Rapport de projet Dumber De Stijl

version 15 avril 2021

—

Benjamin VENDEVILLE

Lien github : `https://github.com/BenjaminVendeville/Temps-reel/tree/main/dumber`

Table des matières

1 Conception

Dans cette partie se trouvent un diagramme AADL et les diagrammes d'activités correspondant au cahier des charges. La conception est décomposée en trois parties :

— communication entre le robot et le superviseur
— traitement de la vidéo
— contrôle du robot.

1.1 Diagramme fonctionnel général

Le diagramme fonctionnel est ci-dessous. Il est aussi disponible à l'adresse `https://drive.google.com/file/d/1la7dGryRr9Iu-vNiBqZ4GGKRrwlCuQV9/view?usp=sharing`.

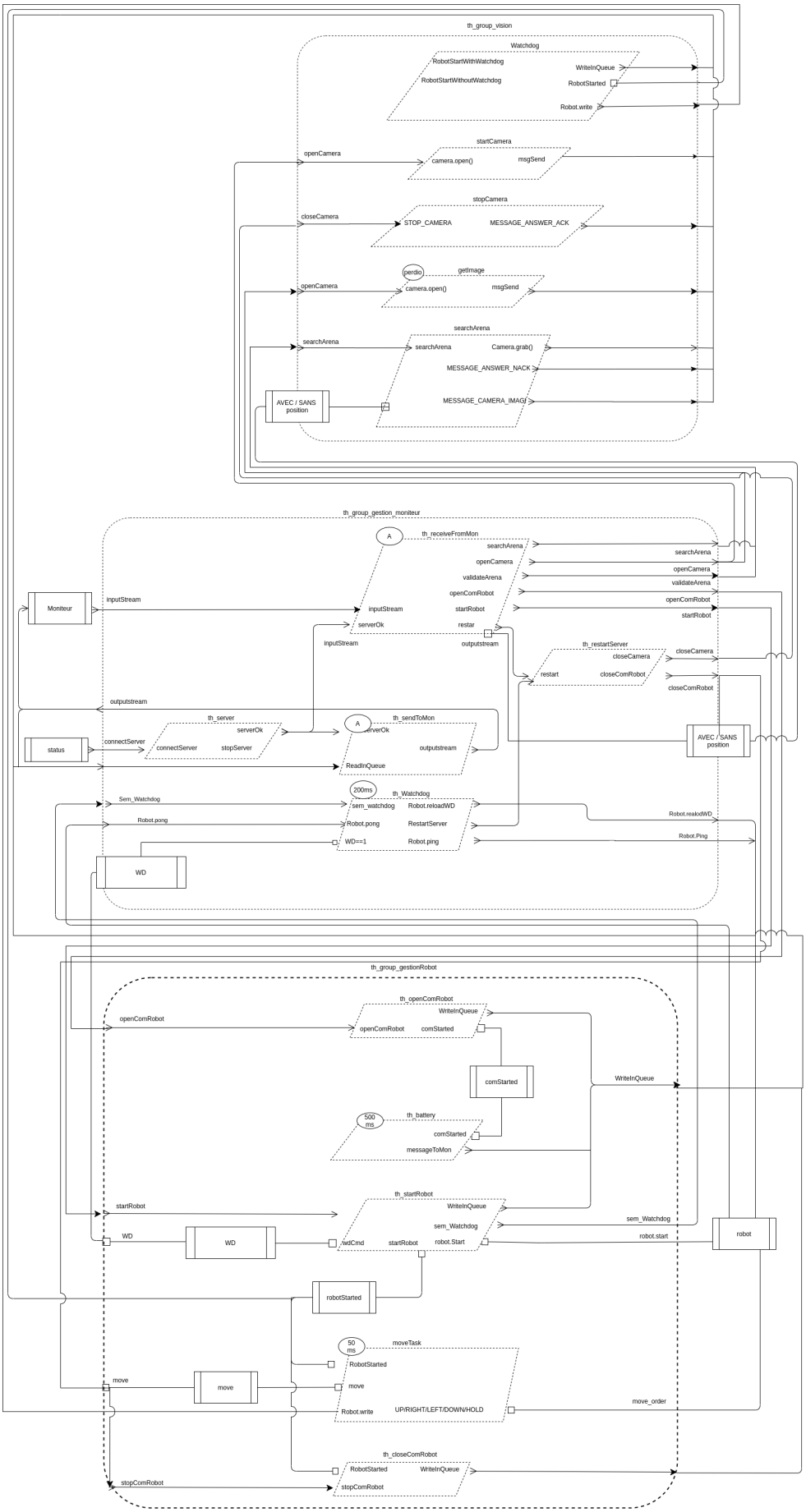

Fig. 1: Diagramme fonctionnel du système en AADL

1.2 Groupe de threads gestion du moniteur

Le groupe de thread de gestion moniteur est composé les threads suivants :

— **th_receiveFromMon** : permet de gérer la réception des instructions sous forme de message venant du moniteur ;
— th_server : lance le serveur ;
— th_restartServer : ferme les communications, kill les différents threads et relance le tout le programme et le serveur en cas de besoin.

1.2.1 Diagramme fonctionnel du groupe gestion du moniteur

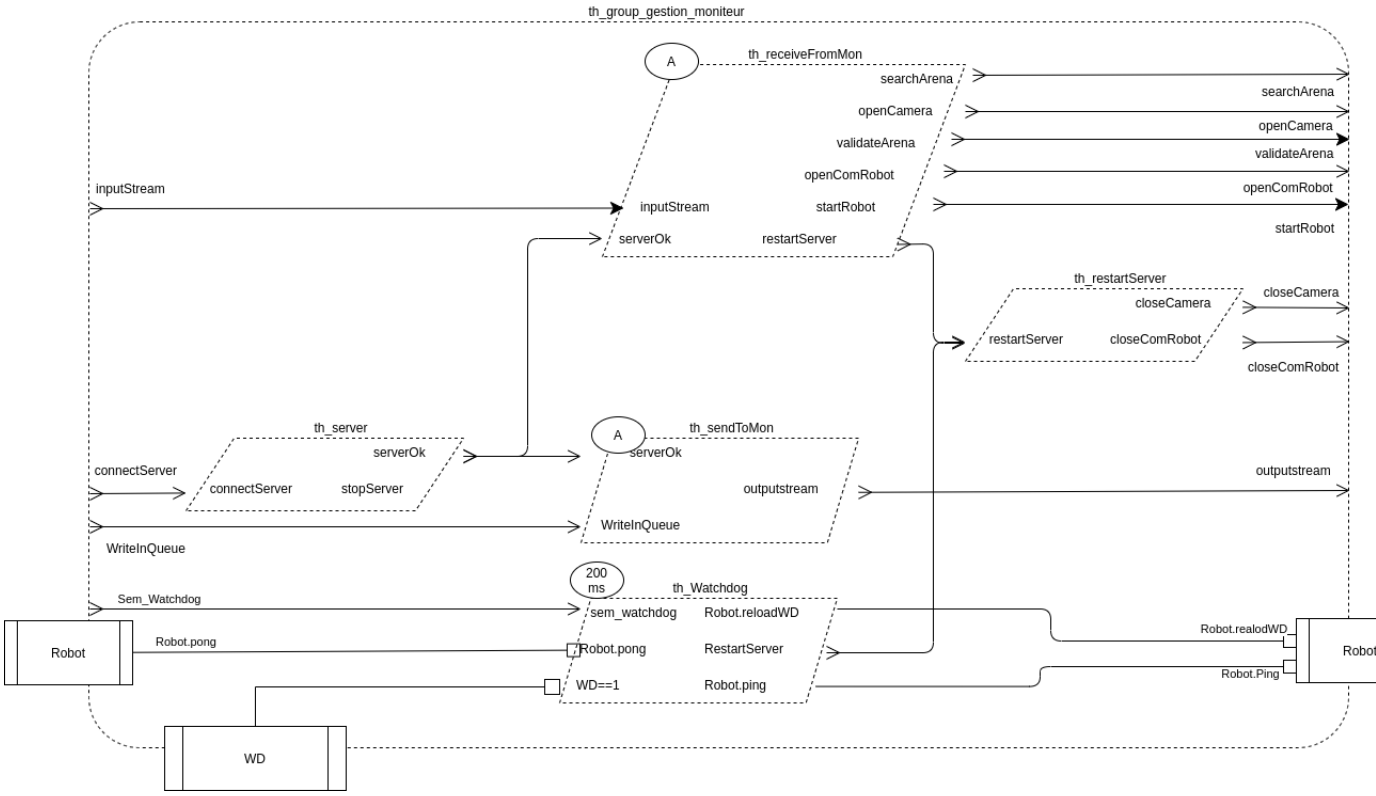

Fig. 2: Diagramme fonctionnel en AADL du groupe de threads gestion du moniteur

1.2.2 Diagrammes d'activité du groupe gestion du moniteur

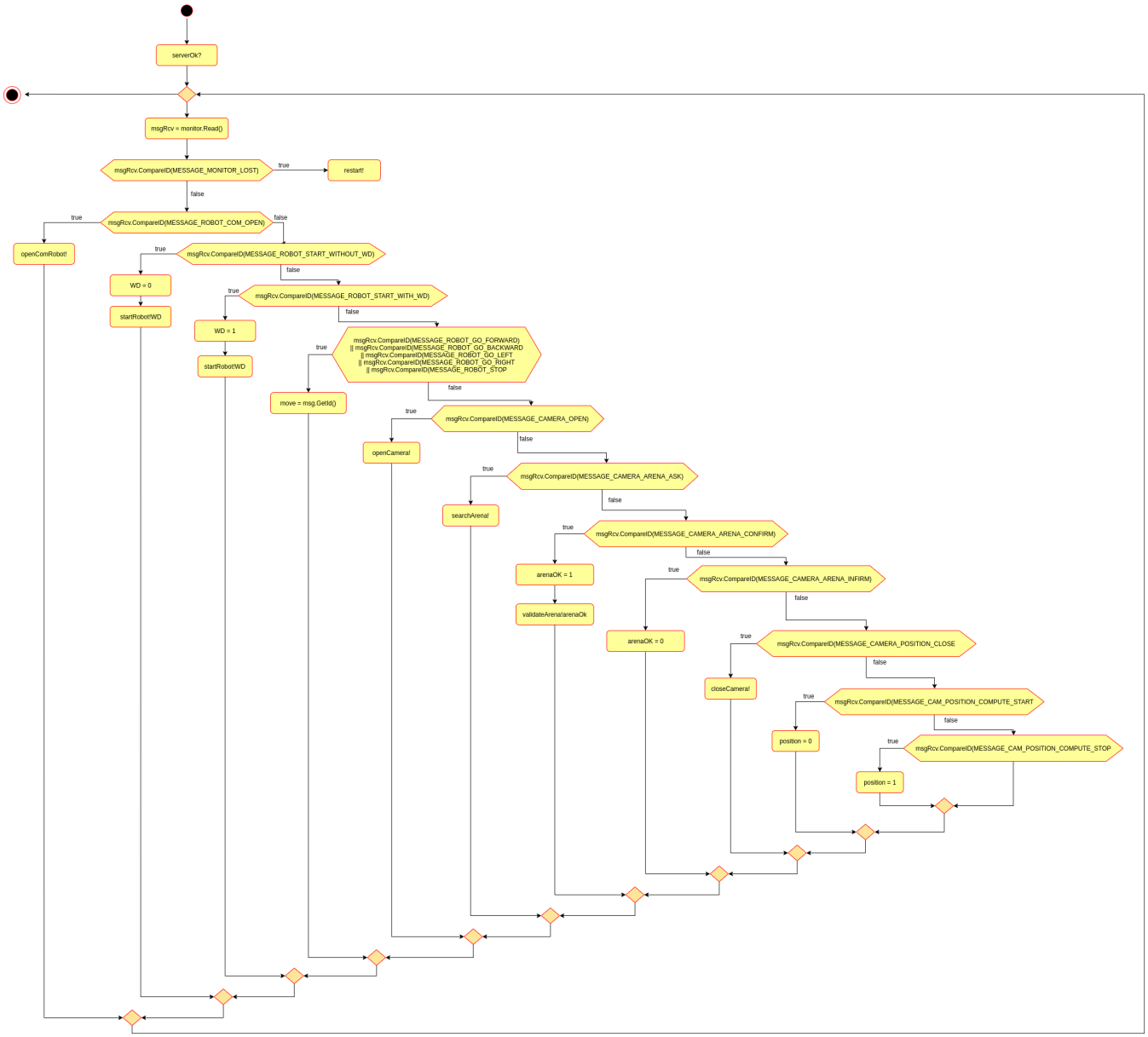

Fig. 3: Diagramme d'activité du thread `th_receiveFromMon`

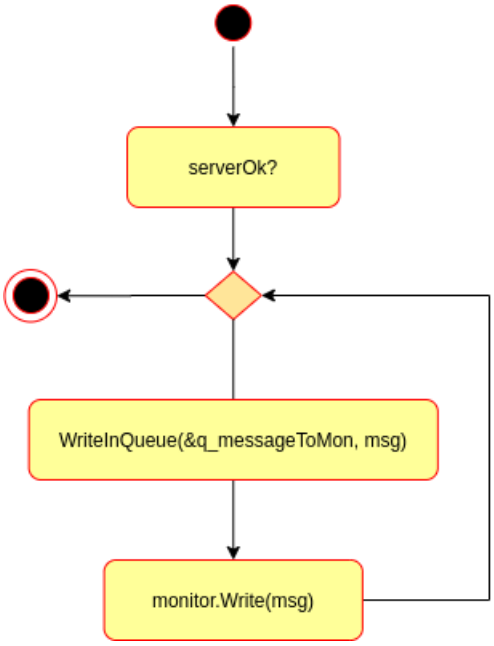

Fig. 4: Diagramme d'activité du thread `th_sendToMon`

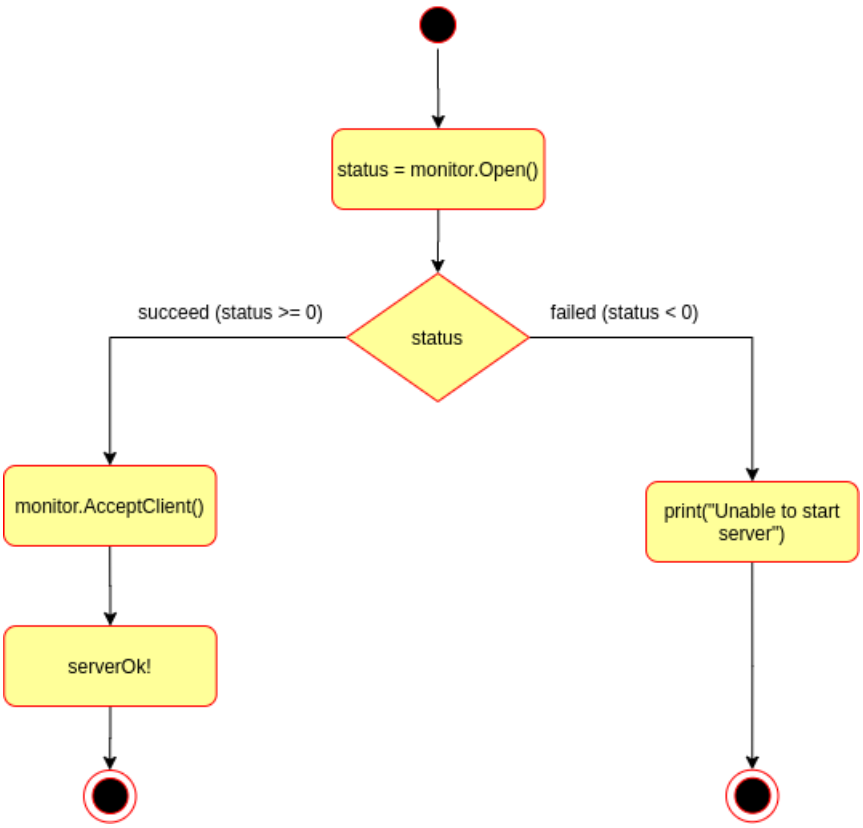

Fig. 5: Diagramme d'activité du thread `th_server`

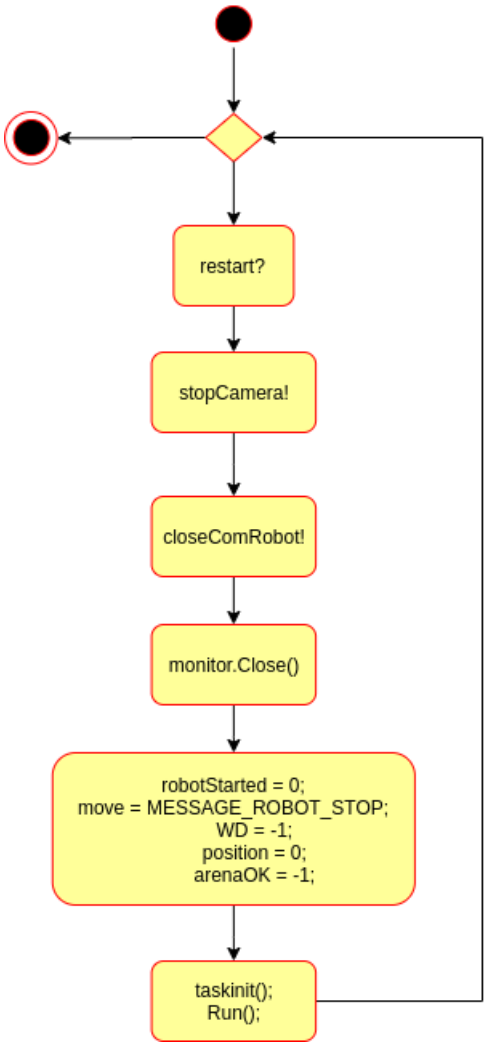

Fig. 6: Diagramme d'activité du thread `th_restartServer`

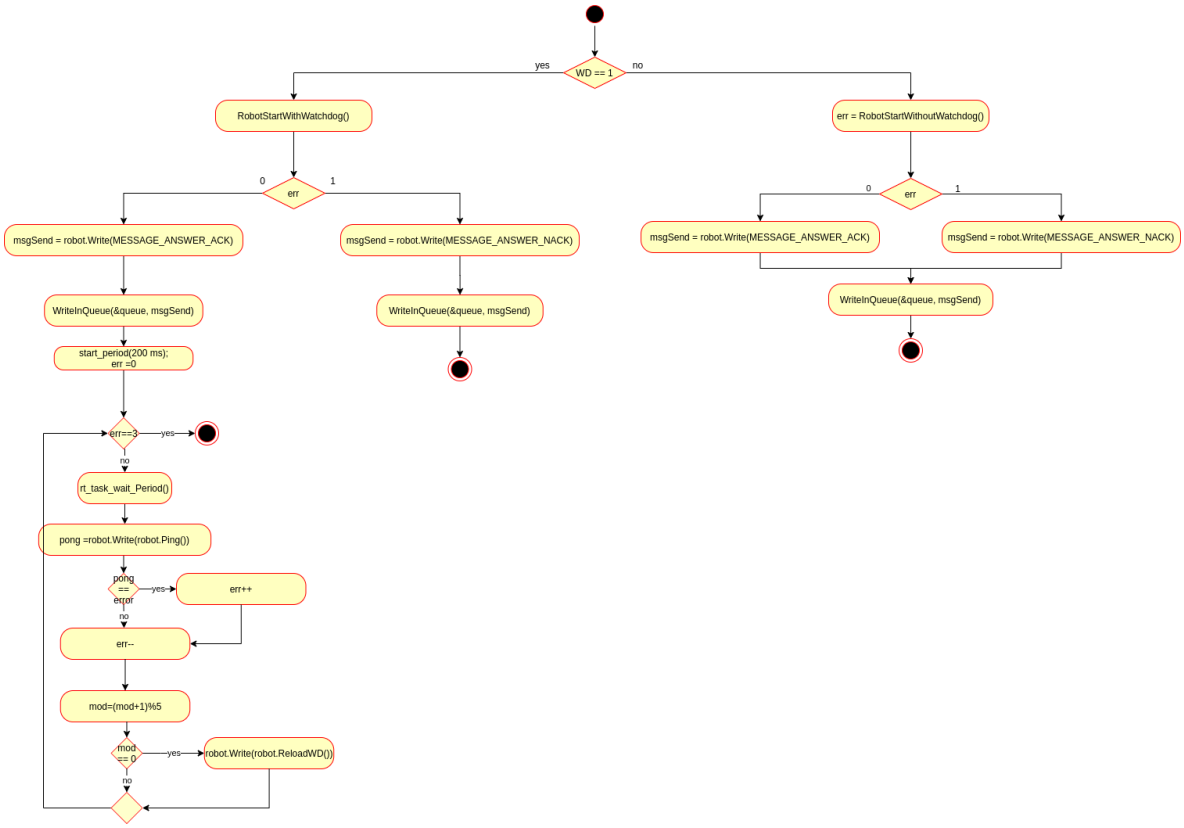

Fig. 7: Diagramme d'activité du thread `th_watchdog`

1.3 Groupe de threads gestion du robot

La gestion du robot est réalisée par plusieurs threads dont :
— **th_openComRobot** : gère de l'ouverture de la communication avec le robot ;
— **th_battery** : actualise le niveau de batterie de manière périodique pour en informer l'utilisateur ;
— **th_startRobot** : démarre le robot avec ou sans watchdog. Il a aussi pour rôle de prévenir tout les thread qui en ont besoin que le robot a démarré ;
— **th_moveTask** : gère les déplacements du robot ;
— **th_closeComRobot** : gère la fermeture de la communication avec le robot.

1.3.1 Diagramme fonctionnel du groupe gestion robot en AADL

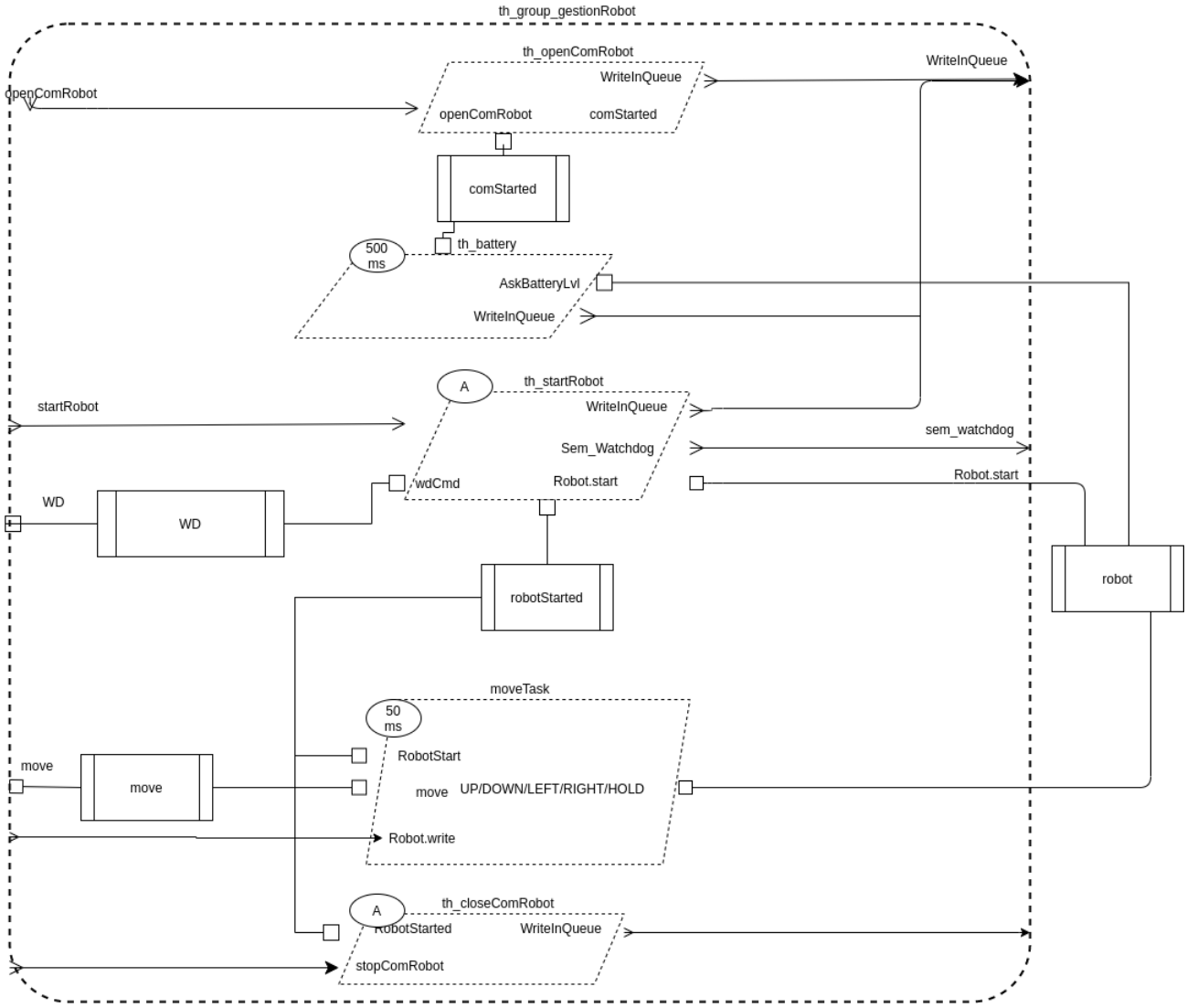

Fig. 8: Diagramme fonctionnel du groupe gestion robot

1.3.2 Diagrammes d'activité du groupe robot

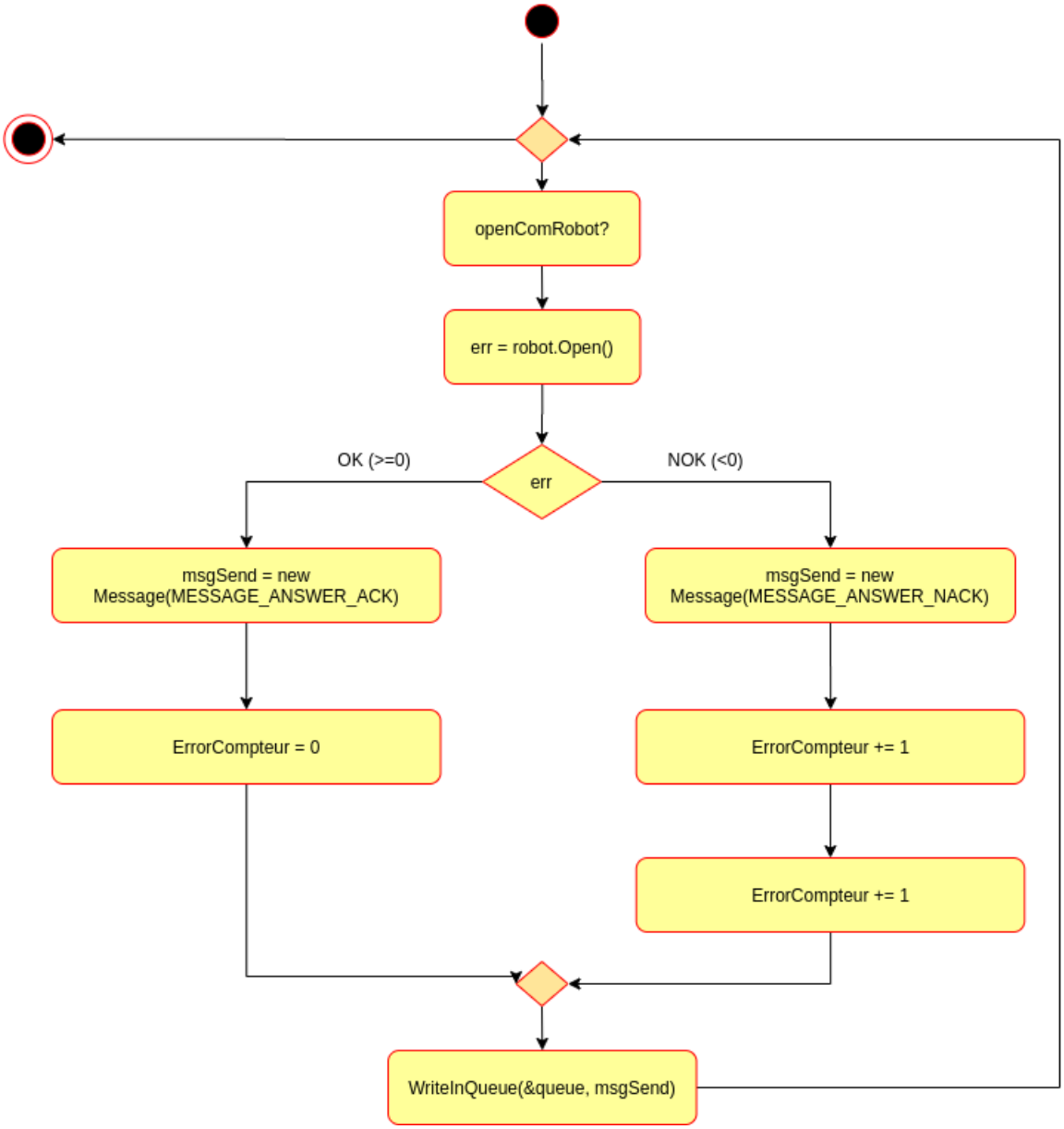

Fig. 9: Diagramme d'activité du thread `th_openComRobot`

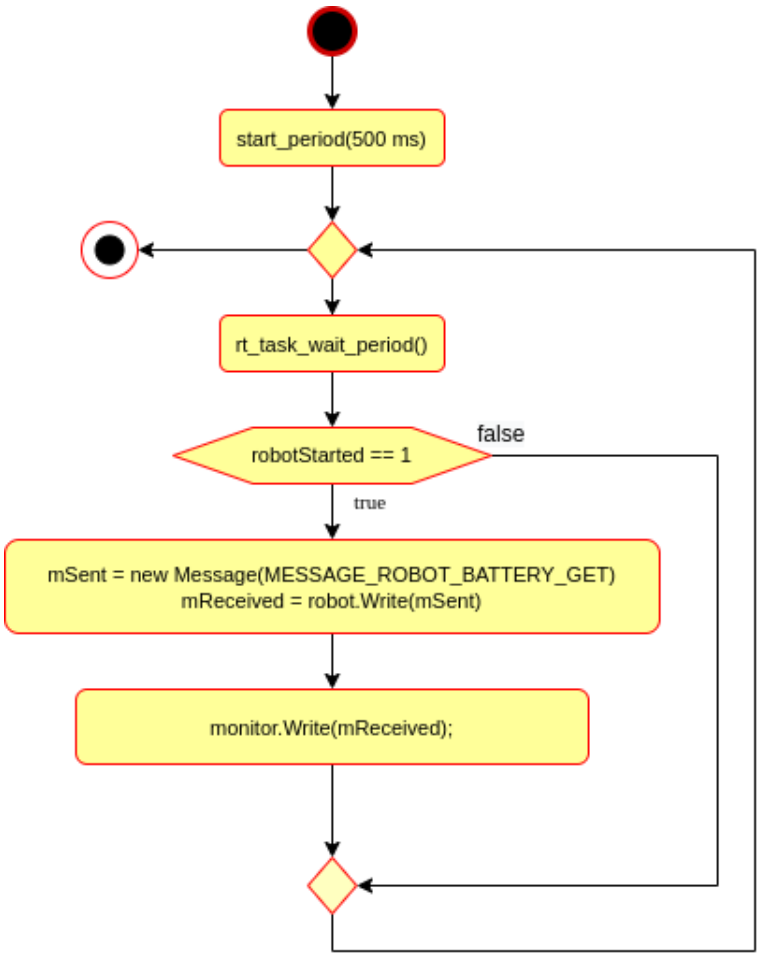

Fig. 10: Diagramme d'activité du thread `th_battery`

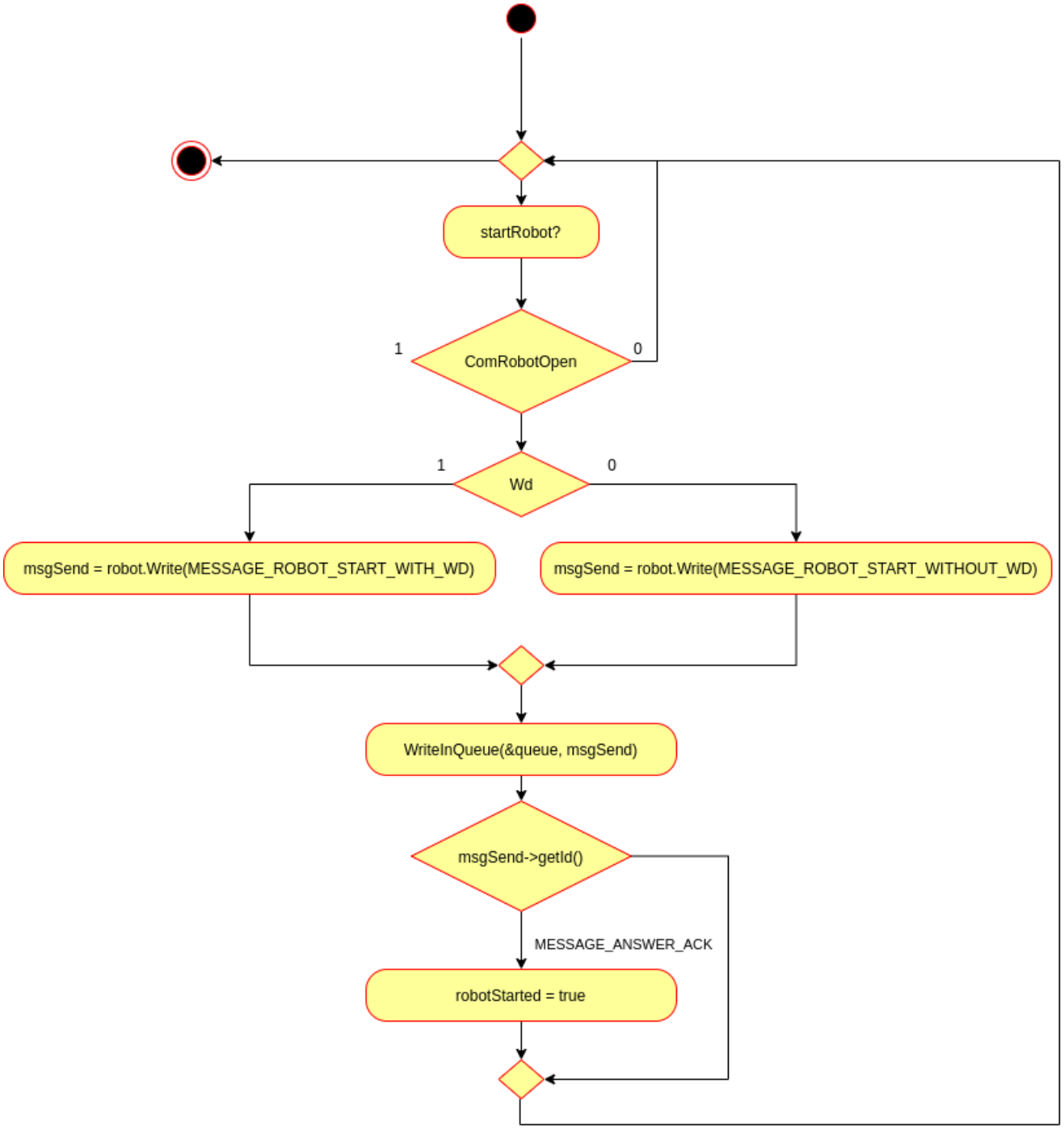

Fig. 11: Diagramme d'activité du thread `th_startRobot`

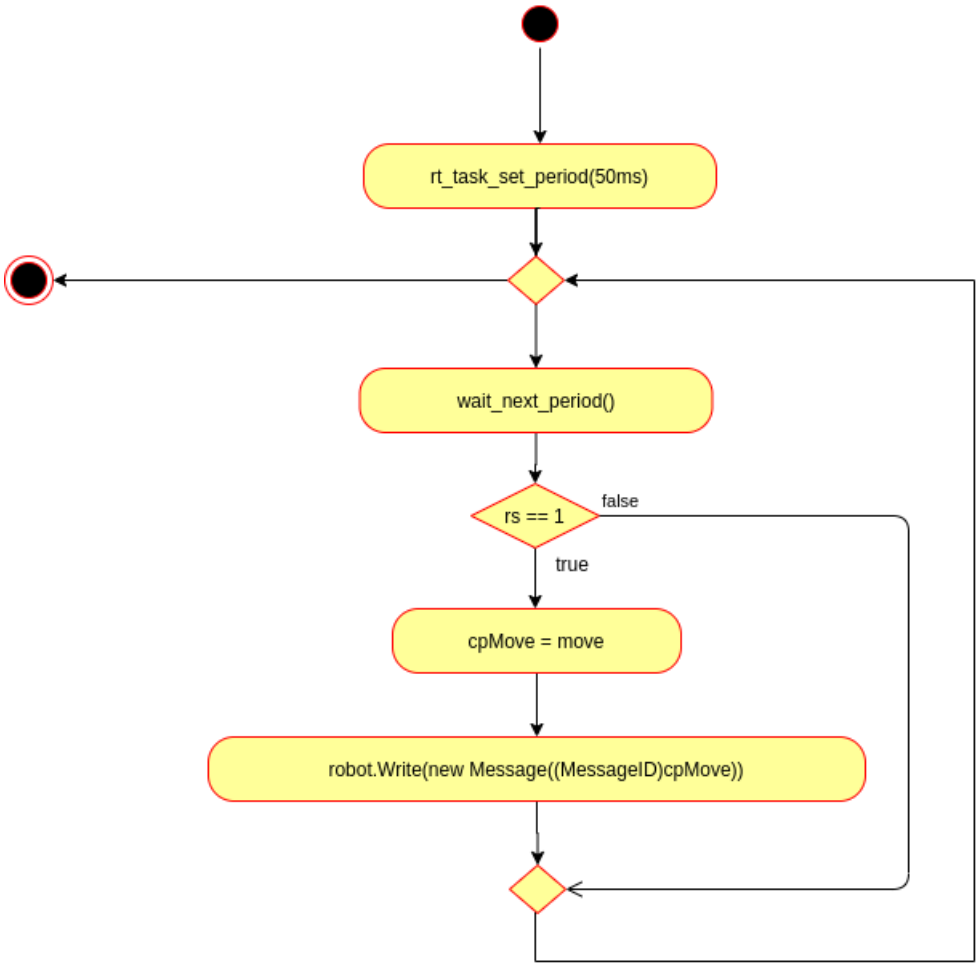

Fig. 12: Diagramme d'activité du thread `th_moveTask`

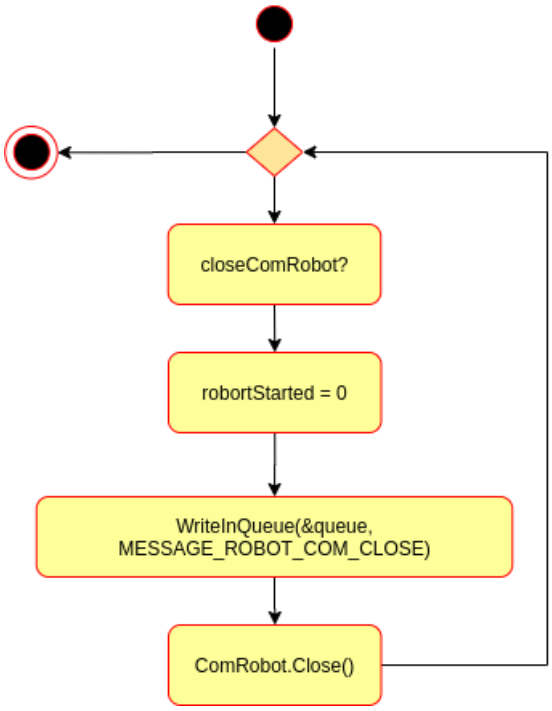

Fig. 13: Diagramme d'activité du thread `th_closeComRobot`

1.4 Groupe de threads vision

La gestion de la caméra est réalisée par les threads :
— **th_startCamera** : démarre la caméra ;
— **th_stopCamera** : permet d'éteindre la caméra ;
— **th_getImage** : gère l'acquisition périodique des images ;
— **th_searchArena** : gère la recherche d'arène pour le robot.

Remarque : la gestion de la caméra n'a pas été implémentée.

1.4.1 Diagramme fonctionnel du groupe vision

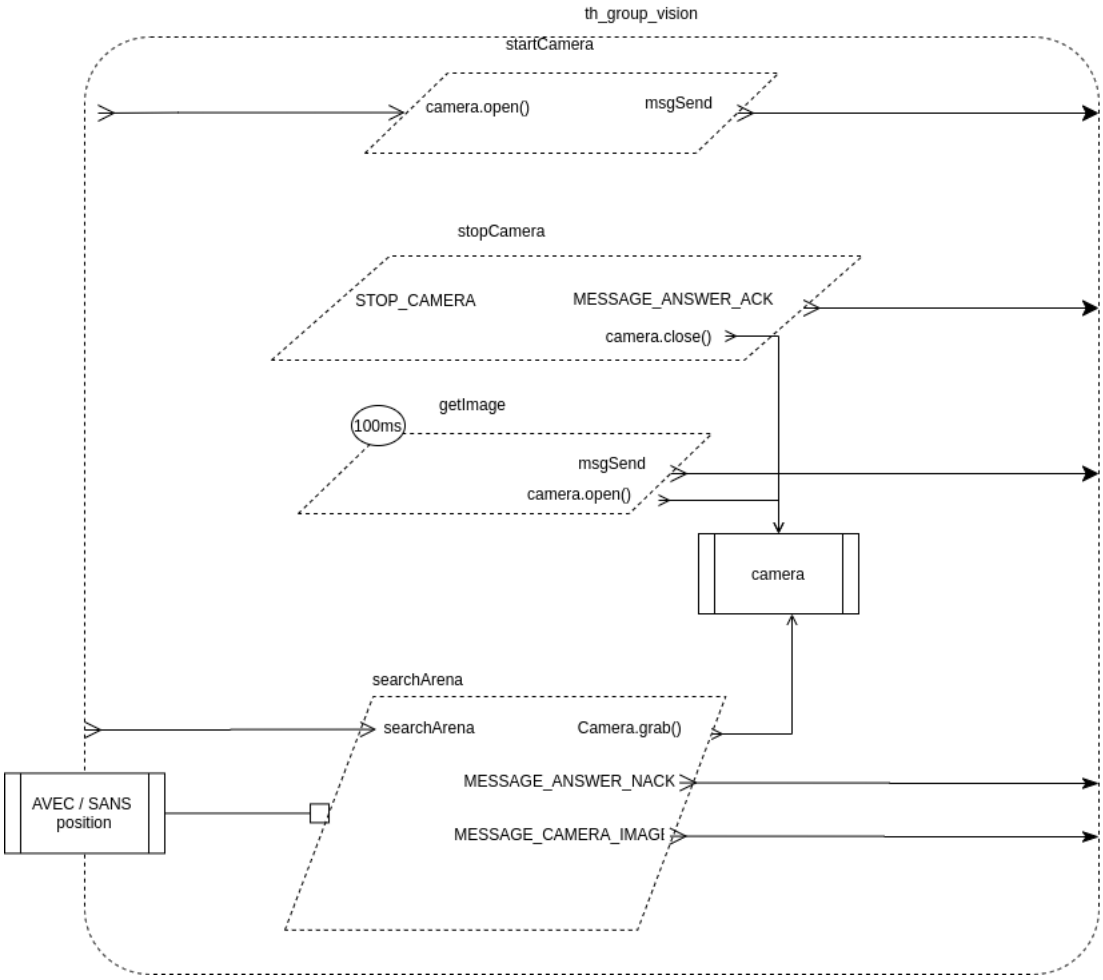

Fig. 14: Diagramme fonctionnel du groupe vision

1.4.2 Diagrammes d'activité du groupe vision

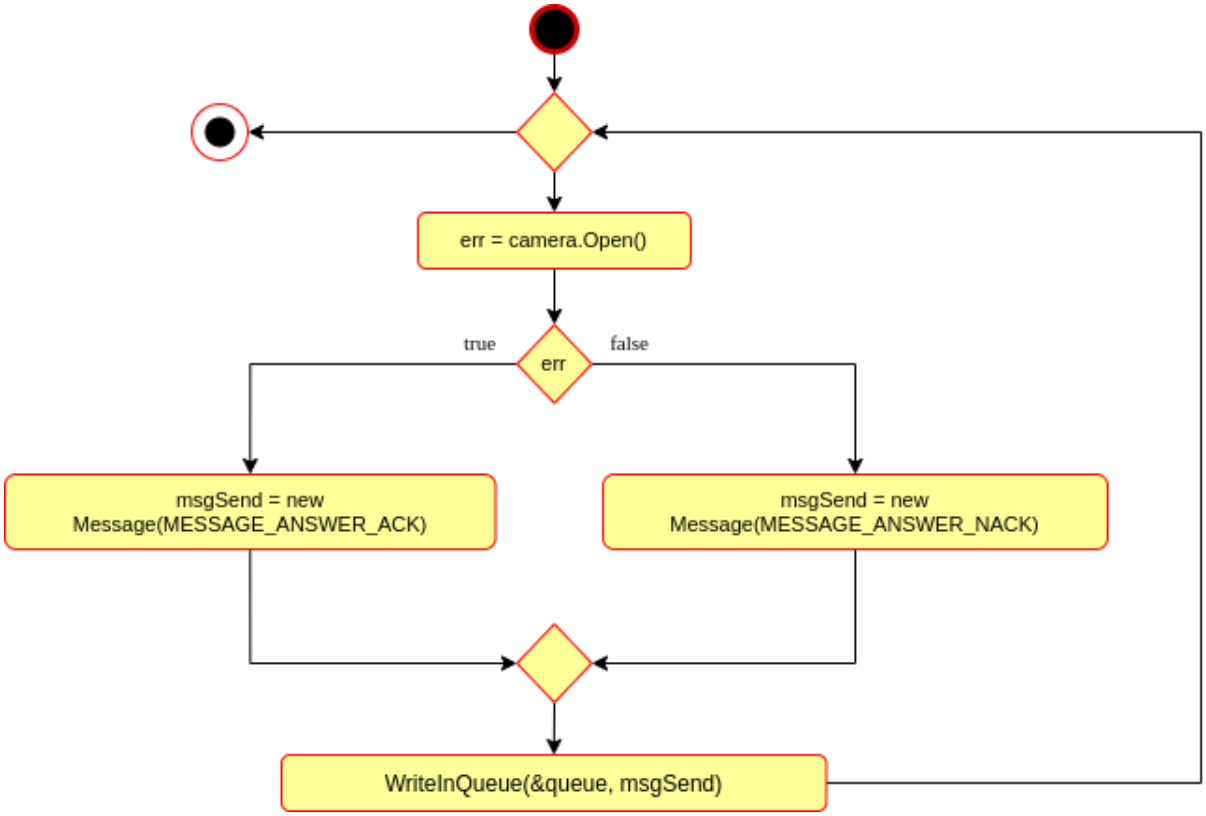

Fig. 15: Diagramme d'activité du thread `th_startCamera`

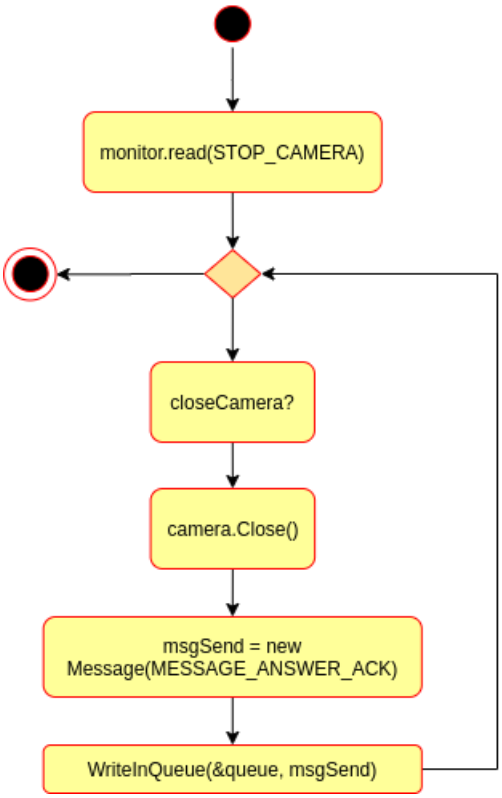

Fig. 16: Diagramme d'activité du thread `th_stopCamera`

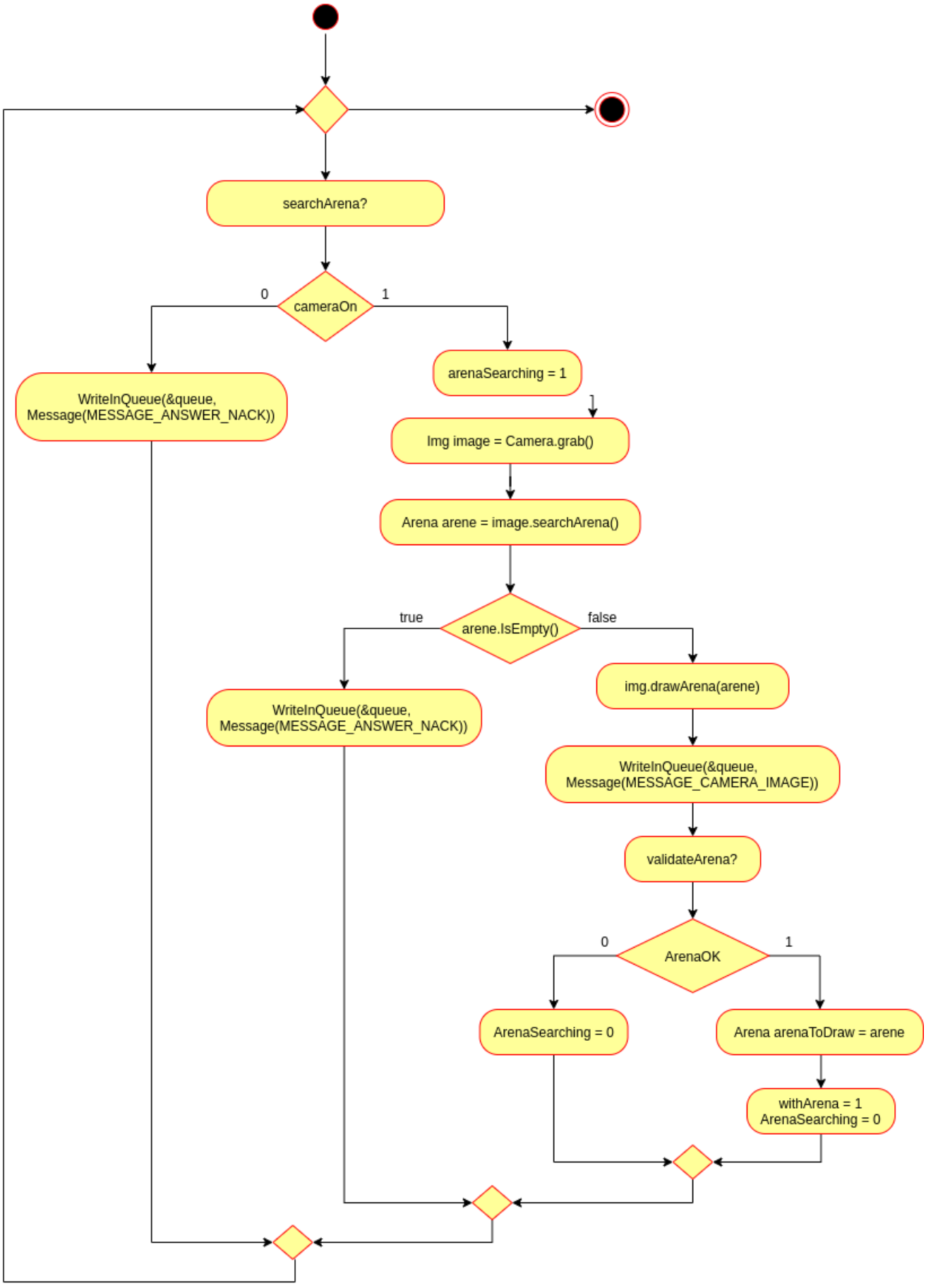

Fig. 17: Diagramme d'activité du thread `th_searchArena`

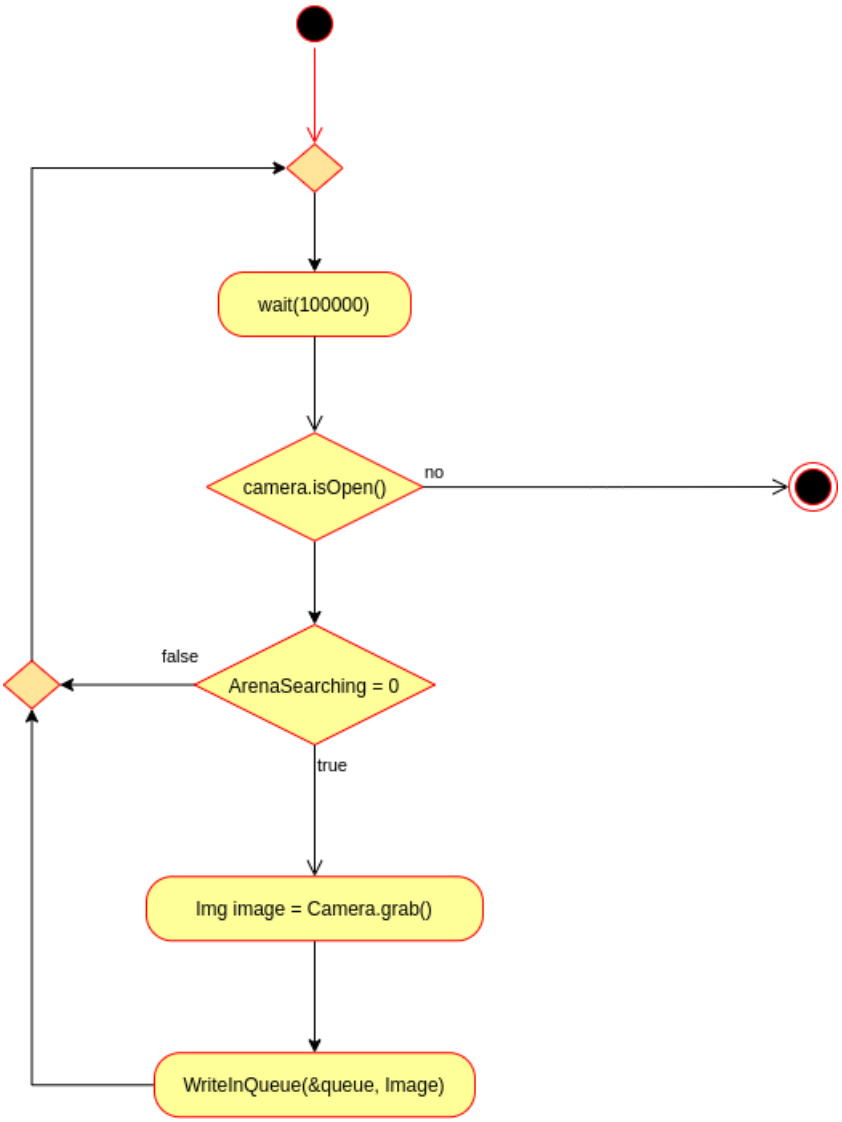

Fig. 18: Diagramme d'activité du thread `th_getImage`

2 Transformation AADL vers Xenomai

La conception obtenue a permis d'implémenter les diverses fonctionnalités du cahier des charges en C++ sous Xenomai.

2.1 Thread

2.1.1 Instanciation et démarrage

Chaque thread a été implémenté par un `RT_TASK` déclarés dans le fichier `tasks.h`. La création de la tâche se fait à l'aide du service `rt_task_create` et son démarrage à l'aide de `rt_task_start`. Toutes les tâches sont créées dans la méthode `init` de `tasks.cpp` et démarrées dans la méthode `run`.

Par exemple, pour la tâche `th_server`, sa déclaration est faite ligne 76 dans le fichier `tasks.h`

```
RT_TASK th_server;
```

sa création ligne 145 de `tasks.cpp` lors de l'appel de

```
rt_task_create(&th_server, "th_server", 0, PRIORITY_TSERVER, 0)
```

et son démarrage ligne 193 avec

```
rt_task_start(&th_server, (void(*)(void*)) & Tasks::ServerTask, this)
```

2.1.2 Code à exécuter

Sous Xenomai, chaque thread est associé à une fonction. Dans ce projet tout se passe dans la classe *task*. Tout d'abord, tous les threads sont déclarés dans *task.h*.

```
/************************************
/* Tasks
/************************************
RT_TASK th_server;
RT_TASK th_restartServer;
RT_TASK th_sendToMon;
RT_TASK th_receiveFromMon;
RT_TASK th_openComRobot;
RT_TASK th_closeComRobot;
RT_TASK th_startRobot;
RT_TASK th_move;
RT_TASK th_battery;
RT_TASK th_startCamera; //optionnel
RT_TASK th_stopCamera; //optionnel
RT_TASK th_watchdog;
RT_TASK th_getImage; //optionnel
RT_TASK th_searchArena; //optionnel
```

Fig. 19: Déclaration des threads *tasks.h, l.76*

Puis, à chaque thread seront associés une priorité et un morceau de code (ie. une tâche). Ce traitement est effectué dans la méthode **Init()** de la classe *task*.
Dans un deuxième temps, nous allons demander à tous les threads de s'exécuter. Cela se passe dans la méthode **Run()** de la classe *task*. A partir de ce moment, tous les threads sont actifs en parallèle, en respectant la hiérarchie de priorité donnée à l'initialisation.

2.1.3 Niveau de priorités

Le niveau de priorité est défini comme constante par un *#define* dans le fichier *tasks.cpp*. Chaque thread a un niveau de priorité qui lui est propre. La valeur de la constante est attribuée en fonction ce qui nous parrait être le plus prioritaire. Plus la valeur est élevée, plus le thread est prioritaire. Par exemple, la priorité du thread *th_restartServer* est la plus haute et vaut 31 (ligne 23).

```
#define PRIORITY_TRESTARTSERVER 31
```

Fig. 20: Définition de la priorité du thread th_restartServer

2.1.4 Activation périodique

Afin d'activer périodiquement un thread, on utilise la fonction *rt_task_set_periodic* qui permet de définir la période d'activation, puis *rt_task_wait_period(NULL)* à l'endroit où on lance l'activation périodique.

Par exemple, le cahier des charges spécifie que la mise à jour de la batterie doit être réalisée toutes les $500ms$. *TM_NOW* spécifie l'activation immédiate du thread et *500000000* correspond à la période en nano secondes.

```
void Tasks::BatteryLevelTask(void * arg){
    cout << "Start " << __PRETTY_FUNCTION__ << endl << flush;
    int err, rs;
    // Synchronization barrier (waiting that all tasks are starting)
    rt_sem_p(&sem_barrier, TM_INFINITE);
    cout << "Launch " << __PRETTY_FUNCTION__ << endl << flush;
    rt_task_set_periodic(NULL, TM_NOW, 500000000);

    while (1) {

        //Attente du lancement de la Comm avec le robot
        rt_task_wait_period(NULL);
```

Fig. 21: Activation périodique de la tâche mettant à jour la batterie toutes les $500ms$

2.2 Donnée partagée

2.2.1 Instanciation

Les données partagées sont instanciées dans le fichier *tasks.h* en tant que éléments private de la classe **tasks**. La donnée est du type voulu (int, float, ComRobot etc.).

```
private:
    /**********************************
    /* Shared data
    /**********************************
    ComMonitor monitor;
    ComRobot robot;
    int robotStarted = 0;
    int move = MESSAGE_ROBOT_STOP;
    int WD =-1;
    int position=0;
    int arenaOK=-1;
```

Fig. 22: Instanciation des données partagées

2.2.2 Accès en lecture et écriture

L'accès à une donnée partagée se fait avec l'utilisation d'un mutex. Un mutex est implémenté par un **RT_MUTEX** et déclaré dans le fichier *tasks.h*. Il est ensuite créé via le service **rt_mutex_create** dans la méthode **Init()** du fichier *tasks.cpp*. Afin d'affecter une valeur à une variable partagée est de s'assurer qu'un seul thread la modifie, on utilise les services **rt_mutex_acquire** avant de la modifier, puis **rt_mutex_release** une fois la modification terminée.

Par exemple, le mutex permettant d'accéder à la variable partagée permettant d'indiquer si le robot est démarré est déclaré ligne 96 dans *tasks.h*.

```
RT_MUTEX mutex_robotStarted;
```

Fig. 23: Déclaration du mutex mutex_robotStarted

Il est ensuite créé ligne 72 de *tasks.cpp*.

```
if (err = rt_mutex_create(&mutex_robotStarted, NULL)) {
    cerr << "Error mutex create: " << strerror(-err) << endl << flush;
    exit(EXIT_FAILURE);
}
```

Fig. 24: Création du mutex mutex_robotStarted

A la ligne 321 de *tasks.cpp*, on affecte la valeur de la variable partagée *robotStarted* à une variable nommée *rs*.

```
rt_mutex_acquire(&mutex_robotStarted, TM_INFINITE);
rs = robotStarted;
rt_mutex_release(&mutex_robotStarted);
```

Fig. 25: Modification de la donnée partagée robotStarted

Remarque : lorsque le thread n'est pas encore lancé, on peut modifier directement la variable sans utiliser de mutex. Comme dans Restart Serveur ou au démarrage.

2.3 Port d'événement

2.3.1 Instanciation

Un port d'événement est implémenté par un sémaphore **RT_SEM** dans *tasks.h*. Il est ensuite crée via le service **rt_sem_create** dans la méthode **Init()** dans *tasks.cpp*.

Par exemple, le port d'événement *sem_openComRobot* associé à l'ouverture de la communication avec le robot (thread *th_openComRobot*) est instancié de la manière suivante.

```
RT_SEM sem_openComRobot;
```

Fig. 26: Déclaration du port d'événement sem_openComRobot (*tasks.h*, l.103)

```
if (err = rt_sem_create(&sem_barrier, NULL, 0, S_FIFO)) {
    cerr << "Error semaphore create: " << strerror(-err) << endl << flush;
    exit(EXIT_FAILURE);
}
```

Fig. 27: Instanciation du port d'événement sem_openComRobot (*tasks.cpp*, l.89)

2.3.2 Envoi d'un événement

Afin de signaler un événement, on utilise le service **rt_sem_v**. Par exemple, lorsque le moniteur reçoit un message demandant l'ouverture de la communication avec le robot, on signale l'événement de la manière ci-dessous.

```
} else if (msgRcv->CompareID(MESSAGE_ROBOT_COM_OPEN)) {
    rt_sem_v(&sem_openComRobot);
```

Fig. 28: Signalement de l'événement openComRobot (*tasks.cpp*, l.501)

Remarque : l'appel de rt_sem_broadcast permet de débloquer tous les threads qui attendent sur le même sémaphore.

2.3.3 Réception d'un événement

L'attente d'un événement se fait grâce au service **rt_sem_p**. Par exemple, l'attente de l'événement ouvrant la communication avec le robot est la suivante.

```
rt_sem_p(&sem_openComRobot, TM_INFINITE);
```

Fig. 29: Attente de l'événement openComRobot (*tasks.cpp*, l.555)

2.4 Ports d'événement-données

2.4.1 Instanciation

L'implémentation d'un port d'événement-données est de type **Message ***. Pour créer un nouveau message, on utilise le constructeur **Message(MessageID id)** qui prend en argument un **MessageID**.

Un **MessageID** est défini par un type énuméré (*typedef enum {...} MessageID ;*) dans le fichier *messages.h* à la ligne 30.

Par exemple, pour créer un message à envoyer pour récupérer le niveau de batterie, on procède ainsi :

```
Message * mSent = new Message(MESSAGE_ROBOT_BATTERY_GET);
```

Fig. 30: Définition d'un message pour récupérer le niveau de batterie (*tasks.cpp*, l.326)

Le message associé est créé comme ci-dessous.

```
MESSAGE_ROBOT_BATTERY_GET,
```

Fig. 31: Définition d'un MessageID pour récupérer le niveau de batterie (*messages.h*, l.77)

2.4.2 Envoi d'une donnée

Afin d'envoyer des données, il faut d'abord créer une **queue** qui correspond à un buffer dans lequel les messages sont ajoutés de façon *First In First Out*. Le buffer est créé avec le service **rt_queue_create**. Les messages sont ensuite envoyés avec la fonction **WriteInQueue(RT_QUEUE *queue, Message *msg)** qui prend en argument la queue à utiliser et le message à envoyer et qui fait appel au service **rt_queue_write**.

Une queue est définie par un **RT_QUEUE** de la manière suivante. Ici, on a créé un integer pour fixer sa taille.

```
int MSG_QUEUE_SIZE;
RT_QUEUE q_messageToMon;
```

Fig. 32: Définition d'une queue (*tasks.h*, l.117)

La queue est ensuite créée en spécifiant son nom, sa taille et son type (ici FIFO).

```
/*************************************************************************************/
/* Message queues creation                                                           */
/*************************************************************************************/
if ((err = rt_queue_create(&q_messageToMon, "q_messageToMon", sizeof (Message*)*50, Q_UNLIMITED, Q_FIFO)) < 0) {
    cerr << "Error msg queue create: " << strerror(-err) << endl << flush;
    exit(EXIT_FAILURE);
}
```

Fig. 33: Création d'une queue (*tasks.cpp*, l.133)

Enfin, l'envoi d'un message dans la file se fait de la manière suivante.

```
WriteInQueue(&q_messageToMon, msgSend);
```

Fig. 34: Envoi d'un message via WriteInQueue (*tasks.cpp*, l.569)

Afin de réaliser un envoi de messages "buffeurisés" et éviter l'envoi de messages simultanés, on n'utilisera pas la fonctionnalité **monitor.Write**, sauf au moment "réel" de l'envoi des messages par **Tasks : :SendToMonTask**.
Le principe est le suivant : la tâche lit le message le plus ancien dans la file et le supprime une fois sa lecture achevée. Cette lecture est assurée par **Tasks : :SendToMonTask** grâce à la fonction **ReadInQueue** qui va envoyer tous les messages écrits dans la queue au moniteur. Cette tâche a une priorité suffisante pour assurer la bonne communication avec le moniteur.

```
msg = ReadInQueue(&q_messageToMon);
```

Fig. 35: Lecture d'un message via ReadInQueue (*tasks.cpp*, l.482)

2.4.3 Réception d'une donnée

La réception de données se fait via la fonction **moniteur.read(RT_QUEUE *queue)**, qui fait dans la **Tasks : :ReceiveFromMonTask**. Les messages sont ensuite traités en envoyant des signaux asynchrones afin de ne pas exécuter de traitement avec cette fonction, qui sert juste à donner les ordres au reste du programme.

```
msgRcv = monitor.Read();
```

Fig. 36: Réception d'une donnée (*tasks.cpp*, l.506)

Cette tâche ne gérera donc que des libérations de sémaphores ou des initialisations de variables partagées. En aucun cas elle ne doit effectuer une opération bloquante. Le but, ici, étant d'assurer la communication avec le moniteur.

3 Analyse et validation de la conception

Cette partie est constituée de vidéos qui montrent le bon fonctionnement des fonctionnalités 1 à 13 qui étaient à implémenter.

Les vidéos sont fournies dans l'archive. Elles sont aussi disponibles à l'adresse `https://drive.google.com/drive/folders/1lBd2DcxmmXsU9iZMcR085dhfqVsgw8jj?usp=sharing`.