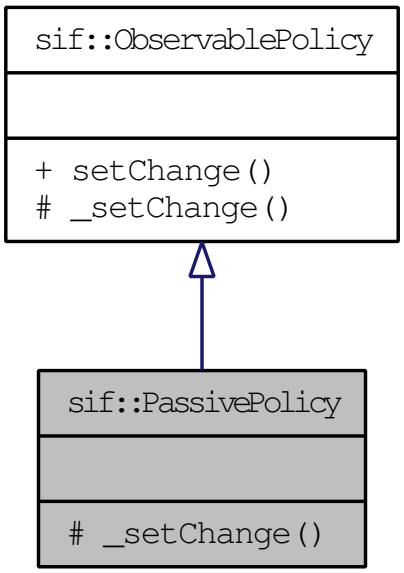
- obstacle.hpp
- obstacle.cpp

5.33 sif::PassivePolicy Class Reference

PassivePolicy (p. **??**) : Change the internal state of the observable.

`#include <passivePolicy.hpp>`

Inheritance diagram for sif::PassivePolicy:



Public Member Functions

- virtual void **setChange** () final

  Set change : the meaning of this function depends on the policy.

Protected Member Functions

- virtual void **_setChange** ()

  Implemention of the setChange method.

5.33.1 Detailed Description

PassivePolicy (p. **??**) : Change the internal state of the observable.

This policy will change the internal state of the observable. It is the responsability of the observer to check the state in order to know if the observable has changed since the last observation.

See Also

sif::Observer (p. **??**), **sif::ObservablePolicy** (p. **??**), **sif::ActivePolicy** (p. **??**)

Definition at line 44 of file passivePolicy.hpp.

The documentation for this class was generated from the following files:

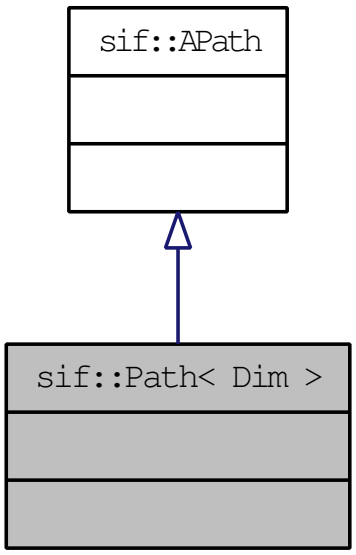
- passivePolicy.hpp
- passivePolicy.cpp

5.34 sif::Path< Dim > Class Template Reference

Path (p. **??**) : **Path** (p. **??**) data structure.

`#include <path.hpp>`

Inheritance diagram for sif::Path< Dim >:



Public Attributes

- T **elements**

  STL member.

5.34.1 Detailed Description

template<int Dim>class sif::Path< Dim >

Path (p. **??**) : **Path** (p. **??**) data structure.

A path is list of data elements used for resources in order they know how to reach an aim.

See Also

sif::Tree (p. **??**)

Definition at line 48 of file path.hpp.

The documentation for this class was generated from the following file:

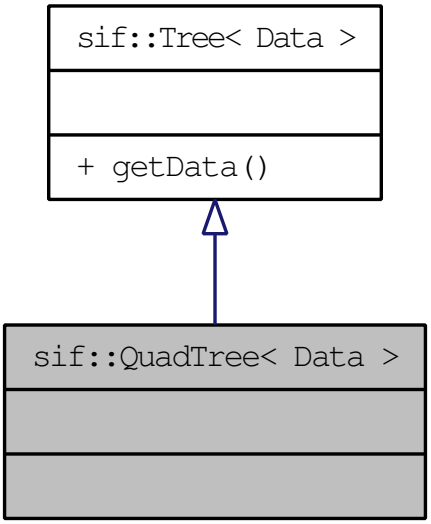
- path.hpp

5.35 sif::QuadTree< Data > Class Template Reference

QuadTree (p. **??**) : **Tree** (p. **??**) data structure.

`#include <quadTree.hpp>`

Inheritance diagram for sif::QuadTree< Data >:



Public Member Functions

- virtual std::vector< Data * > **getData** ()=0
  Return a vector of all objects of the tree.

5.35.1 Detailed Description

template<class Data>class sif::QuadTree< Data >

QuadTree (p. **??**) : **Tree** (p. **??**) data structure.

QuadTree (p. **??**) is a tree data structure in which each internal node has exactly four children, used to partition a space (generally a two-dimensional one).

See Also

sif::Tree (p. **??**)

Definition at line 45 of file quadTree.hpp.

5.35.2 Member Function Documentation

5.35.2.1 template<class Data> virtual std::vector<Data*> sif::Tree< Data >::getData ( ) [pure virtual], [inherited]

Return a vector of all objects of the tree.

Returns

Vector of objets

Implemented in **sif::SimpleIndex< Data >** (p. **??**), **sif::SimpleIndex< sif::ATaskSpot >** (p. **??**), and **sif::SimpleIndex< sif::AResource >** (p. **??**).

The documentation for this class was generated from the following file:

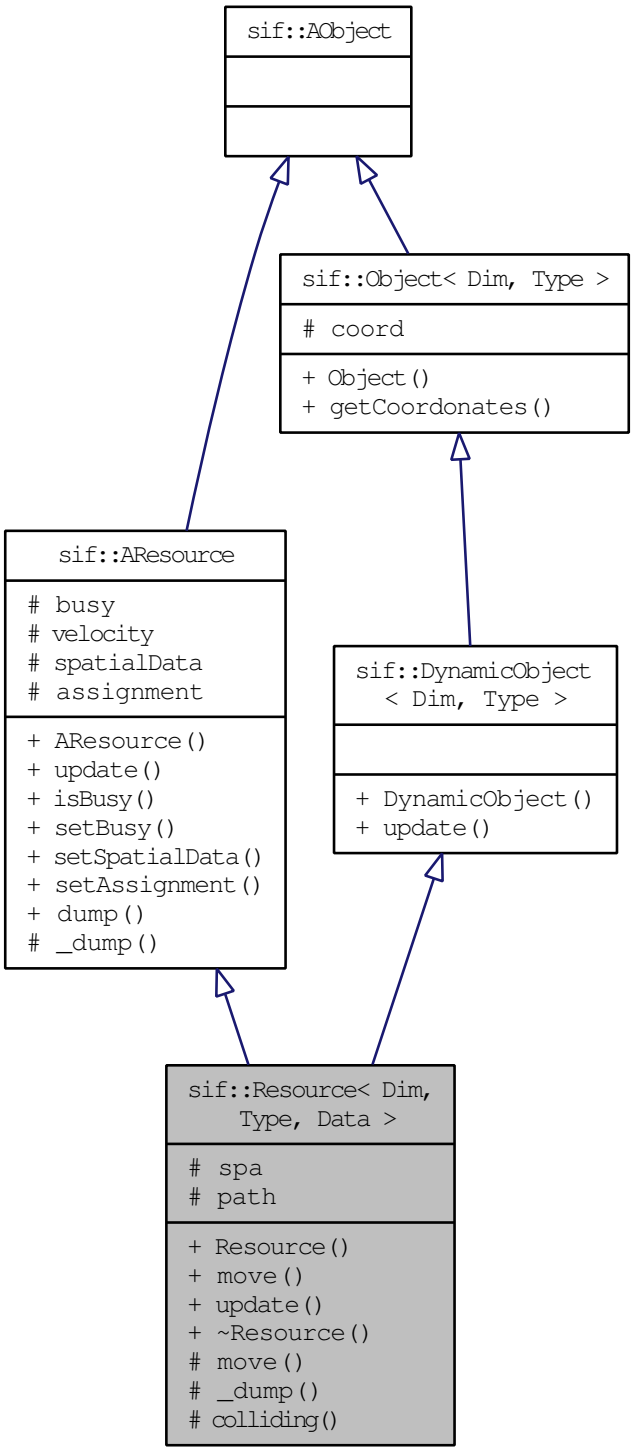
- quadTree.hpp

5.36 sif::Resource< Dim, Type, Data > Class Template Reference

Ressource : Ressource that can be affected to a task.

`#include <resource.hpp>`

Inheritance diagram for sif::Resource< Dim, Type, Data >:



Public Member Functions

- **Resource** (**Coordonate**< Dim, Type > _coord, double _velocity, bool _busy, **ShortestPath**< Dim, Type > _spa)

  Constructor.
- void **move** (**Direction**< Dim > _dir, double _time)

  Move the resource in specified direction.
- virtual void **update** (double _time)

  Update dynamic object.
- ∼**Resource** ()

  Destructor.
- bool **isBusy** () const

  Check if the resource is busy (it cannot be assigned)
- void **setBusy** (bool _status)

  Set the status of the resource.
- void **setSpatialData** (**SpatialData** &_spatialData)

  Set spatialData.
- void **setAssignment** (**ATaskSpot** &_assignment)

  Set a new assignement if it is possible.
- void **dump** ()

  Debug function to show all informations about the class.
- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Member Functions

- void **move** (double _time)

  Move the resource along its path.
- void **_dump** ()

  Implementation of dump function.
- bool **colliding** (**Direction**< Dim > _dir)

  Check the collision according to the direction of the movement.

Protected Attributes

- **ShortestPath**< Dim, Type > **spa**

  Shortest path algorithm.
- **Path**< Dim > ∗ **path**

  Path *(*p. **??***) to follow to reach the assigned taskSpot.*
- bool **busy**

  Status of the resource.
- double **velocity**

  Velocity of the ressource.
- **SpatialData** ∗ **spatialData**

  SpatialData *(*p. **??***).*
- **ATaskSpot** ∗ **assignment**

  Assignment *(*p. **??***).*
- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.36.1 Detailed Description

template<int Dim, class Type, class Data>class sif::Resource< Dim, Type, Data >

Ressource : Ressource that can be affected to a task.

A ressource is an object that can be affected to a task. It modelizes a point in space that can move. Thanks to a shortest path algorithm it is independant when it know the **Task** (p. **??**) Spot he has to join.

See Also

sif::Environment (p. **??**), **sif::Object** (p. **??**), **sif::DynamicObject** (p. **??**), **sif::TaskSpot** (p. **??**)

Definition at line 53 of file resource.hpp.

5.36.2 Constructor & Destructor Documentation

5.36.2.1 template<int Dim, class Type , class Data > sif::Resource< Dim, Type, Data >::Resource (Coordonate< Dim, Type > *_coord,* double *_velocity,* bool *_busy,* ShortestPath< Dim, Type > *_spa*)

Constructor.

Parameters

_coord	Initial coordonates
_velocity	Velocity of the resource
_busy	Status of the resource
_spa	Shortest **Path** (p. **??**) Algorithm

Definition at line 34 of file src/SIF/environment/resource.cpp.

5.36.3 Member Function Documentation

5.36.3.1 template<int Dim, class Type , class Data > void sif::Resource< Dim, Type, Data >::move (Direction< Dim > *_dir,* double *_time*)

Move the resource in specified direction.

Parameters

_dir	**Direction** (p. **??**)
_time	Ellapsed time since the last move

Definition at line 42 of file src/SIF/environment/resource.cpp.

References sif::Direction< Dim >::getValue().

5.36.3.2 template<int Dim, class Type , class Data > void sif::Resource< Dim, Type, Data >::update (double *_time*) [virtual]

Update dynamic object.

Parameters

_time	Ellapsed time since the last update

Implements **sif::AResource** (p. **??**).

Definition at line 57 of file src/SIF/environment/resource.cpp.

5.36.3.3 template<int Dim, class Type , class Data > void sif::Resource< Dim, Type, Data >::move (double *_time*) [protected]

Move the resource along its path.

Parameters

_time	Ellapsed time since the last move

Definition at line 88 of file src/SIF/environment/resource.cpp.

References sif::Object< Dim, Type >::getCoordonates().

5.36.3.4 template<int Dim, class Type, class Data> bool sif::Resource< Dim, Type, Data >::colliding (Direction< Dim > *_dir*) [protected]

Check the collision according to the direction of the movement.

Parameters

_dir	**Direction** (p. **??**)

Returns

true is colliding, false otherwise

5.36.3.5 bool sif::AResource::isBusy ( ) const [inherited]

Check if the resource is busy (it cannot be assigned)

Returns

boolean

Definition at line 41 of file aResource.cpp.

References sif::AResource::busy.

5.36.3.6 void sif::AResource::setBusy (bool *_status*) [inherited]

Set the status of the resource.

Parameters

_status	Bool

Definition at line 46 of file aResource.cpp.

References sif::AResource::busy.

Referenced by sif::AResource::setAssignment().

5.36.3.7 void sif::AResource::setSpatialData (SpatialData & *_spatialData*) [inherited]

Set spatialData.

Parameters

_spatialData	New **SpatialData** (p. **??**)

Definition at line 51 of file aResource.cpp.

References sif::AResource::spatialData.

5.36.3.8 void sif::AResource::setAssignment (ATaskSpot & *_assignment*) [inherited]

Set a new assignement if it is possible.

Parameters

_assignment	New assignment

Definition at line 56 of file aResource.cpp.

References sif::AResource::assignment, and sif::AResource::setBusy().

5.36.3.9 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const [inherited]

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

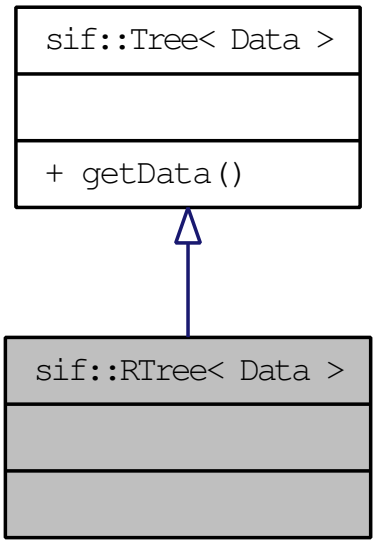
- resource.hpp
- src/SIF/environment/resource.cpp

5.37 sif::RTree< Data > Class Template Reference

RTree (p. **??**) : **Tree** (p. **??**) data structure for indexing multi-dimensional information.

`#include <rTree.hpp>`

Inheritance diagram for sif::RTree< Data >:



Public Member Functions

- virtual std::vector< Data * > **getData** ()=0
  Return a vector of all objects of the tree.

5.37.1 Detailed Description

template<class Data>class sif::RTree< Data >

RTree (p. **??**) : **Tree** (p. **??**) data structure for indexing multi-dimensional information.

RTree (p. **??**) is a tree data structure for spatial access methods, ie, for indexing multi-dimensional information.

See Also

sif::Tree (p. **??**)

Definition at line 45 of file rTree.hpp.

5.37.2 Member Function Documentation

5.37.2.1 template<class Data> virtual std::vector<Data*> sif::Tree< Data >::getData ( ) [pure virtual], [inherited]

Return a vector of all objects of the tree.

Returns

Vector of objets

Implemented in **sif::SimpleIndex< Data >** (p. **??**), **sif::SimpleIndex< sif::ATaskSpot >** (p. **??**), and **sif::SimpleIndex< sif::AResource >** (p. **??**).

The documentation for this class was generated from the following file:

- rTree.hpp

5.38 sif::ShortestPathFactory Class Reference

ShortestPathFactory (p. **??**) :

`#include <shortestPathFactory.hpp>`

Static Public Member Functions

- template<int Dim, class Type >
  static **AStar** && **AStarInstance** (**TaskSpot**< Dim, Type > &_taskSpot)

  AStar *(*p. **??***) instanciation : tranform a* ***TaskSpot*** *(*p. **??***) into coordonates.*

5.38.1 Detailed Description

ShortestPathFactory (p. **??**) :

See Also

sif::ShortestPath (p. **??**)

Definition at line 45 of file shortestPathFactory.hpp.

5.38.2 Member Function Documentation

5.38.2.1 template<int Dim, class Type > static AStar&& sif::ShortestPathFactory::AStarInstance (TaskSpot< Dim, Type > & *_taskSpot*) [static]

AStar (p. **??**) instanciation : tranform a **TaskSpot** (p. **??**) into coordonates.

Returns

AStar (p. **??**) instance (right-value)

The documentation for this class was generated from the following file:

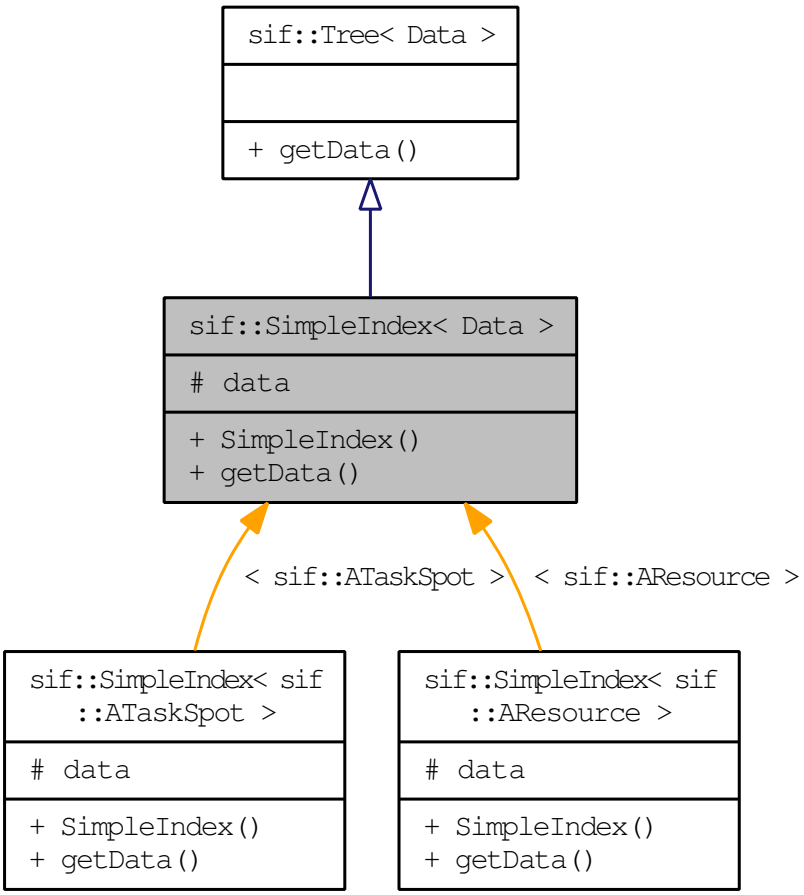
- shortestPathFactory.hpp

5.39 sif::SimpleIndex< Data > Class Template Reference

Simplexe : Stock information in vector.

`#include <simpleIndex.hpp>`

Inheritance diagram for sif::SimpleIndex< Data >:



Public Member Functions

- **SimpleIndex** (std::vector< Data * > &_data)

  Constructor.
- virtual std::vector< Data * > **getData** ()

  Get data - TODO : Get data with a predicate.

Protected Attributes

- std::vector< Data * > & **data**

  Indexed data.

5.39.1 Detailed Description

template<class Data>class sif::SimpleIndex< Data >

Simplexe : Stock information in vector.

Simpliest indexing data ever. :)

See Also

sif::Tree (p. **??**)

Definition at line 48 of file simpleIndex.hpp.

5.39.2 Constructor & Destructor Documentation

5.39.2.1 template<class Data> sif::SimpleIndex< Data >::SimpleIndex (std::vector< Data * > & *_data*)

Constructor.

Parameters

_data	Vector of data to index

Definition at line 33 of file simpleIndex.cpp.

5.39.3 Member Function Documentation

5.39.3.1 template<class Data > std::vector< Data * > sif::SimpleIndex< Data >::getData ( ) [virtual]

Get data - TODO : Get data with a predicate.

Returns

Vector of all data

Implements **sif::Tree**< **Data** > (p. **??**).

Definition at line 38 of file simpleIndex.cpp.

The documentation for this class was generated from the following files:

- simpleIndex.hpp
- simpleIndex.cpp

5.40 sif::SimpleSP Class Reference

SimplePath.

`#include <simpleSP.hpp>`

Public Member Functions

- **SimpleSP** ()=default

  Default constructor.

- template<int Dim, class Type >
  Path< Dim > * **operator()** (const **Coordonate**< Dim, Type > &_from, const **Coordonate**< Dim, Type > &_to)

  Start the shortest path computation.

5.40.1 Detailed Description

SimplePath.

See Also

sif::ShortestPath (p. **??**)

Definition at line 40 of file simpleSP.hpp.

5.40.2 Member Function Documentation

5.40.2.1 template<int Dim, class Type > Path< Dim > * sif::SimpleSP::operator() (const Coordonate< Dim, Type > & *_from,* const Coordonate< Dim, Type > & *_to*)

Start the shortest path computation.

Parameters

_from	Origin
_to	Goal

Returns

A path

Definition at line 40 of file simpleSP.cpp.

The documentation for this class was generated from the following files:

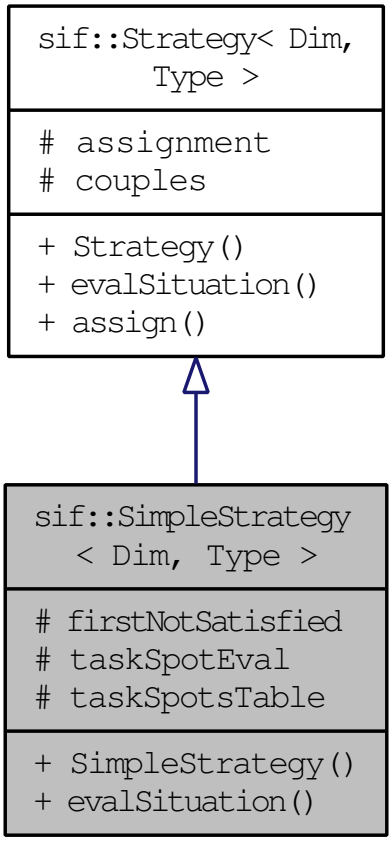
- simpleSP.hpp
- simpleSP.cpp

5.41 sif::SimpleStrategy< Dim, Type > Class Template Reference

SimpleStrategy (p. **??**) : Defines a simple strategy.

`#include <simpleStrategy.hpp>`

Inheritance diagram for sif::SimpleStrategy< Dim, Type >:



Public Member Functions

- **SimpleStrategy** (**Assignment** &_assignment)

  Constructor.
- virtual int **evalSituation** (**SpatialData** &_spatialData, **ConstraintSystem** &_constraintSystem)

  Evaluation of the situation.
- virtual std::map< **AResource** *, **ATaskSpot** * > **assign** ()

  Perform the assignment of resources.

Protected Attributes

- **Constraint** * **firstNotSatisfied**

  First constraint not satisfied.
- **TaskSpotEval taskSpotEval**

  Eval function for taskSpot.
- **EvalTable**< **ATaskSpot** > **taskSpotsTable**

  Evaluation of taskspots.
- **Assignment** & **assignment**

  Assignment *(*p. **??***) algorithm.*
- std::map< std::pair< **AResource** *, **ATaskSpot** * >, int > **couples**

  Couples to be assigned.

5.41.1 Detailed Description

template<int Dim, class Type>class sif::SimpleStrategy< Dim, Type >

SimpleStrategy (p. **??**) : Defines a simple strategy.

See Also

sif::Model (p. **??**),

Definition at line 46 of file simpleStrategy.hpp.

5.41.2 Constructor & Destructor Documentation

5.41.2.1 template<int Dim, class Type > sif::SimpleStrategy< Dim, Type >::SimpleStrategy (Assignment & *_assignment*)

Constructor.

Parameters

_assignment	Assigment algorithm

Definition at line 41 of file simpleStrategy.cpp.

References sif::SimpleStrategy< Dim, Type >::firstNotSatisfied, sif::Constraint::getTask(), and sif::SimpleStrategy< Dim, Type >::taskSpotEval.

5.41.3 Member Function Documentation

5.41.3.1 template<int Dim, class Type > int sif::SimpleStrategy< Dim, Type >::evalSituation (SpatialData & *_spatialData,* ConstraintSystem & *_constraintSystem*) [virtual]

Evaluation of the situation.

Parameters

_spatialData	**SpatialData** (p. **??**) information
_constraint-System	**Constraint** (p. **??**) system

Returns

Value of the situation

Implements **sif::Strategy< Dim, Type >** (p. **??**).

Definition at line 55 of file simpleStrategy.cpp.

References sif::EvalLoop(), sif::SpatialData::getResources(), and sif::SpatialData::getTaskSpots().

5.41.3.2 template<int Dim, class Type > std::map< AResource *, ATaskSpot * > sif::Strategy< Dim, Type >::assign ( ) [virtual], [inherited]

Perform the assignment of resources.

Returns

assigment information

Definition at line 38 of file strategy.cpp.

The documentation for this class was generated from the following files:

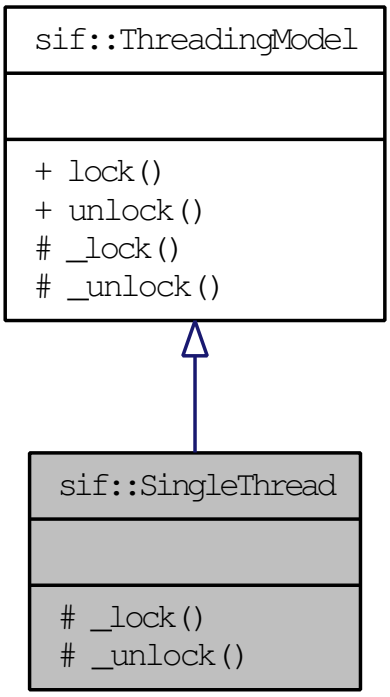
- simpleStrategy.hpp
- simpleStrategy.cpp

5.42 sif::SingleThread Class Reference

SingleThread (p. **??**) : Singlethread policy.

`#include <singleThread.hpp>`

Inheritance diagram for sif::SingleThread:



Public Member Functions

- virtual void **lock** () final
  Lock the scope.
- virtual void **unlock** () final
  Unlock the scope.

Protected Member Functions

- virtual void **_lock** ()=0
  Implemention of the lock method.
- virtual void **_unlock** ()=0
  Implemention of the unlock method.

5.42.1 Detailed Description

SingleThread (p. **??**) : Singlethread policy.

Dummy policy in case or the user does not want to use SIF in a multithread environnement. This permits to avoid overhead due to parallel communications.

See Also

sif::MultiThread (p. **??**), **sif::ThreadingModel** (p. **??**)

Definition at line 44 of file singleThread.hpp.

The documentation for this class was generated from the following file:

- singleThread.hpp

5.43 sif::Space< Dim, Type > Class Template Reference

Space (p. **??**) : Defines the space.

`#include <space.hpp>`

Public Member Functions

- **Space** ()
  Constructor.
- void **setBoundaries** (unsigned _pos, Type _left, Type _right)
  Set boundaries for a specific dimension.
- std::pair< Type, Type > **getBoundaries** (unsigned _pos) const
  Get boundaries for a specific dimension.
- bool **inSpace** (**Coordonate**< Dim, Type > _coord) const
  Check if a point is in the space.

5.43.1 Detailed Description

template<int Dim, class Type>class sif::Space< Dim, Type >

Space (p. **??**) : Defines the space.

It defines the space by its coordonates system and its boundaries.

See Also

sif::Coordonate (p. **??**), **sif::Environment** (p. **??**)

Definition at line 48 of file space.hpp.

5.43.2 Member Function Documentation

5.43.2.1 template<int Dim, class Type > void sif::Space< Dim, Type >::setBoundaries (unsigned *_pos*, Type *_left*, Type *_right*)

Set boundaries for a specific dimension.

Parameters

_pos	Dimension index
_left	Left value
_right	Right value

Definition at line 39 of file space.cpp.

5.43.2.2 template<int Dim, class Type > std::pair< Type, Type > sif::Space< Dim, Type >::getBoundaries (unsigned _pos) const

Get boundaries for a specific dimension.

Parameters

_pos	

Returns

left and right values in a std::pair

Definition at line 46 of file space.cpp.

5.43.2.3 template<int Dim, class Type > bool sif::Space< Dim, Type >::inSpace (Coordonate< Dim, Type > _coord) const

Check if a point is in the space.

Parameters

_coord	**Coordonate** (p. **??**) of the point

Returns

boolean

Definition at line 52 of file space.cpp.

The documentation for this class was generated from the following files:

- space.hpp
- space.cpp

5.44 sif::SpatialData Class Reference

SpatialData (p. **??**) : Contains the index of the objets and the partition of the space.

`#include <spatialData.hpp>`

Public Member Functions

- **SpatialData** (**AEnvironment** *_env)

  Constructor.
- void **startPartitioning** ()

  Start the partition of the environment.

- void **startIndexing** ()

  Start the indexation of the environment.
- **Tree**< **ATaskSpot** > & **getTaskSpots** () const

  Get TaskSpots.
- **Tree**< **AResource** > & **getResources** () const

  Get Resources.
- ~**SpatialData** ()

  Destructor.

Protected Attributes

- **SimpleIndex**< **ATaskSpot** > * **taskSpots**

  Indexed TaskSpots.
- **SimpleIndex**< **AResource** > * **resources**

  *Indexed **Resource** (p. **??**).*
- **AEnvironment** * **env**

  ***Environment** (p. **??**) pointer in the case we would like to launch an algorithm once again.*

5.44.1 Detailed Description

SpatialData (p. **??**) : Contains the index of the objets and the partition of the space.

This is the object that partition and index the environment. All objects that need this information need to use its interface. Final user should not see this class.

Definition at line 51 of file spatialData.hpp.

5.44.2 Constructor & Destructor Documentation

5.44.2.1 sif::SpatialData::SpatialData (AEnvironment * _env)

Constructor.

Parameters

_env	**Environment** (p. **??**) to work on

Definition at line 34 of file spatialData.cpp.

5.44.3 Member Function Documentation

5.44.3.1 Tree< ATaskSpot > & sif::SpatialData::getTaskSpots ( ) const

Get TaskSpots.

Returns

Tree (p. **??**) that contains taskspots

Definition at line 59 of file spatialData.cpp.

References taskSpots.

Referenced by sif::SimpleStrategy< Dim, Type >::evalSituation().

5.44.3.2 Tree< AResource > & sif::SpatialData::getResources ( ) const

Get Resources.

Returns

Tree (p. **??**) that contains Resources

Definition at line 64 of file spatialData.cpp.

References resources.

Referenced by sif::SimpleStrategy< Dim, Type >::evalSituation().

The documentation for this class was generated from the following files:

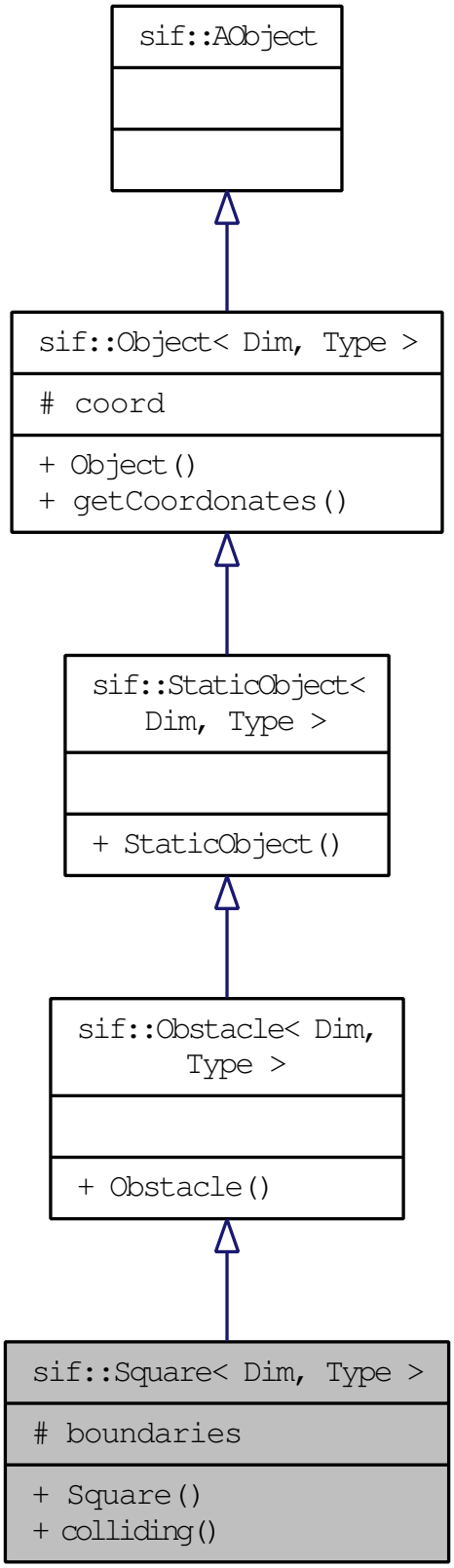
- spatialData.hpp
- spatialData.cpp

5.45 sif::Square< Dim, Type > Class Template Reference

Square (p. **??**) : A square shaped obstacle.

`#include <square.hpp>`

Inheritance diagram for sif::Square< Dim, Type >:



Public Member Functions

- **Square** (**Coordonate**< Dim, Type > _coord, std::vector< std::pair< Type, Type >> _boundaries)

Constructor.

- virtual bool **colliding** (**Coordonate**< Dim, Type > _coord)

  Colliding.

- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Attributes

- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.45.1 Detailed Description

template<int Dim, class Type>class sif::Square< Dim, Type >

Square (p. **??**) : A square shaped obstacle.

It defines a square shaped obstacle.

See Also

sif::Environment (p. **??**), **sif::Object** (p. **??**), **sif::DynamicObject** (p. **??**), **sif::Obstacle** (p. **??**)

Definition at line 45 of file square.hpp.

5.45.2 Constructor & Destructor Documentation

5.45.2.1 template<int Dim, class Type > sif::Square< Dim, Type >::Square (Coordonate< Dim, Type > *_coord*, std::vector< std::pair< Type, Type >> *_boundaries*)

Constructor.

Parameters

_coord	Coordonates
_boundaries	Boundaries in a Dim space

Definition at line 33 of file square.cpp.

5.45.3 Member Function Documentation

5.45.3.1 template<int Dim, class Type > bool sif::Square< Dim, Type >::colliding (Coordonate< Dim, Type > *_coord*) [virtual]

Colliding.

Returns

Boolean to check collision

Definition at line 39 of file square.cpp.

5.45.3.2 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const [inherited]

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

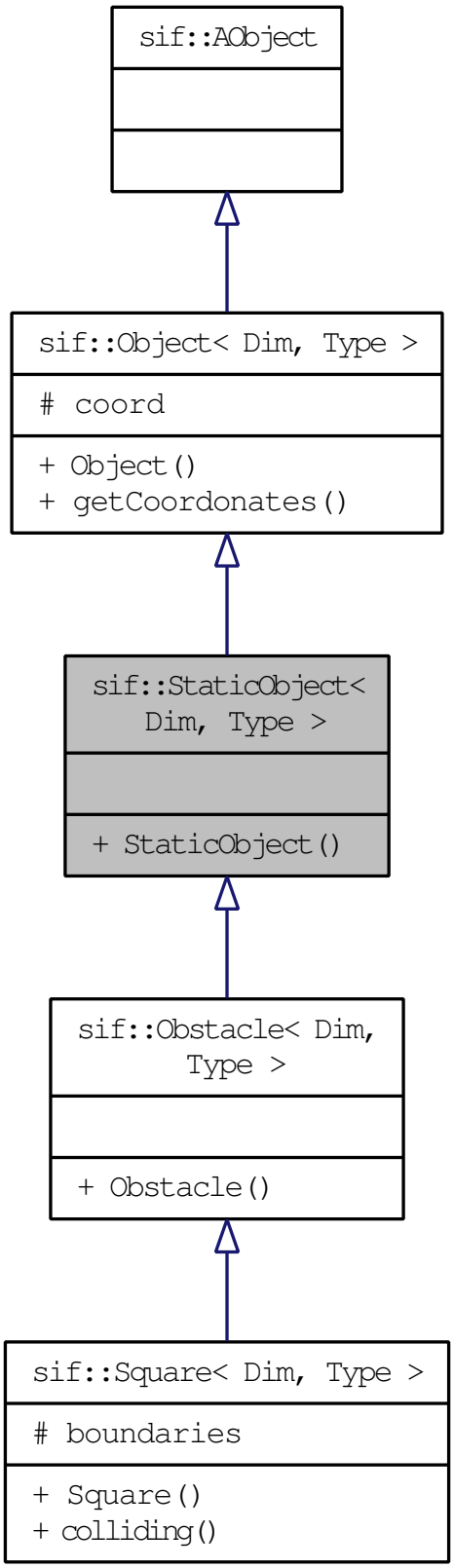
- square.hpp
- square.cpp

5.46 sif::StaticObject< Dim, Type > Class Template Reference

StaticObject (p. **??**) : Objects that cannot evolve in time.

`#include <staticObject.hpp>`

Inheritance diagram for sif::StaticObject< Dim, Type >:



Public Member Functions

- **StaticObject** (**Coordonate**< Dim, Type > _coord)

Constructor.

- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Attributes

- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.46.1 Detailed Description

template<**int Dim, class Type**>**class sif::StaticObject**< **Dim, Type** >

StaticObject (p. **??**) : Objects that cannot evolve in time.

It defines an abstract API for object that cannot move during the simulation such as **sif::Obstacle** (p. **??**) that defines obstacle in space.

See Also

sif::Environment (p. **??**), **sif::Object** (p. **??**), **sif::DynamicObject** (p. **??**), **sif::Obstacle** (p. **??**)

Definition at line 46 of file staticObject.hpp.

5.46.2 Constructor & Destructor Documentation

5.46.2.1 template<int Dim, class Type > sif::StaticObject< Dim, Type >::StaticObject (Coordonate< Dim, Type > _coord)

Constructor.

Parameters

_coord	Coordonates of the resource

Definition at line 33 of file staticObject.cpp.

5.46.3 Member Function Documentation

5.46.3.1 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const [inherited]

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

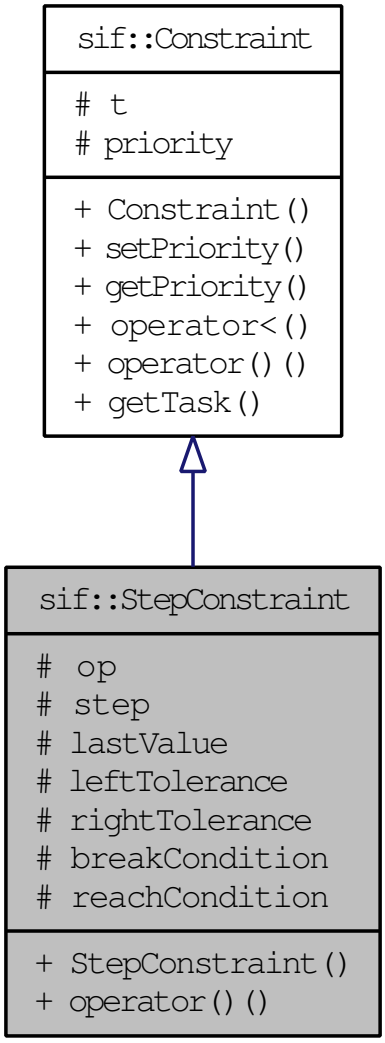
- staticObject.hpp
- staticObject.cpp

5.47 sif::StepConstraint Class Reference

StepConstraint (p. **??**) : Special constraint.

`#include <stepConstraint.hpp>`

Inheritance diagram for sif::StepConstraint:



Public Member Functions

- **StepConstraint** (unsigned _priority, **Task** &_task, ConstraintComp _comp, int _step, std::function< void()> _reachCondition=[]{}, std::function< void()> _breakCondition=[]{}, int _rT=0, int _lT=0)

  Default constructor.
- virtual bool **operator()** ()

  Check the constraint satisfaction and launch callback function if needed.
- void **setPriority** (unsigned _priority)

  Change the priority.
- unsigned **getPriority** () const

  Get the priority.
- bool **operator**< (**Constraint** &_const)

  Comparaison operator based on priority.

- const **Task** & **getTask** () const

  Get task.

Protected Attributes

- ConstraintComp **op**

  Comparaison operator.
- int **step**

  Value of the step.
- bool **lastValue**

  Value at the last check.
- int **leftTolerance**

  Value for the left tolerance.
- int **rightTolerance**

  Value for the right tolerance.
- std::function< void()> **breakCondition**

  Called when the constraint is not satisfied.
- std::function< void()> **reachCondition**

  Called when the constraint is satisfied.
- **Task** & **t**

  Task (p. **??**) *concerned by the constraint.*
- unsigned **priority**

  Priority of the constraint.

5.47.1 Detailed Description

StepConstraint (p. **??**) : Special constraint.

StepConstraint (p. **??**) is a special constraint related to the step.

See Also

sif::Constraint (p. **??**)

Definition at line 53 of file stepConstraint.hpp.

5.47.2 Constructor & Destructor Documentation

5.47.2.1 sif::StepConstraint::StepConstraint (unsigned *_priority*, Task & *_task*, ConstraintComp *_comp*, int *_step*, std::function< void()> *_reachCondition* = [ ]{}, std::function< void()> *_breakCondition* = [ ]{}, int *_rT* = 0, int *_lT* = 0)

Default constructor.

Parameters

_priority	Priority of the task
_task	**Task** (p. **??**) concerned by the constraint
_comp	Comparaison operator
_step	Step value
_reachCondition	Callback function called when the constraint is satisfied
_breakCondition	Callback function called when the constraint is not satisfied
_rT	Right value of the tolerance interval
_lT	Left value of the tolerance interval

Definition at line 33 of file stepConstraint.cpp.

References lastValue.

5.47.3 Member Function Documentation

5.47.3.1 bool sif::StepConstraint::operator() ( ) [virtual]

Check the constraint satisfaction and launch callback function if needed.

Returns

boolean

Implements **sif::Constraint** (p. **??**).

Definition at line 52 of file stepConstraint.cpp.

References breakCondition, sif::Task::getValue(), lastValue, leftTolerance, op, reachCondition, rightTolerance, step, and sif::Constraint::t.

5.47.3.2 void sif::Constraint::setPriority (unsigned *_priority*) [inherited]

Change the priority.

Parameters

_priority	**Constraint** (p. **??**) priority

Definition at line 39 of file constraint.cpp.

References sif::Constraint::priority.

5.47.3.3 unsigned sif::Constraint::getPriority ( ) const [inherited]

Get the priority.

Returns

Constraint (p. **??**) priority

Definition at line 44 of file constraint.cpp.

References sif::Constraint::priority.

Referenced by sif::Constraint::operator<().

5.47.3.4 bool sif::Constraint::operator< (Constraint & *_const*) [inherited]

Comparaison operator based on priority.

Parameters

_const	**Constraint** (p. **??**) to compare

Returns

boolean

Definition at line 49 of file constraint.cpp.

References sif::Constraint::getPriority(), and sif::Constraint::priority.

5.47.3.5 const Task & sif::Constraint::getTask ( ) const [inherited]

Get task.

Returns

Associated task

Definition at line 59 of file constraint.cpp.

References sif::Constraint::t.

Referenced by sif::SimpleStrategy< Dim, Type >::SimpleStrategy().

The documentation for this class was generated from the following files:

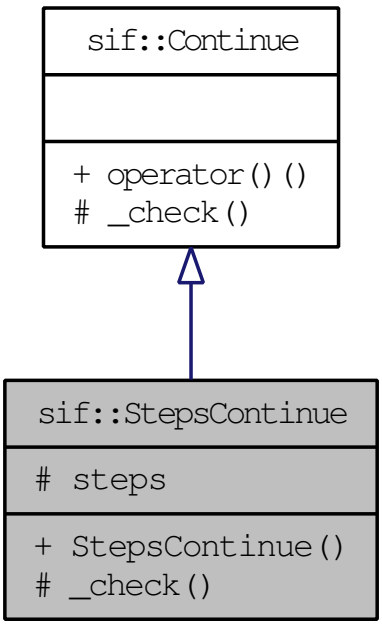
- stepConstraint.hpp
- stepConstraint.cpp

5.48 sif::StepsContinue Class Reference

StepsContinue (p. **??**) : Criterion for ending the calculation / simulation after a user-defined number of steps.

`#include <stepsContinue.hpp>`

Inheritance diagram for sif::StepsContinue:



Public Member Functions

- **StepsContinue** (unsigned _steps)

Default constructor.

- virtual bool **operator()** ()

  Check the criterion.

Protected Member Functions

- virtual bool **_check** ()

  Implementation of the test.

Protected Attributes

- unsigned **steps**

  Number of steps to reach.

5.48.1 Detailed Description

StepsContinue (p. **??**) : Criterion for ending the calculation / simulation after a user-defined number of steps.

Criterion for ending the calculation / simulation after a user-defined number of steps

See Also

sif::Continue (p. **??**), **sif::TimeContinue** (p. **??**), **sif::Controller** (p. **??**)

Definition at line 44 of file stepsContinue.hpp.

5.48.2 Member Function Documentation

5.48.2.1 virtual bool sif::StepsContinue::_check ( ) [protected], [virtual]

Implementation of the test.

Returns

boolean

Implements **sif::Continue** (p. **??**).

5.48.2.2 bool sif::Continue::operator() ( ) [virtual], [inherited]

Check the criterion.

Returns

boolean

Definition at line 34 of file continue.cpp.

References sif::Continue::_check().

The documentation for this class was generated from the following files:

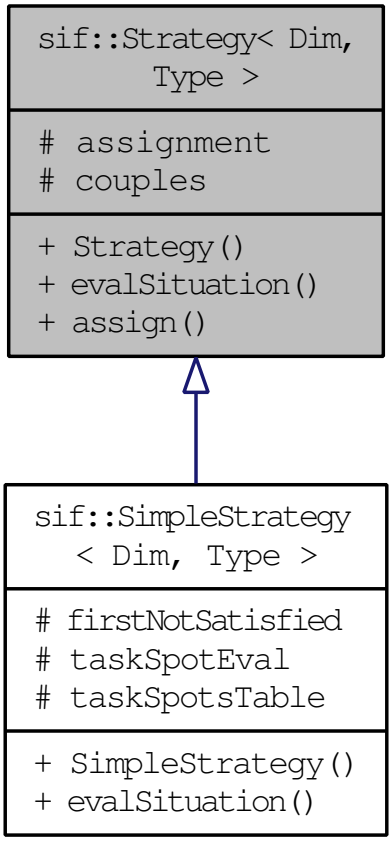
- stepsContinue.hpp
- stepsContinue.cpp

5.49 sif::Strategy< Dim, Type > Class Template Reference

Strategy (p. **??**) : Defines the behavior of the **IA** (p. **??**) at a precise moment.

`#include <strategy.hpp>`

Inheritance diagram for sif::Strategy< Dim, Type >:



Public Member Functions

- **Strategy** (**Assignment** &_assignment)

  Constructor.
- virtual int **evalSituation** (**SpatialData** &_spatialData, **ConstraintSystem** &_constraintSystem)=0

  Perform the evaluation of the situation.
- virtual std::map< **AResource** *, **ATaskSpot** * > **assign** ()

  Perform the assignment of resources.

Protected Attributes

- **Assignment** & **assignment**

  Assignment *(p. **??**) algorithm.*
- std::map< std::pair< **AResource** *, **ATaskSpot** * >, int > **couples**

  Couples to be assigned.

5.49.1 Detailed Description

template<int Dim, class Type>class sif::Strategy< Dim, Type >

Strategy (p. **??**) : Defines the behavior of the **IA** (p. **??**) at a precise moment.

The strategy define the whole process of the evaluation and assignment

See Also

sif::Model (p. **??**),

Definition at line 48 of file strategy.hpp.

5.49.2 Constructor & Destructor Documentation

5.49.2.1 template<int Dim, class Type > sif::Strategy< Dim, Type >::Strategy (Assignment & *_assignment*)

Constructor.

Parameters

_assignment	Assigment algorithm

Definition at line 33 of file strategy.cpp.

5.49.3 Member Function Documentation

5.49.3.1 template<int Dim, class Type> virtual int sif::Strategy< Dim, Type >::evalSituation (SpatialData & *_spatialData,* ConstraintSystem & *_constraintSystem*) [pure virtual]

Perform the evaluation of the situation.

Parameters

_spatialData	Data structures of the environments
_constraint-System	Constraints on tasks

Returns

Value of the evaluation

Implemented in **sif::SimpleStrategy< Dim, Type >** (p. **??**).

5.49.3.2 template<int Dim, class Type > std::map< AResource *, ATaskSpot * > sif::Strategy< Dim, Type >::assign ( ) [virtual]

Perform the assignment of resources.

Returns

assigment information

Definition at line 38 of file strategy.cpp.

The documentation for this class was generated from the following files:

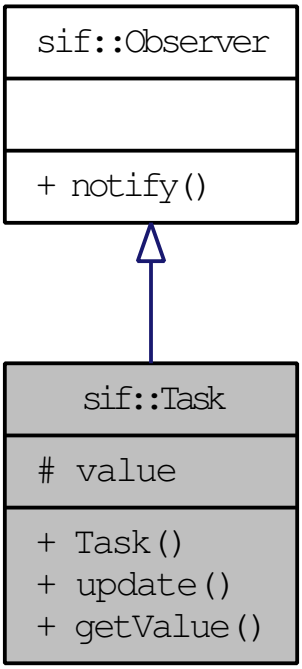
- strategy.hpp
- strategy.cpp

5.50 sif::Task Class Reference

Task (p. **??**) : Abstract object.

`#include <task.hpp>`

Inheritance diagram for sif::Task:



Public Member Functions

- **Task** (int _value=0)

  Constructor.
- void **update** (double _time, std::function< int(int &, double _time)> _f)

  *Update **Task** (p. **??**).*
- int **getValue** () const

  Get the current value.
- void **notify** ()

  Notify observable.

Protected Attributes

- int **value**

  Value of the task.

5.50.1 Detailed Description

Task (p. **??**) : Abstract object.

A **Task** (p. **??**) is materialized by a counter.

See Also

sif::Task (p. **??**), sif::PeriodicTaskSpot, **sif::Observable** (p. **??**)

Definition at line 46 of file task.hpp.

5.50.2 Constructor & Destructor Documentation

5.50.2.1 sif::Task::Task (int *_value* = 0)

Constructor.

Parameters

_value	Defaut value

Definition at line 35 of file task.cpp.

5.50.3 Member Function Documentation

5.50.3.1 void sif::Task::update (double *_time*, std::function< int(int &, double _time)> *_f*)

Update **Task** (p. **??**).

Parameters

_time	Ellapsed time since the last update
_f	The result will remplace value and the int parameter will be the initial value

Definition at line 38 of file task.cpp.

References value.

Referenced by sif::ATaskSpot::update().

5.50.3.2 int sif::Task::getValue ( ) const

Get the current value.

Returns

value

Definition at line 44 of file task.cpp.

References value.

Referenced by sif::StepConstraint::operator()().

The documentation for this class was generated from the following files:

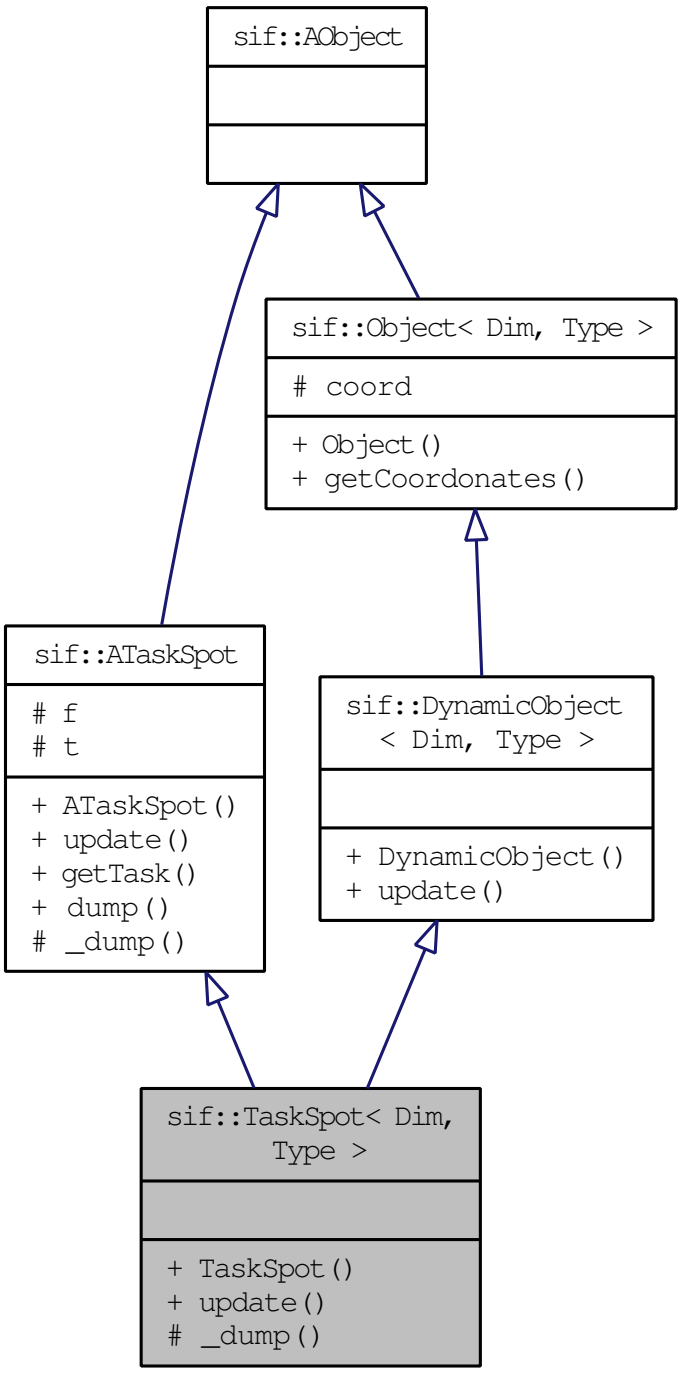
- task.hpp
- task.cpp

5.51 sif::TaskSpot< Dim, Type > Class Template Reference

TaskSpot (p. **??**) : Physical object that can modify a task.

`#include <taskSpot.hpp>`

Inheritance diagram for sif::TaskSpot< Dim, Type >:



Public Member Functions

- **TaskSpot** (**Coordonate**< Dim, Type > _coord, **Task** &_t, std::function< int(int &, double)> _f)

  Constructor.
- virtual void **update** (double _time)

  Update Ressource.
- const **Task** & **getTask** () const

  *Return **Task** (p. **??**).*

- void **dump** ()

  Debug function to show all informations about the class.
- **Coordonate**< Dim, Type > **getCoordonates** () const

  Get coordonates of the object.

Protected Member Functions

- void **_dump** ()

  Implementation of dump function.

Protected Attributes

- std::function< int(int &, double)> **f**

  Function in order to update task.
- **Task** & **t**

  Attached task.
- **Coordonate**< Dim, Type > **coord**

  Coordonates of the object.

5.51.1 Detailed Description

template<int Dim, class Type>class sif::TaskSpot< Dim, Type >

TaskSpot (p. **??**) : Physical object that can modify a task.

The **TaskSpot** (p. **??**) is an object that can modify a task when a specific action occured. Different types of **TaskSpot** (p. **??**) are provided such as periodic ones.

See Also

sif::Task (p. **??**), sif::PeriodicTaskSpot, **sif::Observable** (p. **??**)

Definition at line 52 of file taskSpot.hpp.

5.51.2 Constructor & Destructor Documentation

5.51.2.1 template<int Dim, class Type > sif::TaskSpot< Dim, Type >::TaskSpot (Coordonate< Dim, Type > *_coord*, Task & *_t*, std::function< int(int &, double)> *_f*)

Constructor.

Parameters

_coord	Coordonates of the taskSpot
_t	The associated task
_f	Function which serves to update the task

Definition at line 35 of file taskSpot.cpp.

5.51.3 Member Function Documentation

5.51.3.1 template<int Dim, class Type > Coordonate< Dim, Type > sif::Object< Dim, Type >::getCoordonates ( ) const [inherited]

Get coordonates of the object.

Returns

Coordonate (p. **??**)

Definition at line 36 of file object.cpp.

Referenced by sif::Environment< Dim, Type, Data >::addObject(), and sif::Resource< Dim, Type, Data >::move().

The documentation for this class was generated from the following files:

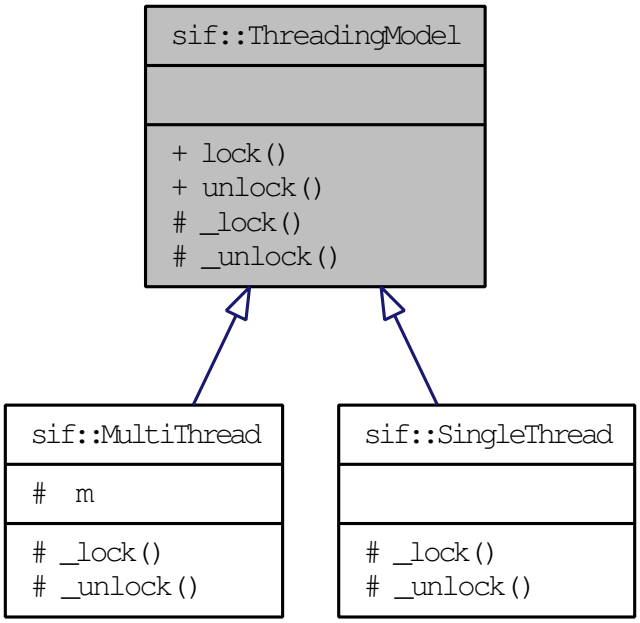
- taskSpot.hpp
- taskSpot.cpp

5.52 sif::ThreadingModel Class Reference

ThreadingModel (p. **??**) : Threading Policy.

`#include <threadingModel.hpp>`

Inheritance diagram for sif::ThreadingModel:



Public Member Functions

- virtual void **lock** () final

  Lock the scope.

- virtual void **unlock** () final

  Unlock the scope.

Protected Member Functions

- virtual void **_lock** ()=0

  Implemention of the lock method.
- virtual void **_unlock** ()=0

  Implemention of the unlock method.

5.52.1 Detailed Description

ThreadingModel (p. **??**) : Threading Policy.

Define a policy regarding the threading environnement.

See Also

sif::SingleThread (p. **??**), **sif::MultiThread** (p. **??**)

Definition at line 42 of file threadingModel.hpp.

The documentation for this class was generated from the following files:

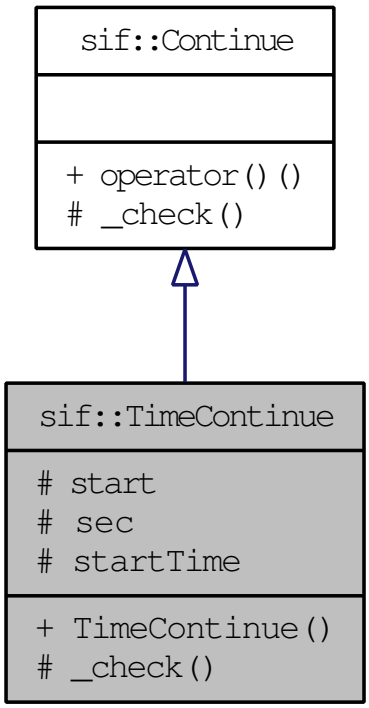
- threadingModel.hpp
- threadingModel.cpp

5.53 sif::TimeContinue Class Reference

TimeContinue (p. **??**) : Criterion for ending the calculation / simulation after a user-defined time.

`#include <timeContinue.hpp>`

Inheritance diagram for sif::TimeContinue:



Public Member Functions

- **TimeContinue** (double _sec)

  Constructor with time in seconds.
- virtual bool **operator()** ()

  Check the criterion.

Protected Member Functions

- virtual bool **_check** ()

  Implementation of the test.

Protected Attributes

- bool **start**

  Boolean to know if the chrono is started.
- double **sec**

  Number of seconds to reach.
- std::chrono::steady_clock::time_point **startTime**

  Time when the chrono has been started.

5.53.1 Detailed Description

TimeContinue (p. **??**) : Criterion for ending the calculation / simulation after a user-defined time.

Criterion for ending the calculation / simulation after a user-defined time

See Also

sif::Continue (p. **??**), **sif::StepsContinue** (p. **??**), **sif::Controller** (p. **??**)

Definition at line 46 of file timeContinue.hpp.

5.53.2 Constructor & Destructor Documentation

5.53.2.1 sif::TimeContinue::TimeContinue (double _sec)

Constructor with time in seconds.

Parameters

_sec	Time in seconds

Definition at line 37 of file timeContinue.cpp.

5.53.3 Member Function Documentation

5.53.3.1 virtual bool sif::TimeContinue::_check ( ) [protected], [virtual]

Implementation of the test.

Returns

boolean

Implements **sif::Continue** (p. **??**).

5.53.3.2 bool sif::Continue::operator() ( ) [virtual],[inherited]

Check the criterion.

Returns

boolean

Definition at line 34 of file continue.cpp.

References sif::Continue::_check().

The documentation for this class was generated from the following files:

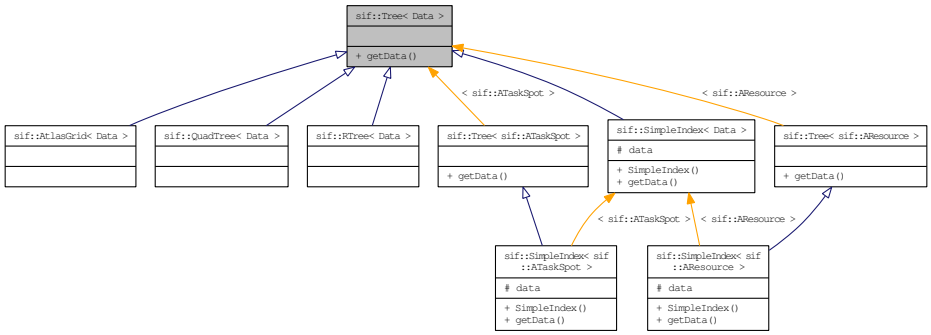
- timeContinue.hpp
- timeContinue.cpp

5.54 sif::Tree< Data > Class Template Reference

Tree (p. **??**) : General API for tree.

`#include <tree.hpp>`

Inheritance diagram for sif::Tree< Data >:



Public Member Functions

- virtual std::vector< Data * > **getData** ()=0

  Return a vector of all objects of the tree.

5.54.1 Detailed Description

template<class Data>class sif::Tree< Data >

Tree (p. **??**) : General API for tree.

Tree (p. **??**) is the class which list trees.

See Also

sif::TreeFactory (p. **??**), **sif::RTree** (p. **??**), **sif::QuadTree** (p. **??**), **sif::AtlasGrid** (p. **??**)

Definition at line 43 of file tree.hpp.

5.54.2 Member Function Documentation

5.54.2.1 template<class Data> virtual std::vector<Data*> sif::Tree< Data >::getData ( ) [pure virtual]

Return a vector of all objects of the tree.

Returns

Vector of objets

Implemented in **sif::SimpleIndex< Data >** (p. **??**), **sif::SimpleIndex< sif::ATaskSpot >** (p. **??**), and **sif::SimpleIndex< sif::AResource >** (p. **??**).

The documentation for this class was generated from the following file:

- tree.hpp

5.55 sif::TreeFactory Class Reference

TreeFactory (p. **??**) : General API for tree.

`#include <treeFactory.hpp>`

5.55.1 Detailed Description

TreeFactory (p. **??**) : General API for tree.

TreeFactory (p. **??**) is the factory for trees.

See Also

sif::Tree (p. **??**)

Definition at line 42 of file treeFactory.hpp.

The documentation for this class was generated from the following file:

- treeFactory.hpp