Intercepteur de Missile

Gnebehi Bagre

Ingénierie des logiciels

Ouriachi Khadir

I-Objectif du modèle

Produire un logiciel qui contrôle un système d'auto-défense par contre mesure qui répond aux règles de sûreté de fonctionnement:

chaque fois qu'un missile est détecté, il est toujours possible de le détecter et de le détruire.

II-Définir les besoins

R1 :Un détecteur qui active les capteurs

R2 :Des capteurs qui identifie la menace (en distance, en longitude, et en latittude)

R3 :Un contrôleur qui donne les ordres via un algorithme

R4 :Un dispenseur pour lancer des antimissiles de type A,B,C

III-Analyser le comportement fonctionnel du système

III-1 Quels sont les classes

Modèle Classique

-classe Capteur pour modéliser les capteurs

-classe CapteurDistance, CapteurLatitude ,et classe CapteurLongitude des sous classes de capteurs pour modeliser les capteurs de distance, de latitude et de longitude

-classe Controleur

-classe Environnement : l'environnement du capteur et du dispenseur

-Dispenseur: pour modéliser les Dispenseurs

-DispenseurA, DispenseurB, DispenseurC : modeliser les dispenseurs de missile A,B,C

-SystemeAM: classe qui configure le système

-Horloge : modélise l'horloge de base

-Application : pour faire intérargir les composants du système

Modèle Concurrent

-classe Capteur pour modéliser les capteurs

-classe CapteurDistance, CapteurLatitude ,et classe CapteurLongitude des sous classes de capteurs pour modeliser les capteurs de distance, de latitude et de longitude

-classe Contexte : contexte d'application du système

-classe Controleur

-classe Environnement : l'environnement du capteur et du dispenseur

-Dispenseur: pour modéliser les Dispenseurs

-DispenseurA, DispenseurB, DispenseurC : modeliser les dispenseurs de missile A,B,C

-SystemeAM: classe qui configure le système

-Horloge : modélise l'horloge de base

-Application : pour faire intérargir les composants du système

-ThreadDeBase : créer un thread générique

-TimeStamp

Modèle Temps Réel

-classe Capteur pour modéliser les capteurs

-classe CapteurDistance, CapteurLatitude ,et classe CapteurLongitude des sous classes de capteurs pour modeliser les capteurs de distance, de latitude et de longitude

-classe Controleur

-classe Environnement : l'environnement du capteur et du dispenseur

-Dispenseur: pour modéliser les Dispenseurs

-DispenseurA, DispenseurB, DispenseurC : modeliser les dispenseurs de missile A,B,C

-SystemeAM: classe qui configure le système

-Application : pour faire intérargir les composants du système

III-3 Production du modèle classique

classe Capteur

```
class Capteur
-- classe pour modéliser les capteurs
instance variables
protected ID    : nat := 0; -- identité des capteurs
protected Type  : Controleur`typeActeur := <NUL>;  --type des capteurs
protected Valeur        : nat := 0;   -- valeur mesurée par le capteur
protected Env   : Environnement := new Environnement();  -- environnement du capteur

operations
--donner l'identité d'un capteur
public GetID: () ==> nat
        GetID() ==
                return ID;
--donner le type  d'un capteur
public GetType: () ==> Controleur`typeActeur
        GetType() ==
                return Type;
--retourne la valeur mesurée par le capteur
public ReadValeur: () ==> nat
        ReadValeur() ==
                return Valeur;
--
--Fonction réalisée par un capteur: renvoyée à la responsabilité de la sous-classe en fonction du capteur utilisé
public Action: () ==> ()
        Action() ==
                is subclass responsibility
end Capteur
```

classe CapteurDistance

```
class CapteurDistance is subclass of Capteur
--modelise un capteur de distance
operations

public CapteurDistance: nat * Controleur`typeActeur * nat ==> CapteurDistance
        CapteurDistance(id, type, val) ==
                (ID := id;
                Type := type;
                Valeur := val;
                );
-- action du CapteurDistance:  "mesurer la distance de l'environnement"
public Action: () ==> ()
        Action() ==
                (Valeur := Application`env.ReadDistance();
                );

end CapteurDistance
```

classe CapteurLatitude

```
class CapteurLatitude is subclass of Capteur
--modelise un capteur de latitude
operations

public CapteurLatitude: nat * Controleur`typeActeur * nat ==> CapteurLatitude
        CapteurLatitude(id, type, val) ==
                (ID := id;
                Type := type;
                Valeur := val;
                );
-- action du CapteurLatitude:  "mesurer la latitude de l'environnement"
public Action: () ==> ()
        Action() ==
                (Valeur := Application`env.ReadLatitude();
                );

end CapteurLatitude
```

classe CapteurLongitude

```
class CapteurLongitude is subclass of Capteur
--modelise un capteur de longitude
operations

public CapteurLongitude: nat * Controleur`typeActeur * nat ==> CapteurLongitude
        CapteurLongitude(id, type, val) ==
                (ID := id;
                Type := type;
                Valeur := val;
                );
-- action du CapteurLatitude:  "mesurer la latitude de l'environnement"
public Action: () ==> ()
        Action() ==
                (Valeur := Application`env.ReadLongitude();
                );

end CapteurLongitude
```

classe Dispenseur

```
class Dispenseur
--classe pour modeliser les Dispenseurs
instance variables

protected ID    : nat := 0;   -- identité
protected Type  : Controleur`typeActeur := <NUL>;  --type Dispenseurs
protected Corr  : Controleur`typeMissile := <NUL>;   -- type de correction
protected Env   : Environnement := new Environnement();         -- environnement du regulateur

operations
--donne l'identité d'un Regulateur
public GetID: () ==> nat
        GetID() ==
                return ID;

--donne le type d'un Regulateur
public GetType: () ==> Controleur`typeActeur
        GetType() ==
                return Type;

--Fonction du regulateur: de la responsabilité de la sous-classe en fonction du type régulateur appliqué
public Action: () ==> ()
        Action() ==
                is subclass responsibility

end Dispenseur
```

classe DispenseurA

```
class DispenseurA is subclass of Dispenseur
operations

public DispenseurA: nat * Controleur`typeActeur ==> DispenseurA
        DispenseurA (id, type) ==
                (ID := id;
                Type := type;
                Corr := <NUL>
                );
--simule action du DispenseurA

        public Action: () ==> ()
        Action() ==
                (if (Corr = <A>)    then Application`env.DispA()
                );

public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <A>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() ==
                return Corr;

end DispenseurA
```

classe DispenseurB

```
class DispenseurB is subclass of Dispenseur
operations

public DispenseurB: nat * Controleur`typeActeur ==> DispenseurB
        DispenseurB (id, type) ==
                (ID := id;
                Type := type;
                Corr := <NUL>
                );
--simule action du DispenseurB

        public Action: () ==> ()
        Action() ==
                (if (Corr = <B>)    then Application`env.DispB()
                );

public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <B>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() ==
                return Corr;

end DispenseurB
```

classe DispenseurC

```
class DispenseurC is subclass of Dispenseur
operations

public DispenseurC: nat * Controleur`typeActeur ==> DispenseurC
        DispenseurC (id, type) ==
                (ID := id;
                Type := type;
                Corr := <NUL>
                );
--simule action du DispenseurC

        public Action: () ==> ()
        Action() ==
                (if (Corr = <C>)    then Application`env.DispC()
                );

public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <C>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() ==
                return Corr;

end DispenseurC
```

classe Horloge

```
class Horloge
--classe pour mod◆liser l'horloge: base de temps
instance variables
--date courante:  horloge
dateActuelle: nat := 0;

values
--p◆riode de l'horloge: 1 unit◆ de temps
periode : nat = 1;

operations
--cadencement de l'horloge
public StepTime : () ==> ()
StepTime() ==
  dateActuelle:= dateActuelle + periode;

--retourne  la date actuelle ?
public GetTime : () ==> nat
GetTime() ==
  return dateActuelle;

end Horloge
```

classe SystemeAM

```
class SystemeAM
-- classe qui configure le systeme: Pilote + 6 Dispenseurs + 3 capteurs
instance variables
--cree des instances "public static" pour permettre l'acces a partir de toutes les classes du modele
public static  Pilote   : Controleur := new Controleur(25, 90, 180);
public static CapteurDistance1  : CapteurDistance := new CapteurDistance(1, <CAPTEUR_DISTANCE>, 0);  -- un seul capteur de distance
public static CapteurLatitude1  : CapteurLatitude := new CapteurLatitude(2, <CAPTEUR_LATITUDE>, 0);  -- un seul capteur de latitude
public static CapteurLongitude1 : CapteurLongitude := new CapteurLongitude(3, <CAPTEUR_LONGITUDE>, 0);  -- un seul capteur de longitude
public static DispenseurA1      : DispenseurA := new DispenseurA(4, <DISPENSEUR_A>);
public static DispenseurA2      : DispenseurA := new DispenseurA(5, <DISPENSEUR_A>);
public static DispenseurB1      : DispenseurB := new DispenseurB(6, <DISPENSEUR_B>);
public static DispenseurB2      : DispenseurB := new DispenseurB(7, <DISPENSEUR_B>);
public static DispenseurC1      : DispenseurC := new DispenseurC(8, <DISPENSEUR_C>);
public static DispenseurC2      : DispenseurC := new DispenseurC(9, <DISPENSEUR_C>);

operations
-- Pilote gere les acteurs suivants:3 capteurs + 6 dispenseurs

public SystemeAM: () ==> SystemeAM
        SystemeAM() ==
                (
                Pilote.AjouterActeur(CapteurDistance1.GetID(), CapteurDistance1.GetType());
                Pilote.AjouterActeur(CapteurLatitude1.GetID(), CapteurLatitude1.GetType());
                Pilote.AjouterActeur(CapteurLongitude1.GetID(), CapteurLongitude1.GetType());

                Pilote.AjouterActeur(DispenseurA1.GetID(), DispenseurA1.GetType());
                Pilote.AjouterActeur(DispenseurA2.GetID(), DispenseurA2.GetType());
                Pilote.AjouterActeur(DispenseurB1.GetID(), DispenseurB1.GetType());
                Pilote.AjouterActeur(DispenseurB2.GetID(), DispenseurB2.GetType());
                Pilote.AjouterActeur(DispenseurC1.GetID(), DispenseurC1.GetType());
                Pilote.AjouterActeur(DispenseurC2.GetID(), DispenseurC2.GetType());
                );
end SystemeAM
```

Les classes Controleur et Environnement sont disponible dans le dossier fourni.

III-4 Production du modèle Concurrent

Les classes Capteur ,Dispenseur et Horloge sont les les mêmes que celles dans le modèle classique.

classe CapteurDistance

```
class CapteurDistance is subclass of Capteur, ThreadDeBase

instance variables
fini : bool := false
--modelise un capteur de distance
operations

public CapteurDistance: nat * Controleur`typeActeur * nat * Contexte * nat1 * bool ==> CapteurDistance
        CapteurDistance(id, type, val, envir, p, isP) ==
                (ID := id;
                Type := type;
                Valeur := val;
                Env := envir;
                periode := p;
                estPeriodique := isP;
                );

public Finir: () ==> ()
        Finir() ==
  fini := true;

public estFini: () ==> ()
estFini() ==
  skip;

-- action du CapteurDistance:  "mesurer la distance de l'environnement"
public Action: () ==> ()
        Action() ==
                (Valeur := Application`env.ReadDistance();
                )

sync
  per estFini => fini;


end CapteurDistance
```

CapteurLatitude

```
class CapteurLatitude is subclass of Capteur, ThreadDeBase
--modelise un capteur de latitude
instance variables

fini : bool := false;
operations

public CapteurLatitude: nat * Controleur`typeActeur * nat * Contexte * nat1 * bool ==> CapteurLatitude
        CapteurLatitude(id, type, val, envir, p, estP) ==
                (ID := id;
                Type := type;
                Valeur := val;
                Env := envir;
                periode := p;
                estPeriodique := estP;
                );
public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
estFini() ==
  skip;

public Action: () ==> ()
Action() ==
  (Valeur := Env.ReadLatitude();
  )

sync
  per estFini => fini;

end CapteurLatitude
```

classe CapteurLongitude

```
class CapteurLongitude is subclass of Capteur, ThreadDeBase
--modelise un capteur de latitude
instance variables

fini : bool := false;
operations

public CapteurLongitude: nat * Controleur`typeActeur * nat * Contexte * nat1 * bool ==> CapteurLongitude
        CapteurLongitude(id, type, val, envir, p, estP) ==
                (ID := id;
                Type := type;
                Valeur := val;
                Env := envir;
                periode := p;
                estPeriodique := estP;
                );
public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
estFini() ==
  skip;

public Action: () ==> ()
Action() ==
  (Valeur := Env.ReadLongitude();
  )

sync
  per estFini => fini;

end CapteurLongitude
```

classe Contexte

```
class Contexte

instance variables
-- variables environnement
private envDistance      : nat;
private envLatitude : nat;
private envLongitude : nat;

operations
-- fixe une valeur initiale
public Contexte: int * int * int ==> Contexte
Contexte(distance0, latitude0, longitude0) ==
 (envDistance := distance0;
  envLatitude := latitude0;
  envLongitude := longitude0;
 );

--actions de base de asservissement
public SetDistance: nat ==> ()
        SetDistance(t) ==
  envDistance := t;

public SetLatitude: nat ==> ()
        SetLatitude(p) ==
  envLatitude:= p;

public SetLongitude: nat ==> ()
        SetLongitude(r) ==
  envLongitude:= r;

public ReadDistance: () ==> nat
        ReadDistance() ==
  return envDistance;

public ReadLatitude: () ==> nat
ReadLatitude() ==
  return envLatitude;

public ReadLongitude: () ==> nat
ReadLongitude() ==
  return envLongitude;

sync
  mutex(SetDistance);
  mutex(ReadDistance, SetDistance);
  mutex(SetLatitude);
  mutex(ReadLatitude,SetLatitude);
  mutex(SetLongitude);
  mutex(ReadLongitude,SetLongitude);

end Contexte
```

DispenseurA

```
class DispenseurA is subclass of Dispenseur, ThreadDeBase

instance variables
fini : bool := false;

operations

public DispenseurA: nat * Controleur`typeActeur * Contexte * nat1 * bool ==> DispenseurA
        DispenseurA (id, type, envir, p, isP) ==
                (ID := id;
                Type := type;
                Corr := <NUL>;
                Env := envir;
                periode := p;
                estPeriodique := isP;
                );

                public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
estFini() ==
  skip;

--simule action du DispenseurA
        public Action: () ==> ()
        Action() ==
                (if (Corr = <A>)    then Application`env.DispA();
                Corr := <NUL>; --RA0 de la variable--
                );

public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <A>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() ==
                return Corr;

sync
        per estFini => fini;
end DispenseurA
```

DispenseurB

```
class DispenseurB is subclass of Dispenseur, ThreadDeBase

instance variables
fini : bool := false;

operations

public DispenseurB: nat * Controleur`typeActeur * Contexte * nat1 * bool ==> DispenseurB
        DispenseurB(id, type, envir, p, isP) ==
                (ID := id;
                Type := type;
                Corr := <NUL>;
                Env := envir;
                periode := p;
                estPeriodique := isP;
                );

                public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
estFini() ==
  skip;

--simule action du DispenseurB
        public Action: () ==> ()
        Action() ==
                (if (Corr = <B>)    then Application`env.DispB();
                Corr := <NUL>; --RA0 de la variable--
                );

public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <B>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() ==
                return Corr;

sync
        per estFini => fini;
end DispenseurB
```

DispenseurC

```
class DispenseurC is subclass of Dispenseur, ThreadDeBase

instance variables
fini : bool := false;

operations

public DispenseurC: nat * Controleur`typeActeur * Contexte * nat1 * bool ==> DispenseurC
        DispenseurC(id, type, envir, p, isP) ==
                (ID := id;
                Type := type;
                Corr := <NUL>;
                Env := envir;
                periode := p;
                estPeriodique := isP;
                );

                public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
estFini() ==
  skip;

--simule action du DispenseurC
        public Action: () ==> ()
        Action() ==
                (if (Corr = <C>)    then Application`env.DispC();
                Corr := <NUL>; --RA0 de la variable--
                );

public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <C>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() ==
                return Corr;

sync
        per estFini => fini;
end DispenseurC
```

Application

```
class Application
--cette classe qui fait interagir les diferents composants du système: elle lance la simulation

instance variables
--crée un environnement: "env" qui produit des stimuli selon le  scenario décrit dans le fichier "scenario.txt"
static public env : Environnement := new Environnement("scenarioApplication.txt",1,true);

--crée une instance de la base de temps(horloge): horloge= temp de référence
static public horloge : Horloge := new Horloge();
operations

public Simuler: () ==> ()
Simuler() ==
 (-- activer environment pour créer un stimuli
  --env.stimuli();--
  IO`print("\n Simulation terminée  à l'instant: ");
  IO`print(horloge.GetTime());
 );

end Application
```

ThreadDeBase

```
class ThreadDeBase
        -- creer thread generique
instance variables
--p◆riode du thread
protected periode : nat1 := 1;
protected estPeriodique : bool := true;


operations

protected ThreadDeBase : () ==> ThreadDeBase
ThreadDeBase() ==
 (Animation`horloge.EnregisterThread(self);
  if(not Animation`horloge.EstInitialise())
  then start(self);
 );

protected Action : () ==> ()
Action() ==
  is subclass responsibility

thread
 (if estPeriodique
  then (while true
        do
         (Action();
         Animation`horloge.WaitRelative(periode);
         )
       )
  else (Action();
       Animation`horloge.WaitRelative(0);
       Animation`horloge.DesenregistrerThread();
       )
 );

end ThreadDeBase
```

Les classes Controleur,TimeStamp ,Environnement,SystemeAM sont consultable dans le dossier founi.

III-5 Production du modèle Temps Réel

Les classes Capteur ,Dispenseur sont les les mêmes que celles dans le modèle classique.

CapteurDistance

```
class CapteurDistance is subclass of Capteur

instance variables
fini : bool := false
--modelise un capteur de distance
operations

public CapteurDistance: nat * Controleur`typeActeur * nat ==> CapteurDistance
        CapteurDistance(id, type, val) ==
                (ID := id;
                Type := type;
                Valeur := val;
                );

public Action: () ==> ()
Action () ==
  Valeur := Application`env.ReadDistance() ;

public estFini: () ==> ()
estFini() ==
  skip;

sync
  --mutex(Action);       -- inutile!
  per estFini => fini;

-- Gestion de la partie thread (necessite VDM-RT)--
thread
-- periode du thread (periode, jitter, delay, offset)
periodic(1000E6,0,0,0) (Action)
end CapteurDistance
```

CapteurLatitude

```
class CapteurLatitude is subclass of Capteur

instance variables
fini : bool := false
--modelise un capteur de latitude
operations

public CapteurLatitude: nat * Controleur`typeActeur * nat ==> CapteurLatitude
        CapteurLatitude(id, type, val) ==
                (ID := id;
                Type := type;
                Valeur := val;
                );

public Action: () ==> ()
Action () ==
  Valeur := Application`env.ReadLatitude() ;

public estFini: () ==> ()
estFini() ==
  skip;

sync
  --mutex(Action);       -- inutile!
  per estFini => fini;

-- Gestion de la partie thread (necessite VDM-RT)--
thread
-- periode du thread (periode, jitter, delay, offset)
periodic(1000E6,0,0,0) (Action)
end CapteurLatitude
```

CapteurLongitude

```
class CapteurLongitude is subclass of Capteur

instance variables
fini : bool := false
--modelise un capteur de longitude
operations

public CapteurLongitude: nat * Controleur`typeActeur * nat ==> CapteurLongitude
        CapteurLongitude(id, type, val) ==
                (ID := id;
                Type := type;
                Valeur := val;
                );

public Action: () ==> ()
Action () ==
  Valeur := Application`env.ReadLongitude() ;

public estFini: () ==> ()
estFini() ==
  skip;

sync
  --mutex(Action);       -- inutile!
  per estFini => fini;

-- Gestion de la partie thread (necessite VDM-RT)--
thread
-- periode du thread (periode, jitter, delay, offset)
periodic(1000E6,0,0,0) (Action)
end CapteurLongitude
```

DispenseurA

```
class DispenseurA is subclass of Dispenseur

instance variables
fini : bool := false;

operations

public DispenseurA: nat * Controleur`typeActeur ==> DispenseurA
        DispenseurA (id, type) ==
                (ID := id;
                Type := type;
                Corr := <NUL>;
                );

                public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
--mutex(Action);
 estFini() ==
  skip;

--simule action du DispenseurA
        public Action: () ==> ()
        Action() ==
                (if (Corr = <A>)    then Application`env.DispA();
                Corr := <NUL>; --RA0 de la variable--
                );

--async
public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <A>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() == return Corr;

sync
  --mutex(Action);
  per estFini=> fini;
  mutex(SetCorrection, GetCorrection);

thread
-- periode du  thread (periode, jitter, delay, offset)
periodic(1000E6,0,0,0) (Action)

end DispenseurA
```

DispenseurB

```
class DispenseurB is subclass of Dispenseur

instance variables
fini : bool := false;

operations

public DispenseurB: nat * Controleur`typeActeur ==> DispenseurB
        DispenseurB(id, type) ==
                (ID := id;
                Type := type;
                Corr := <NUL>;
                );

                public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
--mutex(Action);
 estFini() ==
  skip;

--simule action du DispenseurB
        public Action: () ==> ()
        Action() ==
                (if (Corr = <B>)    then Application`env.DispB();
                Corr := <NUL>; --RA0 de la variable--
                );

--async
public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <B>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() == return Corr;

sync
  --mutex(Action);
  per estFini=> fini;
  mutex(SetCorrection, GetCorrection);

thread
-- periode du  thread (periode, jitter, delay, offset)
periodic(1000E6,0,0,0) (Action)

end DispenseurB
```

DispenseurC

```
class DispenseurC is subclass of Dispenseur

instance variables
fini : bool := false;

operations

public DispenseurC: nat * Controleur`typeActeur ==> DispenseurC
        DispenseurC(id, type) ==
                (ID := id;
                Type := type;
                Corr := <NUL>;
                );

                public Finir: () ==> ()
Finir() ==
  fini := true;

public estFini: () ==> ()
--mutex(Action);
 estFini() ==
  skip;

--simule action du DispenseurC
        public Action: () ==> ()
        Action() ==
                (if (Corr = <C>)    then Application`env.DispC();
                Corr := <NUL>; --RA0 de la variable--
                );

--async
public SetCorrection: Controleur`typeMissile ==> ()
        SetCorrection(cor) ==
                Corr := cor
pre (cor = <C>) or (cor = <NUL>);

public GetCorrection: () ==> Controleur`typeMissile
        GetCorrection() == return Corr;

sync
  --mutex(Action);
  per estFini=> fini;
  mutex(SetCorrection, GetCorrection);

thread

-- periode du  thread (periode, jitter, delay, offset)
periodic(1000E6,0,0,0) (Action)

end DispenseurC
```

Les classes Controleur,Environnement,SystemeAM et Application sont consultable dans le dossier founi.