Projet Qualité Logicielle

Hexanôme H4314

Kevin ANTOINE - Karim BENHMIDA - Valentin COMTE - Mehdi KITANE - Robin RICARD - Bruno SIVANANDAN
Chef de projet : Valentin Comte

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

INITIALISATION

Sujet

Notre client désire mettre en place un système de livraison par drones et souhaite disposer d'un système de guidage des drones efficace selon des critères bien définis.

Notre étude va porter sur plusieurs parties :

- Une modélisation UML du projet
- Une étude de l'algorithme de calcul de chemin
- Une étude de l'algorithme de communication entre drones

Découpage des tâches

Le projet se découpe en trois grandes branches : la modélisation UML et les deux études algorithmes.

Chaque branche peut se découper en plusieurs tâches qui seront réalisées par binômes :

- Modélisation UML
  - Etude préliminaire et ébauche des classes
  - Création du diagramme de classes incluant pré/post-conditions
  - Création de diagramme d'états-transitions par classe
  - Etudier la cohérence des invariants
  - Prouver l'indépendance du noyau d'invariants
  - Rechercher les relations des autres invariants avec le noyau
  - Illuster des éléments pertinents à travers différents scénarios
- Algorithme séquentiel
  - Résoudre le problème de base : trouver les successeurs et les atteignables
  - Déterminer l'algorithme de calcul de chemin
  - Dériver l'algorithme obtenu
- Algorithme concurrent
  - Trouver l'algorithme concurrent
  - Dériver l'algorithme
  - Effectuer les tests sur Spin

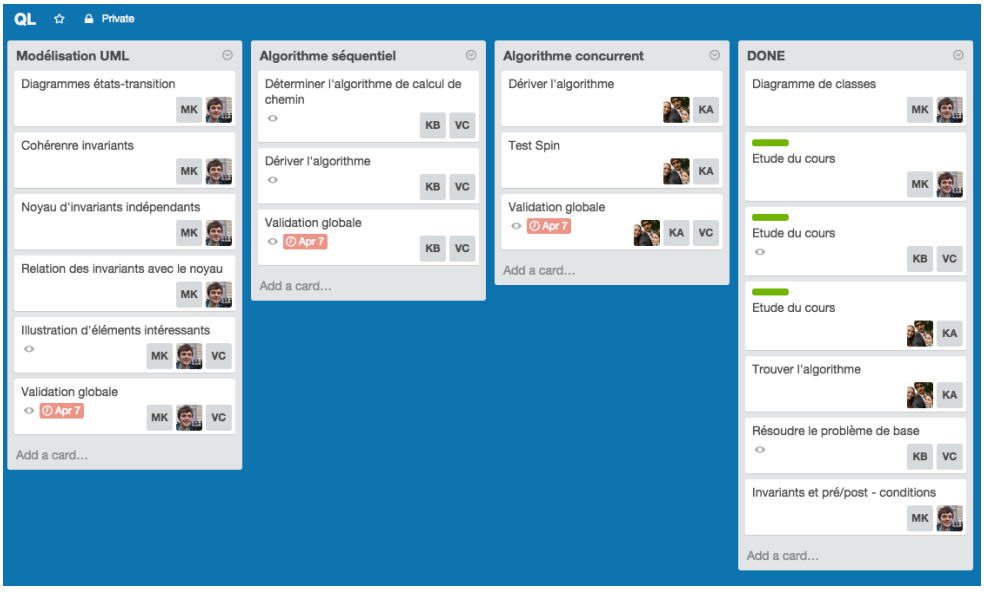
Répartition des tâches

La répartition des tâches entre les différents membres du projet s'est faite selon les affinités de chacun. Le travail s'effectue globalement en binôme afin d'être plus efficace lors de la réalisation de chaque tâche.
Chaque binôme est affilié à une branche du projet afin de garantir un meilleur suivi des tâches. Les trois branches du projet n'étant pas forcément équilibrées entre elles les binômes ayant terminé leurs tâches en avance sont réattribués à la branche demandant le plus gros volume de travail restant.

La répartition des branches au début du projet s'est effectuée comme le montre le tableau ci-dessous

Rôle	**Personne**
Chef Projet	Valentin Comte
Modélisation UML	Robin Ricard / Mehdi Kitane
Algorithme Sequentiel	Karim Benhmida / Valentin Comte
Algorithme Concurrent	Kevin Antoine / Bruno Savanandan

Afin de permettre à chacun de mettre à jour l'avancement de ses tâches et afin de centraliser les tâches à réaliser nous avons décider d'utiliser Trello.
Voici un exemple de notre utilisation de Trello :



Cette organisation permet à chaque membre d'avancer à son rythme, notamment grâce au fait qu'il

n'y ait pas de planning strict à respecter si ce ne sont les dates d'avancement majeure (date de fin de projet, date de création des rendus, dates d'avancement de mi-projet).
Ce choix d'organisation a permis à chacun de gérer la charge importante de travail qui pesait au moment du projet. Ainsi certains ont pu décider de prendre de l'avance sur l'avancement prévu et d'autres ont pu choisir de prendre un peu de retard pour se concentrer sur leurs DS par exemple.

Risques :

Risques liés à la gestion du projet :

- Dépassement des échéances
  Risque = moyen
  Impact = élevé
  Description = le fait d'adopter une gestion de projet "souple" implique le risque d'accepter "trop" de souplesse et ainsi permettre plus facilement les dépassements d'échéances, qui pourraient se répercuter sur tout l'avancement du projet

- Absence de membres
  Risque = faible
  Impact = faible
  Description = nous ne sommes jamais à l'abris d'incidents pouvant mener à l'absence de certains membres. Notre organisation nous permettra de gérer facilement les absences en réatribuant si nécessaire les tâches

- Manque de motivation
  Risque = moyen
  Impact = élevé
  Description = dû à la période de réalisation de ce projet, il sera facile pour les membres de perdre en motivation sur le projet, il faudra donc garder l'avancement du projet à l'esprit de chacun, sans pour autant changer la gestion de projet mise en place

Risques liés au projet :

- Non-respect des spécifications
  Risque = moyen
  Impact = élevé
  Description = comme dans tout projet le notre n'est pas à l'abris d'échecs. Cependant le fait que le projet soit découpé en 3 grandes parties nous permettra, si jamais il y a un soucis avec l'une des parties, de tout de même faire avancer le projet sur les autres parties.

- Manque de maîtrise

  Risque = moyen

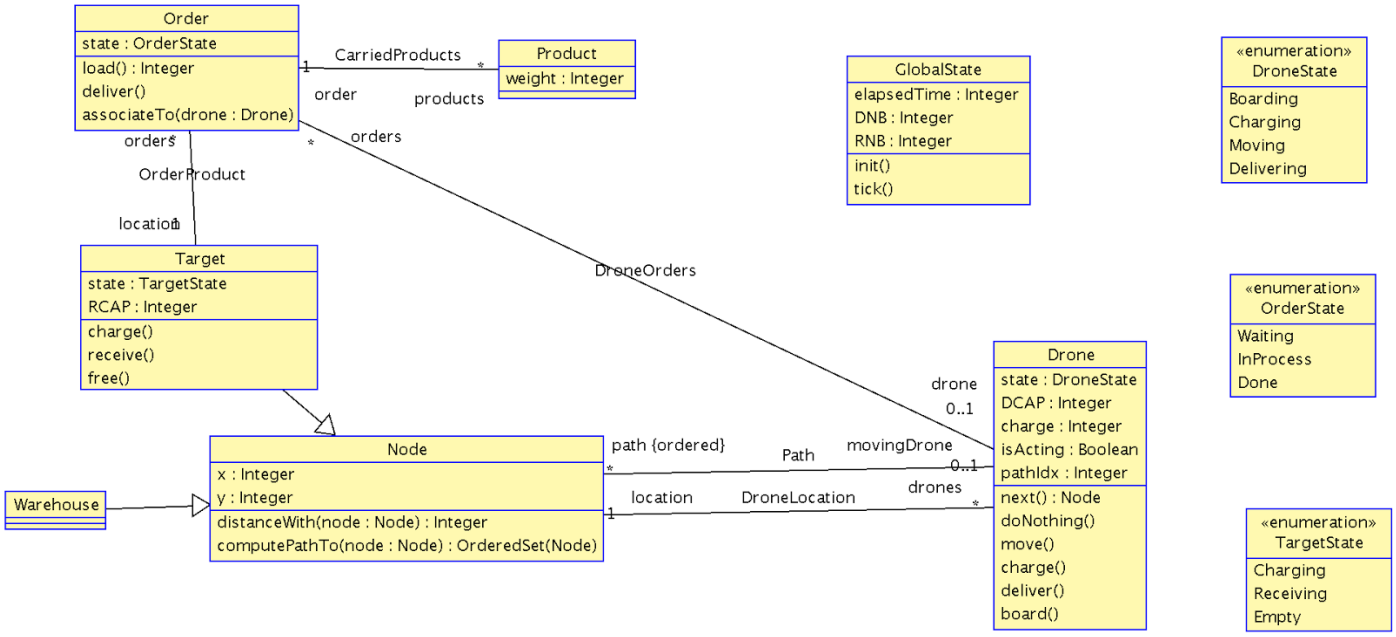
  Impact = moyen

  Description = étant en plein apprentissage, nous sommes susceptibles de ne pas complètement maîtriser le sujet, menacant de nous retarder dans la réalisation du projet. Nous pourrons compter sur nos enseignants et camarades pour nous sortir de ce mauvais pas si jamais nous avons des problèmes.

MODELISATION UML

Cette partie décrit la modélisation mise en oeuvre pour mettre en place ce système de drones. Dans un premier temps, il conviendra de modéliser les classes utilisées puis ensuite les state machines liées à ces classes. Dans un second temps, on complètera les contraintes introduites par le modèle UML par des contraintes OCL: des invariants sur les classes et des pré et post conditons sur les méthodes. Pour terminer, on validera que le modèle est cohérent en utilisant ASSL (A Snapshot Sequence Language) qui va générer des instances de classes automatiquement en prenant en compte les invariants.

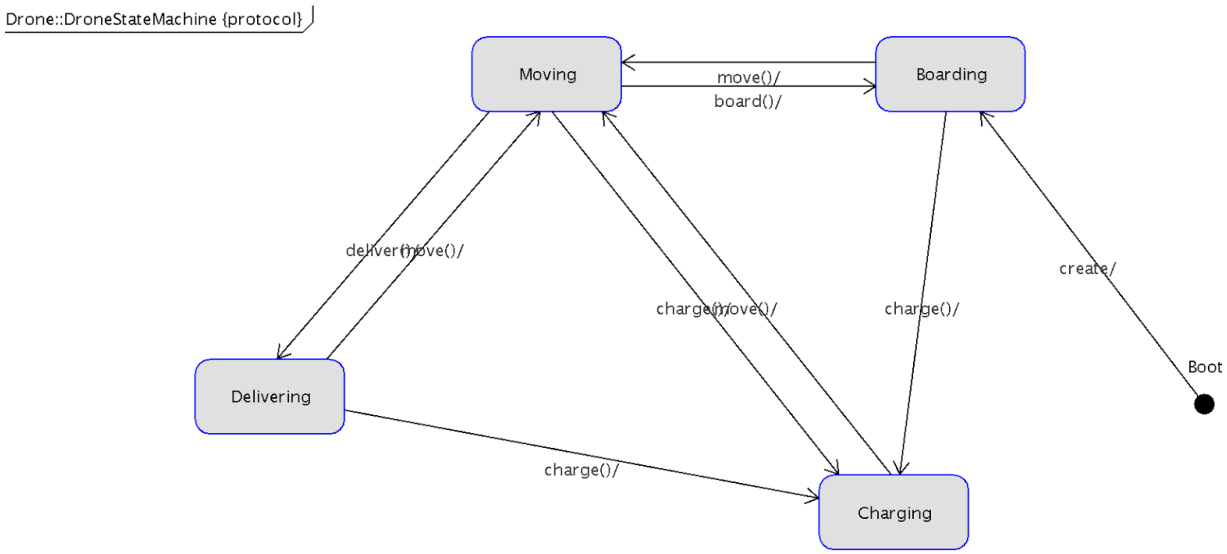
Diagramme de classe



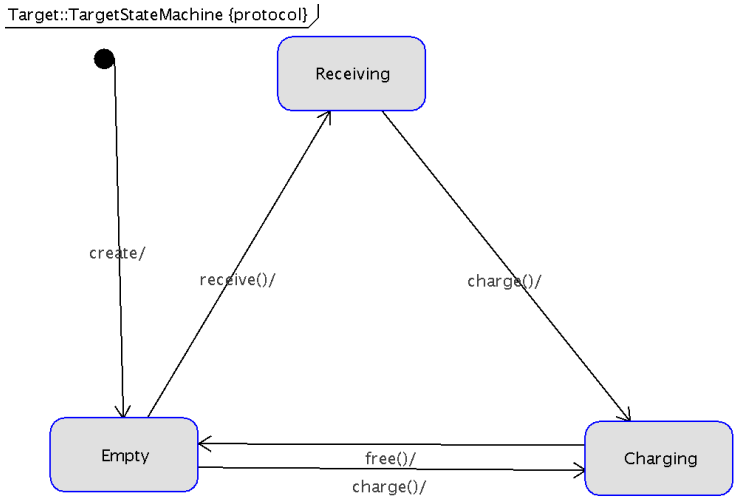
On remarque que cette architecture est orientée héritage et non pas composition. Cependant, on garde la position du Drone en pointant vers un noeud (le Drone n'est pas lui même un noeud).

State machines

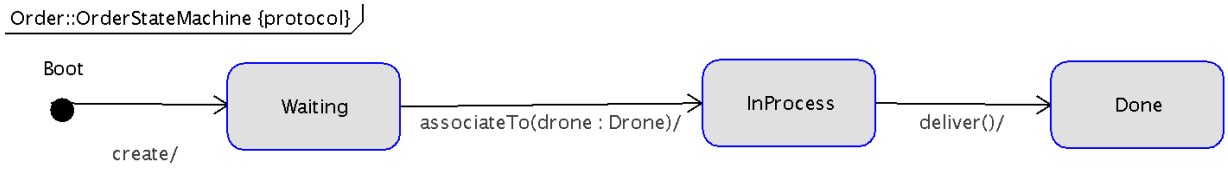
Classe Drone



Classe Target



Classe Order



Contraintes OCL

Invariants

Liste des invariants

Contexte GlobalState :

Il y'a DNB drones :

```
inv droneNumbers : Drone.allInstances()->size() = self.DNB
```

Il y'a RNB targets :

```
inv targetNumbers : Target.allInstances()->size() = self.RNB
```

Le temps est une variable positive :

```
inv positiveElapsedTime : self.elapsedTime >= 0
```

context Node

La coordonnée X d'une instance de Node est une variable positive :

```
inv positiveX : self.x >= 0
```

La coordonnée Y d'une instance de Node est une variable positive :

```
inv positiveY : self.y >= 0
```

On ne peut avoir qu'un instance d'un noeud par "case" (repérés par leurs coordonnées x,y) :

```
inv SingleNodePerCase : Node.allInstances()->select(n | n.x = self.x and n.y = self.y)->size() = 1
```

Un Noeud possede entre 0 et 4 voisins

```
inv successors : Node.allInstances()->select(n | (n.x = self.x - 1 and n.y = self.y) or (n.x = self.x+1 and n.y = self.y) or (n.x = self.x and n.y = self.y -1) or (n.x = self.y and n.y = self.y+1))->size()<=4
```

A un moment donné, il y'a au plus un drone à chaque intersection de la grille.

```
inv oneDronePerNode : not(self.oclIsTypeOf(Warehouse)) implies self.drones->size() <= 1
```

context Drone

La batterie d'un drone est comprise entre 0 et 3

```
inv batteryCapacity : self.charge <= 3 and self.charge>=0
```

La capacité d'un drone est une valeur entière positive :

```
inv positiveCapacity : self.DCAP >= 0
```

Un drone ne peut pas soulever plus de produits que sa capacité initiale DCAP

```
inv ordersNotHeavierThanCapacity : self.orders.products.weight->sum() <= self.DCAP
```

context Order

Il n’y a pas de contraintes pour Order

context Product

Un produit a un poids positif

```
inv positiveWeight : self.weight >= 0
```

context Target

Un receptacle a une capacité positive

```
inv positiveRCAP : self.RCAP >= 0
```

Deux receptacles sont séparés d'une distance inférieure ou egale a 2 (receptacle actuel et un autre)

```
inv targetDistance : Target.allInstances()->excluding(self)->exists(r| ((self.x-r.x).abs()+(self.y-r.y).abs() <= 2) ) or
Target.allInstances()->size() = 1
```

context Warehouse

Il existe un seul entrepot

```
inv oneWarehouse : Warehouse.allInstances()->size() = 1
```

Il existe au moins un receptacle voisin de l’entrepot.

```
inv targetCloseToWareHouse : Target.allInstances()->exists(r | (self.x-r.x).abs()+(self.y-r.y).abs() = 1 )
```

Un récéptacle ne peut etre sur la même case qu'un warehouse

```
inv wareHouseDifferentFromTarget : Target.allInstances()->forAll(r| not(self.x=r.x and self.y=r.y) )
```

Pre et post conditions

Etat initial du systeme

context GlobalState::init()

```
post dronesInWarehouse: let w=
        Warehouse.allInstances().any(true) in Drone.allInstances()->forAll(d | d.location = w)
post initTime: self.elapsedTime = 0
post actingDrones: Drone.allInstances()->forAll(d | d.isActing)
```

Avance dans le temps

context GlobalState::tick()

-- Les drones sont actifs avant de bouger
pre actingDrones: Drone.allInstances()->forAll(d | d.isActing)
post incTime: self.elapsedTime = self.elapsedTime@pre + 1

-- Les drones ont bougé, ils sont inactifs
post idleDrones: Drone.allInstances()->forAll(d | not d.isActing)

Actions du drone

context Drone::doNothing()
post sameState: self = self@pre

context Drone::move()
pre mustBeActing: self.isActing
pre mustHaveCharge: self.charge > 0
pre emptyNextNode: self.next().drones->size = 0
pre mustBeMoving: self.state = DroneState::Moving
post hasMovedToNext: self.location = self@pre.next()
post consumeCharge: self.charge = self.charge@pre - 1

context Drone::charge()
pre mustBeActing: self.isActing
pre mustBeUnderMaxCharge: self.charge < 3
pre mustbeonTarget: Target.allInstances()->exists(t | t = self.location)
post mustBeCharging: self.state = DroneState::Charging
post incCharge: self.charge = self.charge@pre + 1

context Drone::deliver()
pre mustBeActing: self.isActing
pre inDeliveringLocation:
self.orders->exists(o | o.state = OrderState::InProcess and o.location = self.location)
post mustBeDelivering: self.state = DroneState::Delivering
post updateDeliveryStatus:
self.orders->exists(o | o.state@pre = OrderState::InProcess and o.location = self.location and o.state = OrderState::Done)

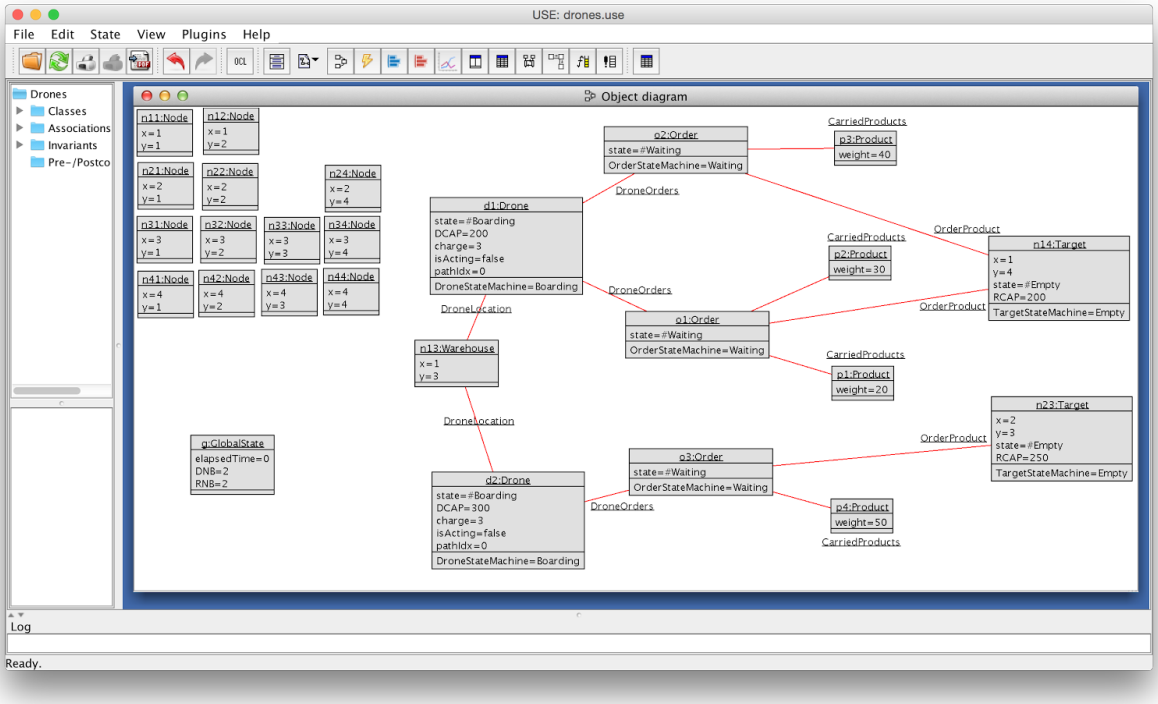
context Drone::board()
pre mustBeActing: self.isActing
pre inWarehouse: Warehouse.allInstances()->exists(w | self.location = w)
post mustBeBoarding: self.state = DroneState::Boarding
post mustHaveChangedOrderState: self.orders->exists(o | o.state = OrderState::InProcess and o.state@pre = OrderState::Waiting)

Test des invariants

A l'aide d'une série de commandes de création d'objets par le langage de commande de use, on peut tester un cas simple (manuel). Il conviendra ensuite d'utiliser le langage ASSL pour s'assurer que notre environnement est suffisamment contraint.

Scénario manuel

On réussit à créer un environnement de départ possible dans les bornes du sujet qui respecte tous les invariants que nous avons fixé. On peut voir le diagramme objet de ce cas:



On retrouve ainsi tous les invariants vérifiés:

Invariant	Loaded	Active	Negate	Satisfied
Drone::batteryCapacity	☐	☑	☐	true
Drone::ordersNotHeavierThanCapacity	☐	☑	☐	true
Drone::positiveCapacity	☐	☑	☐	true
GlobalState::droneNumbers	☐	☑	☐	true
GlobalState::positiveElapsedTime	☐	☑	☐	true
GlobalState::targetNumbers	☐	☑	☐	true
Node::SingleNodePerCase	☐	☑	☐	true
Node::oneDronePerNode	☐	☑	☐	true
Node::positiveX	☐	☑	☐	true
Node::positiveY	☐	☑	☐	true
Node::successors	☐	☑	☐	true
Product::positiveWeight	☐	☑	☐	true
Target::positiveRCAP	☐	☑	☐	true
Target::targetDistance	☐	☑	☐	true
Warehouse::oneWarehouse	☐	☑	☐	true
Warehouse::targetCloseToWareHouse	☐	☑	☐	true
Warehouse::wareHouseDifferentFromTarget	☐	☑	☐	true

```
manual.cmd> check
checking structure...
checked structure in 0ms.
checking invariants...
checking invariant (1) `Drone::batteryCapacity': OK.
checking invariant (2) `Drone::ordersNotHeavierThanCapacity': OK.
checking invariant (3) `Drone::positiveCapacity': OK.
checking invariant (4) `GlobalState::droneNumbers': OK.
checking invariant (5) `GlobalState::positiveElapsedTime': OK.
checking invariant (6) `GlobalState::targetNumbers': OK.
checking invariant (7) `Node::SingleNodePerCase': OK.
checking invariant (8) `Node::oneDronePerNode': OK.
checking invariant (9) `Node::positiveX': OK.
checking invariant (10) `Node::positiveY': OK.
checking invariant (11) `Node::successors': OK.
checking invariant (12) `Product::positiveWeight': OK.
checking invariant (13) `Target::positiveRCAP': OK.
checking invariant (14) `Target::targetDistance': OK.
checking invariant (15) `Warehouse::oneWarehouse': OK.
checking invariant (16) `Warehouse::targetCloseToWareHouse': OK.
checking invariant (17) `Warehouse::wareHouseDifferentFromTarget': OK.
checked 17 invariants in 0.009s, 0 failures.
manual.cmd>
```

Pour générer ces captures, il suffit d'écrire dans l'invite de commande : read manual.cmd (se trouvant dans le dossier scenarios)

Scénario d'initialisation automatique

A l'aide d'ASSL nous sommes capables de générer un monde qui répondra aux contraintes du système

Cohérence des invariants

Pour prouver la cohérence des invariants, il nous suffit de générer via le language ASSL, un monde aléatoire qui satisfait tous les invariants. Ainsi, l'existence de cet état, cohérent, permet de prouver cette cohérence des invariants.

Nous avons pour cela crée une procédure via ASSL qui prend en paramètre le nombre de colonnes et de lignes de la grille ainsi que DNB et RNB, le nombre de drones et de receptacles.

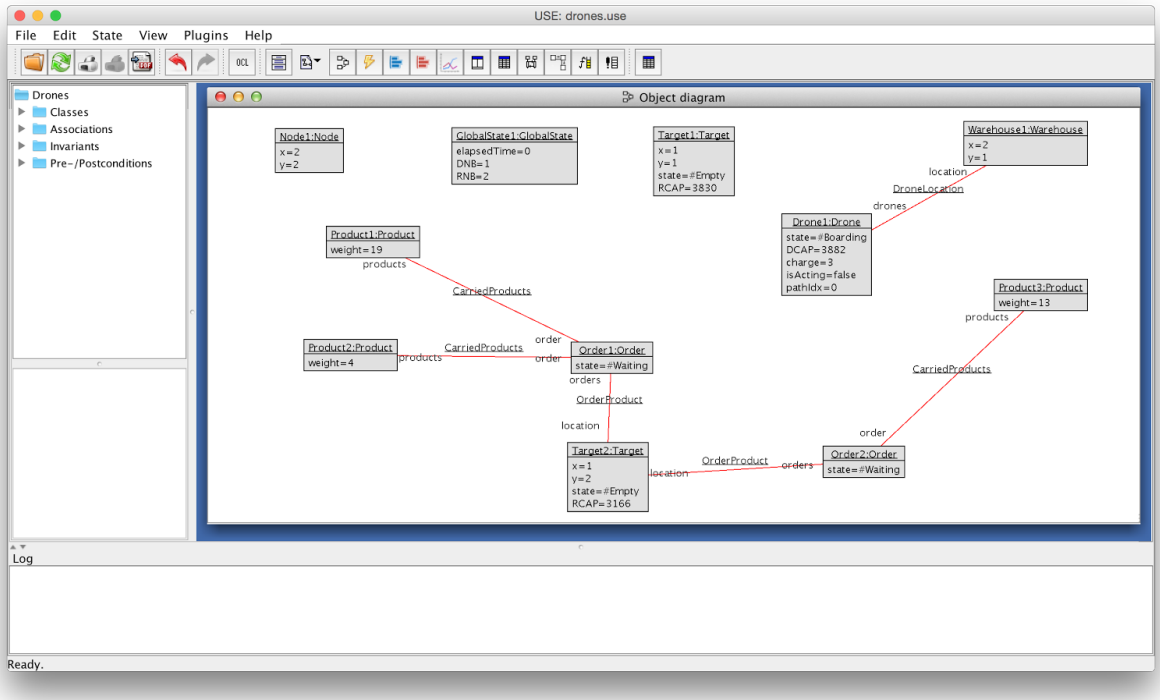
Notre procédure s'occupe dès lors de générer un monde :

La procédure se trouve dans le fichier : scenarios/1.simple/simple.assl

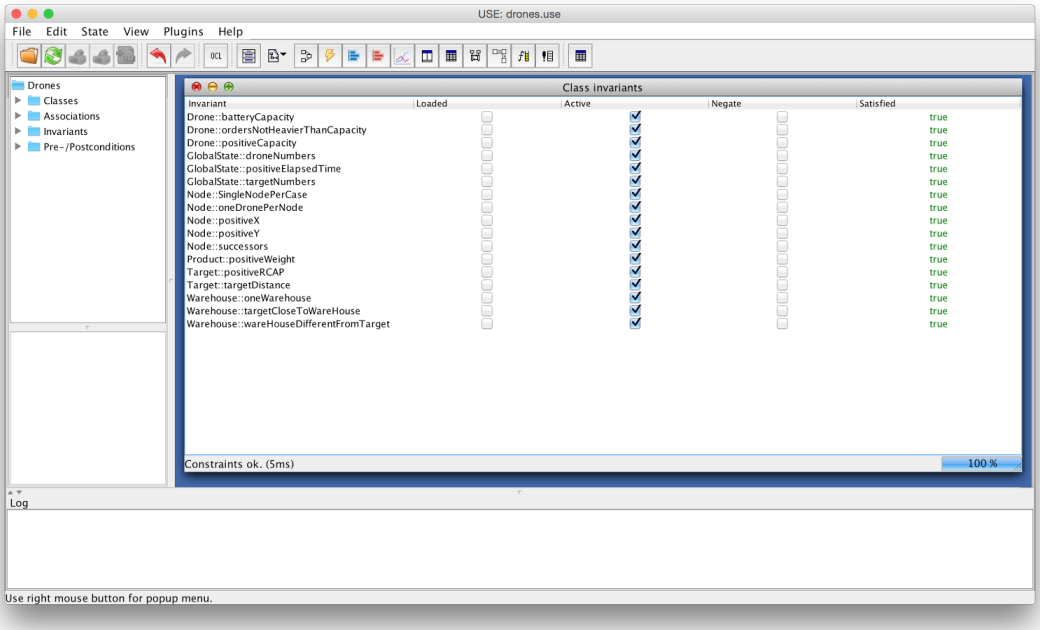
Dans l'invite de command de use : il suffit de lancer le fichier scenarios/1.simple/simple.cmd qui s'occupe de reset, et d'essayer de générer un état cohérent.

```
use> read simple.cmd
simple.cmd> reset
simple.cmd> open ../../drones.use
simple.cmd> gen start ./simple.assl init(2, 2, 1, 2)
Progress of first Try in ASSL-Procedure (2 combinations):
|-------------------------------------------------|
##########################
simple.cmd> gen result accept
Generated result (system state) accepted.
simple.cmd>
use> _
```

Un monde généré est le suivant :



Celui ci respecte les contraintes comme vu ci-dessous :



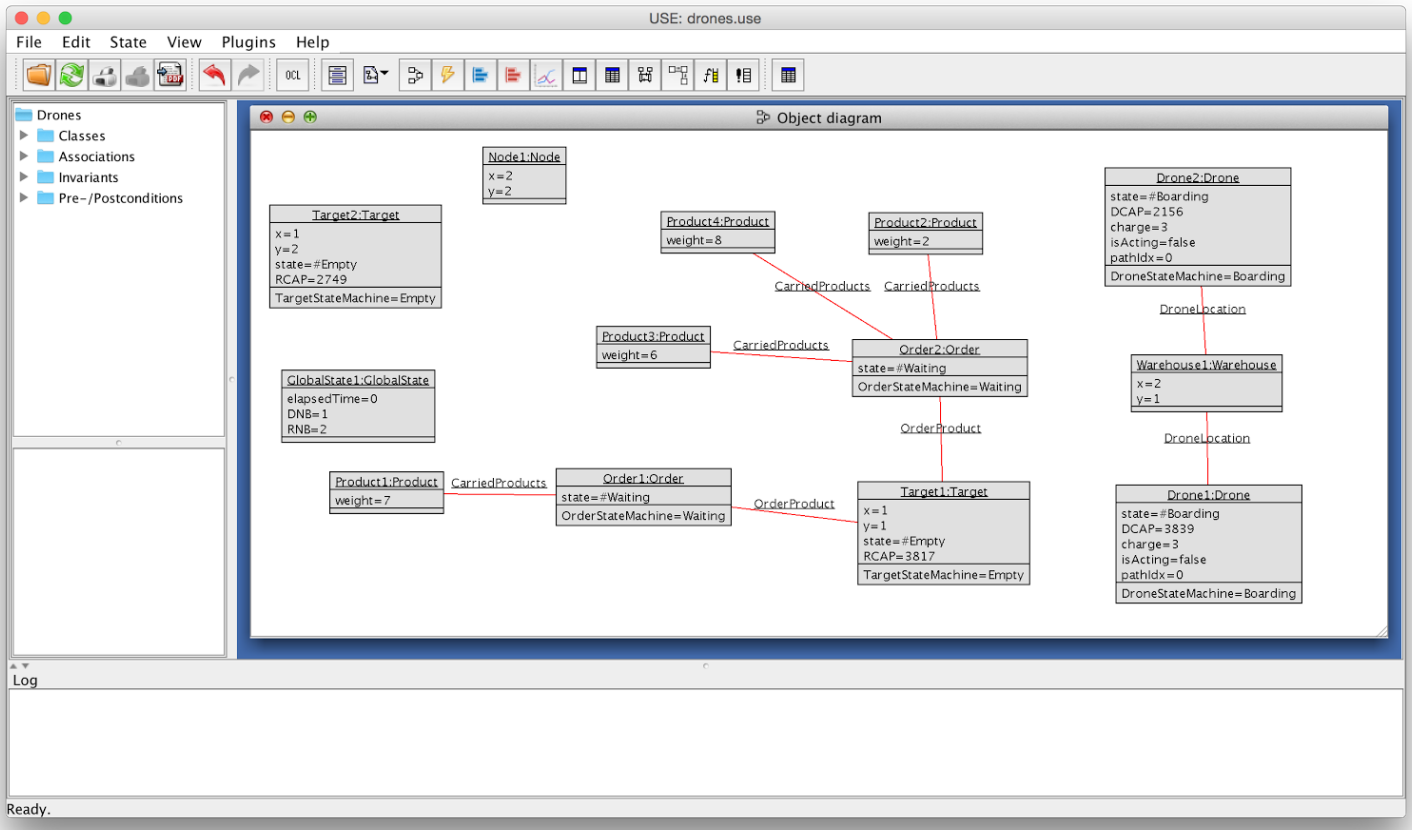
Nota : Pour une grille 3x3, la procédure met 5 minutes à générer un état cohérent contre moins de 5 secondes pour une grille 2x2.

Noyau d'invariants prouvés indépendants

Pour trouver notre noyau d'invariants prouvés indépendants , nous nions un invariant (en gardant les autres, sans les nier) et nous testons notre système. Si nous arrivons à trouver un monde qui vérifiait tous les invariants ainsi que celui que l'on a inversé, alors celui-ci fait partie du noyau d'invariants prouvés indépendants.

Nous testons l'invariant :
inv droneNumbers : Drone.allInstances()->size() = self.DNB
Nous prenons sa négation:
inv droneNumbers : not(Drone.allInstances()->size() = self.DNB)

Nous essayons de générer un monde avec la procédure simple.cmd avec le fichier simple.assl modifié pour créer DNB+1 drones au lieu de DNB drones:



Ce monde contient 2 drones, avec DNB = 1.

On a donc réussit à créer un monde, cet invariant est donc prouvé indépendant.

Nous testons l'invariant :

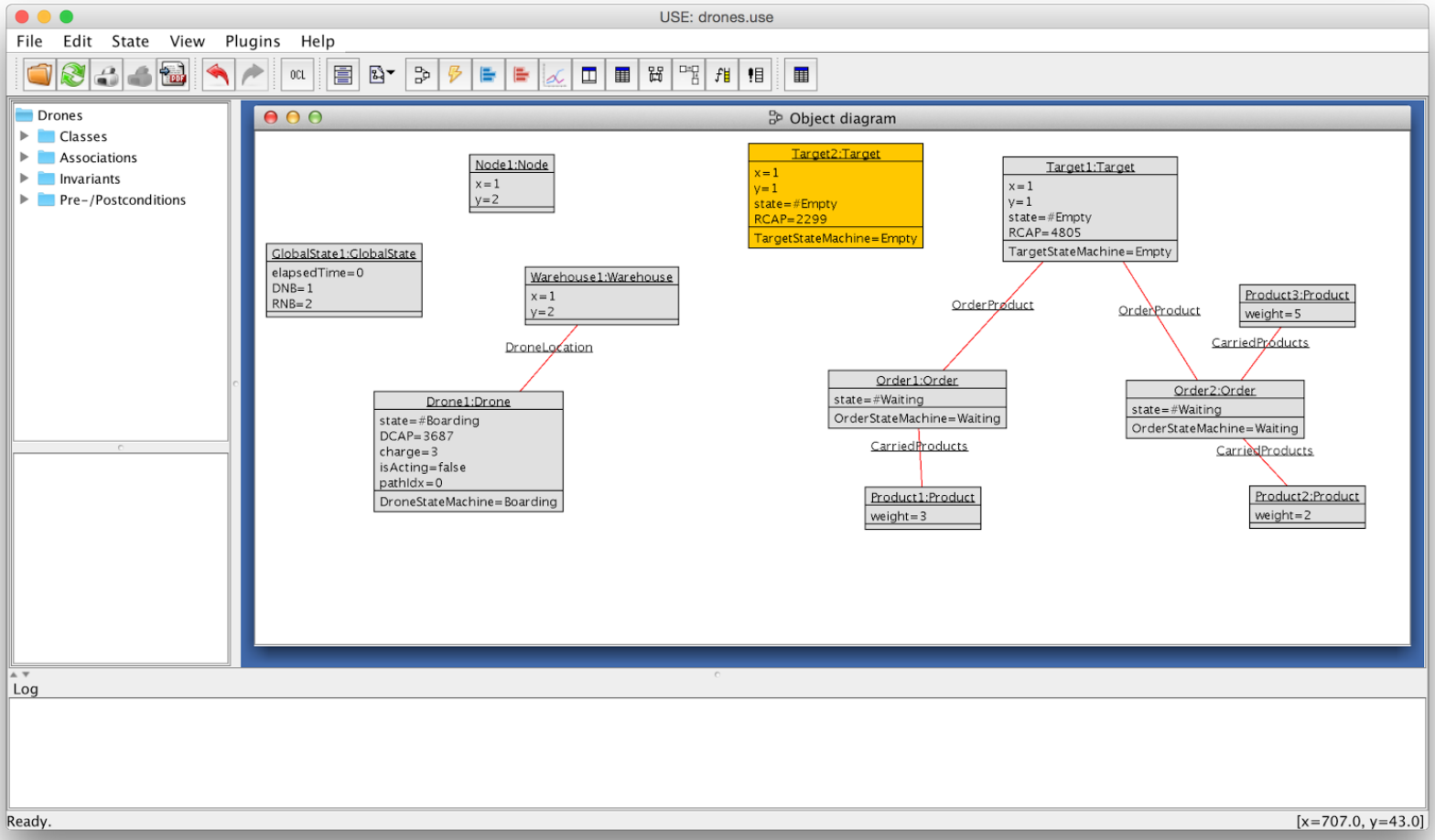
inv SingleNodePerCase : Node.allInstances()->select(n | n.x = self.x and n.y = self.y)->size() = 1

Nous prenons sa négation :

inv SingleNodePerCase : not(Node.allInstances()->select(n | n.x = self.x and n.y = self.y)->size() = 1)

Nous essayons de générer un monde avec la procédure simple.cmd (cette fois le fichier simple.assl n'a pas été modifié).

Nous réussissons à générer le monde suivant :



Notons que Target1 et Target2 possèdent les mêmes coordonnées. (qui correspond bien à l'inverse de notre invariant Single Node per case)

Cet invariant est donc aussi prouvé indépendant.

Relation des autres invariants avec ce noyau

Nous déroulons la méthode précédente pour tous les invariants, en inversant un invariant à la fois et en essayant de trouver un monde.
Nous trouvons que tous nos invariants sont indépendants. Le fait de ne pas avoir trouvé d'invariants dépendants implique que dans notre étape de construction des invariants, nous avons fait attention en essayant d'éviter tout invariant qui nous paraissait similaire et redondant.

ALGORITHME SEQUENTIEL

Version naïve :

Il est facile de trouver une première version de l'algorithme de recherche des noeuds atteignables. Une version basique de l'algorithme de recherche consiste à explorer tous les successeurs, puis les successeurs de chaque successeur et ainsi de suite.
Il semble que cette fonction puisse arriver à une fin et nous allons donc essayer de le démontrer.

Nous avons donc notre algorithme défini par :

- une pré-condition : Q : $\exists x \in V$ tel que G = (V,A)
- une post-condition : a({v}) = a(X)

Afin d'avoir un programme qui explore tous les points désirés de manière itérative il faut faire évoluer l'ensemble X à chaque itération :

$$X := X \cup s(X)$$

Notre programme doit être initialisé afin de vérifier l'invariant, soit intuitivement :

$$X := \{v\}$$

Notre programme agissant de manière itérative sur tous les points possibles, il faut donc établir une condition de sortie du programme. Notre exploration s'arrête lorsqu'il n'existe plus de nouveaux éléments explorables, soit quand tous les éléments atteignables sont inclus dans l'ensemble des atteignables. Cette condition peut se traduire par la présence des suivants de X dans l'ensemble X : les successeurs sont déjà dans l'ensemble des atteignables. On a donc comme condition de sortie du programme :

$$s(X) \subseteq X$$

Au final nous avons donc un programme de la forme :

$$\{Q : \exists x \in V \text{ tel que } G = (V,A)\}$$
$$\text{Initialisation : } X := \{v\}$$
$$\{\text{inv P0 : } a(\{v\}) = a(X)\}$$
$$*[B0 : \neg(s(X) \subseteq X) \rightarrow X := X \cup s(X)]$$
$$\{R : a(\{v\}) = X\}$$

Il nous faut maintenant montrer que la condition de départ respecte l'invariant et que l'invariant est maintenable.
P0 implique t'il la weakest pre-condition ?

$$P0 \Rightarrow wp(X := X \cup s(X);).P0$$

$$a(\{v\}) = a(X)\{X \setminus X \cup s(X)\}$$

Par substitution de X : $a(\{v\}) = a(X \cup s(X))$

Par distributivité de a sur U : $a(\{v\}) = a(X) \cup a(s(X))$

Par la théorie des ensembles : $a(\{v\}) = X \cup a(s(X))$

=> Vrai

Mais ce programme se termine-t'il?

- On suppose que le graphe est fini et qu'il existe donc un nombre fini de chemin entre deux noeuds.
- Chaque noeud n'est ajouté qu'une seule fois à l'ensemble des atteignables.
- A un certain moment du programme l'ensemble des noeuds dont les successeurs n'ont pas encore été calculé correspondra à l'union des noeuds n'ayant pas de successeurs et des noeuds dont les successeurs appartiennent déjà à X.

Donc après un certain nombre d'itérations on aura bien $s(X) \subseteq X$ => le programme se termine.

La post-condition est elle vraie après la terminaison de notre programme?
Il faut prouver que :

$$P \wedge \neg B = R$$

Soit :

$$a(\{v\}) = a(X) \wedge s(X) \subseteq X \Rightarrow a(\{v\}) = X$$

Or :

$$a(\{v\}) = X \cup a(s(X))$$

D'où :

$$a(X) = X \cup a(s(X))$$

On remplace donc :

$$a(\{v\}) = X \cup a(s(X)) \wedge S(X) \subseteq X$$

On obtient :

$$a(s(X)) \subseteq X$$

Et donc :

$$a(\{v\}) = X$$

On a donc bonne terminaison de notre programme.

Cependant ce programme est très redondant et peu efficace.

Version améliorée :

Le but de ce nouvel algorithme plus performant est de supprimer les redondances dans l'exploration des suivants. Il faut donc à chaque itération s'assurer que l'on n'a pas déjà exploré les successeurs d'un ensemble de noeuds donné.

Il faut donc définir deux nouveaux ensembles : X des noeuds déjà explorés et Y des noeuds que l'on n'a pas encore exploré.
On obtient donc :

$$a(\{v\}) = a(X \cup Y)$$

Les deux ensembles X et Y doivent être disjoints :

$$X \cap Y = \varnothing$$

Et on aura aussi :

$$s(X) \subseteq X \cup Y$$

Pour finir nous obtenons donc l'invariant :

$$P1 : a(\{v\}) = a(X \cup Y) \wedge X \cap Y = \varnothing \wedge s(X) \subseteq X \cup Y$$

L'initialisation de notre nouveau programme doit respecter l'invariant, soit :

$$X = \varnothing \; ; \; Y = \{v\} \; ;$$

Notre programme se termine lorsque tous les éléments ont été parcourus, soit une condition de sortie :

$$Y = \varnothing \; ;$$

Ainsi notre programme peut se présenter sous la forme suivante :

$$\{Q : \exists x \in V \text{ tel que } G = (V,A) \}$$
$$\text{Initialisation :} X := \varnothing \; ; \; Y := \{v\} \; ;$$
$$\{\text{inv } P1 : a(\{v\}) = a(X \cup Y) \wedge X \cap Y = \varnothing \wedge s(X) \subseteq X \cup Y\}$$
$$*[B1 : \neg(Y = \varnothing) \rightarrow X := X \cup Y \; ; \; Y := s(Y) \setminus X]$$
$$\{R : a(\{v\}) = X \cup Y\}$$

Il nous faut maintenant vérifier que l'invariant est maintenable. Respecte t'il la weakest precondition?

$$P1 \Rightarrow wp(X := X \cup Y; \; Y := s(Y) \setminus X;).P1$$

Nous allons procéder en prouvant chaque partie de l'invariant indépendement.

Première partie de l'invariant :

$$wp(X := X \cup Y; Y := s(Y)\backslash X).(a(\{v\})=a(X \cup Y))$$

$$[\, [\, a(\{v\}) = a(X \cup Y)]\, [Y\backslash s(Y)\backslash X]\,]\, [X\backslash(X \cup Y)\,]$$

Par substitution de Y :

$$[\, a(\{v\}) = a(X \cup (s(Y)\backslash Y))\,]\, [\, X\backslash(X \cup Y)]$$

Par substitution de X :

$$a(\{v\}) = a(\, (X \cup Y) \cup (s(Y)\backslash(X \cup Y)\,)$$

D'où :

$$a(\{v\}) = a(\, (X \cup Y) \cup s(Y)\,)$$

Or comme $s(Y) \subseteq Y$:

$$a(\{v\}) = a(Y \cup Z)$$

=> Vrai

Deuxième partie de l'invariant :

$$wp(X := X \cup Y; Y := s(Y)\backslash X).(X \cap Y = \varnothing)$$

$$[\, [\;\; X \cap Y = \varnothing\,]\, [\, Y\backslash(\, s(Y)\backslash X\,)\,]]\, [\, X\backslash(X \cup Y)\,]$$

Par substitution de Y :

$$[X \cap (\, s(Y)\backslash X\,) = \varnothing][\, X\backslash(X \cup Y)\,]$$

Par substitution de X :

$$(X \cup Y) \cap (\, s(Y)\backslash(X \cup Y)\,) = \varnothing$$

=> Vrai

Troisième partie de l'invariant :

$$wp(\, X := X \cup Y; Y := s(Y)\backslash X).(s(X) \subseteq X \cup Y\,)$$

$$[[\, s(X) \subseteq X \cup Y]$$

Il nous faut maintenant montrer que R est vrai après la terminaison du programme.
Soit :

$$\neg B1 \wedge P1 \Rightarrow R$$

$$\{P1 \wedge (X = \varnothing) \Rightarrow R\}$$

$$P1 : a(\{v\}) = a(X \cup Y) \wedge (X \cap Y = \varnothing) \wedge s(X) \subseteq X \cup Y$$

$$a(\{v\}) = a(X) \cup a(Y) \wedge s(X) \subseteq X \cup Y$$

Or : $Z = \varnothing \Rightarrow s(Z) = \varnothing \Rightarrow a(Z) = \varnothing$ et $Y \cap Z = \varnothing$ est toujours vrai

D'où :

$$a(\{v\}) = a(X) \wedge s(X) \subseteq X$$

Donc on a :

$$S(X) \subseteq X \Rightarrow a(X) = X$$

Et :

$$X \cup Y = x, Z = \varnothing$$

Et donc :

$$X = x$$
$$a(X) = X$$

Donc :

$$a(\{v\}) = a(X) \cup a(\varnothing) = X = XUY$$

On a donc bien un programme qui se termine.
Il faudrait désormais intégrer les éléments de la partie précédente afin de pour utiliser notre algorithme afin de calculer le plus court chemin.

ALGORITHME CONCURRENT

1 Introduction

Dans cette partie, nous dérivons les algorithmes d'élection pour permettre à des drones qui veulent accéder à la même position au pas temporel suivant de décider qui a la priorité. Le processus Pi (associé au drone i) possède la variable booléenne y.i qu'il est seul à pouvoir modifier. La seule action que fera Pi sera d'affecter une valeur à cette variable. Le problème est de synchroniser les processus de telle sorte qu'ils terminent tous et qu'ils laissent le système dans l'état vérifiant la post condition :

(1) :< #j :: y.j >= 1

2 Solution

2.1 Dérivation de l'algorithme

Nous cherchons donc à dériver cette post-condition. Pour cela, nous supposons qu'il existe une fonction B attribuant le booléen en fonction de i à y.i donc y.i = B.i et nous aimerions prouver que cela revient à dire :

<?#j :: B.j >= 1

Nous savons qu'il existe un i pour lequel B.i est vrai sinon on aurait : avec

< #j :: B.j >< 1

Mais il n'existe pas de j 6= i | B.i car dans ce cas nous aurions l'expression suivante :

< #j :: B.j >> 1

On peut donc décomposer l'expression (1) en deux expressions :

(2) :< ∃ i :: B.i >

(3) :< ∀ i, j :: B.i ⋀ B.j => i = j >

Nous utilisons les propriétés de la transitivité pour faire évoluer l'équation

(3) : i = e, j = e => i = j

Or on sait que :

B.i ☰ i = e et B.j ☰ j = e

Donc si B.i et B.j alors i = e = j =< i = j

On introduit alors cette variable e de la manière suivante :

< ∃i :: i = e >

Chaque processus va alors, au début de son programme, effectuer l'affectation e := i, pour être sur de valider l'égalité ci-dessus. On arrive alors à un programme qui sera de la forme :

e := i
y.i := (i = e)
< #?y.i ☰ i = e >

La post-condition de cette suite d'instruction

< #?y.i ☰ i = e >

doit être vérifiée localement et globalement.

La véracité locale est évidente, à la vue de la première ligne e :=i. Cependant, dès lors qu'un autre processus va, à son tour modifier la variable e, la post-condition ne sera plus vérifiée.

Il nous faut donc trouver un moyen, une weakest liberal pre-condition, pour que { ?y.i ☰ i=e } soit vraie globalement.

On se place donc dans un cas où on a deux processus distincts : { i := j }

D'après (3) : < y.i ☰ i = j > or { i := j } ce qui nous amène à :

Soit :

y.i = f alse

¬y.i

Or avec y.i ≡ i=e vraie globalement, on peut donc déduire que

Nous souhaitons transformer cette implication en utilisant une logique algébrique simple :

y.i ≡ i = e => ¬y.i

(A ≡ B => A) ≡ (A ⋁ !B)

Avec A = y.i et B = i = e

< ¬(y.i) ⋁ (i 6= e) >

En remplacant on obtient :

De manière évidente, localement après la ligne e :=i (i 6= e) est faux car i=e

on peut donc en déduire une “condition” g.i et une f.i telles que :

(4) < ∀j :: j 6= i, f.i ⋀ g.j => ¬y.i ⋁ (i 6= e) >

et grâce à cela on peut modifier l’ébauche de notre algorithme. g.i nous permet de savoir si le programme P.i a changé la valeur de e ou pas : Lorsque g.i est false, Pi a changé e.

Lors de l’éxecution d’un processus nous nous retrouvons donc face au cas suivant :

Si on reprend l’équation (4) on a une équation du type :

g.i => e := i

ou ¬g.i => y.i := (i = e)

A ⋀ B => C

Avec A = f.i , B = g.j et C = ¬y.i(oui 6= e)

qui devient :

Soit :

¬A ⋁ ¬B ⋁ C

A => ¬B ⋁ C

< f.i => ∀j :: j 6= i, ¬g.j ⋁ ¬y.i ⋁ (i 6= e) >

On a

et on peut enlever une des conditions redondante.

y.i ≡ i = edonc¬y.i ≡ i 6= e

< f.i = ∀j :: j 6= i, ¬g.j ⋁ (i 6= e) >

On récrit le programme de la manière suivante :

if(∀j :: j 6= i, ¬g.j ⋁ (i 6= e)− > skipf i

g.i = true

e := i

g.i = f alse

y.i := (i = e)

{y.i ≡ i = e}

2.2 Test de l'algorithme avec SPIN

Dans cette section, nous allons présenter comment nous avons testé l'algorithme ci dessus graçe à JSPIN.

Voici les notations que nous utiliserons dans l'implémentation de l'algorithme :

— yx symbolise y.x et gx symbolise g.x
— i symbolise le processus procI
— j symbolise le processus procJ
— k symbolise le processus procK
— on instaure une variable que l'on notera finProg qui est utilisé pour savoir si l'execution de l'ensemble du programme est terminée. Lorque finProg = nombre de processus lancées (ici 3) on vérifie l'assertion finale.
— On commence par initialiser gx à true (car aucun programme n'a encore changé e).

On implémente ensuite l'algorithme pour les 3 processus (en changeant i, j et k en fonction du processus).

```
bool gi = true,
   gj = true,
   gk = true;
bool yi, yj, yk;
int i = 0, j = 0, k = 0, e = 0, finProg = 0;

proctype ProcI() {
   i = 1;
  e = i;
  gi = false;
  if :: (!gj && !gk || i!=e)
        -> skip
  fi;
  yi = (i==e);
  finProg++;
}
```

```
proctype ProcJ() {
   j = 2;
   e = j;
   gj = false;
   if :: (!gi && !gk|| j!=e)
      -> skip
   fi;
   yj = (j==e);
   finProg++;
}

proctype ProcK() {
   k = 3;
   e = k;
   gk = false;
   if :: (!gi && !gj || k!=e)
      -> skip
   fi;
   yk = (k==e);
   finProg++;
}

init
{
   atomic
   {
      run ProcI();
      run ProcJ();
      run ProcK();
   }
   if  :: (finProg==3)
      -> skip
   fi;
   assert( (yi && !yj &&!yk) || (!yi && !yj && yk) || (!yi && yj && !yk) == true);
}
```