Projet De Stijl 2.0 : Plateforme pour robots mobiles

Programmation et conception de systémes temps réel – 4éme année AE/IR

Institut National des Sciences Appliquées de Toulouse

Dossier de conception

Version 2.0.1 (11 février 2019)

Référent pédagogique : P.-E. Hladik (`pehladik@insa-toulouse.fr`)

---

1 Introduction

Ce document a pour but de vous apprendre à lire un modèle en AADL et les diagrammes d'activité qui décrivent le comportement des threads. Le document présente le résultat d'une conception réalisée à la va-vite. Cette conception ne prend en considération que la mise en place des communications et la gestion des déplacements du robot ce qui correspond aux fonctionnalités 1, 2, 3, 4, 7, 10 et 12 du cahier des charges fonctionnel.

Le code initial qui vous est fourni à la première séance de travaux pratiques correspond à l'implémentation de la conception présentée ici.

Attention, les choix qui ont été faits ne sont pas forcément les meilleurs. Tout ce qui est proposé peut être remis en cause dans la suite du travail. À vous d'ajouter, de modifier, de critiquer de manière pertinente cette conception.

2 Diagramme de contexte

La figure 1 présente le superviseur dans son contexte et ses interactions avec les autres composants de la plate-forme.

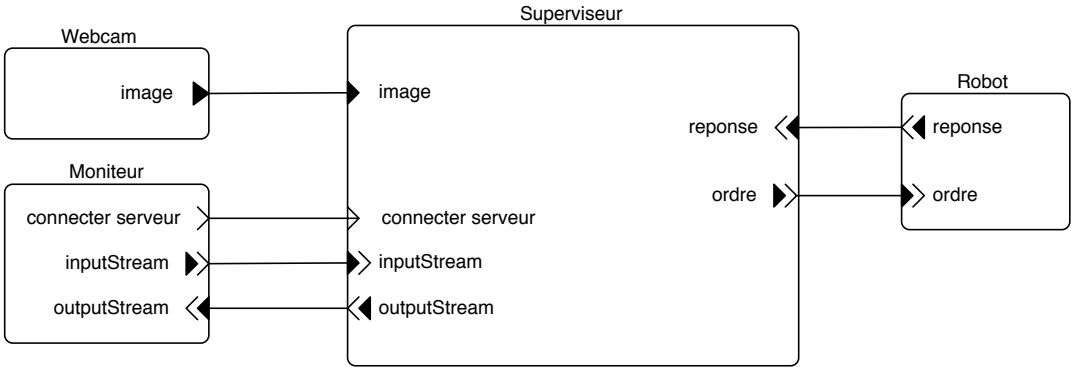


Fig. 1: Diagramme de contexte

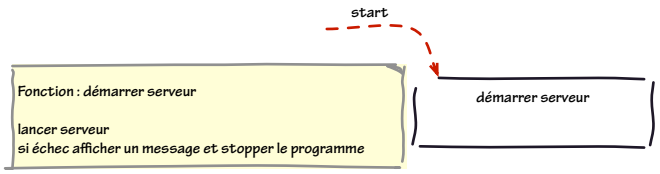
La webcam produit des données (`image`) sous la forme d'un tableau d'octets. Le robot reçoit des ordres (`ordre`) sous la forme d'une chaîne de caractères et retourne une réponse aussi sous la forme d'une chaîne (`reponse`). Le moniteur envoi un événement (`connecter serveur`) pour demander la connexion avec le serveur puis établit une communication bi-directionnelle sous la forme d'une flux d'octets (`inputStream` et `outputStream`).

3 Le travail d'un concepteur : l'analyse fonctionnelle

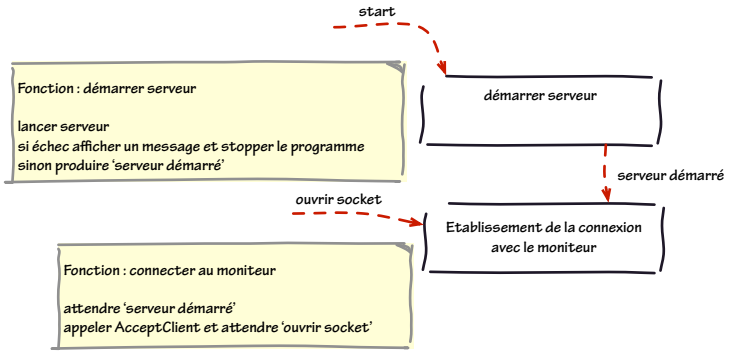
À partir du cahier des charges, le concepteur va analyser chaque fonctionnalité en se demandant quelles sont les données consommées et produites ainsi que les traitement à réaliser. Petit à petit il va ainsi construire l'architecture du programme.

Dans la suite de ce document, nous ne nous intéresserons qu'aux fonctionnalités 1, 2, 3, 4, 7, 10 et 12 présentées dans le cahier des charges fonctionnel. Votre travail consistera à compléter la conception déjà réalisée pour intégrer l'ensemble des fonctionnalités. Vous pouvez modifier tout ce que vous voulez de cette conception.

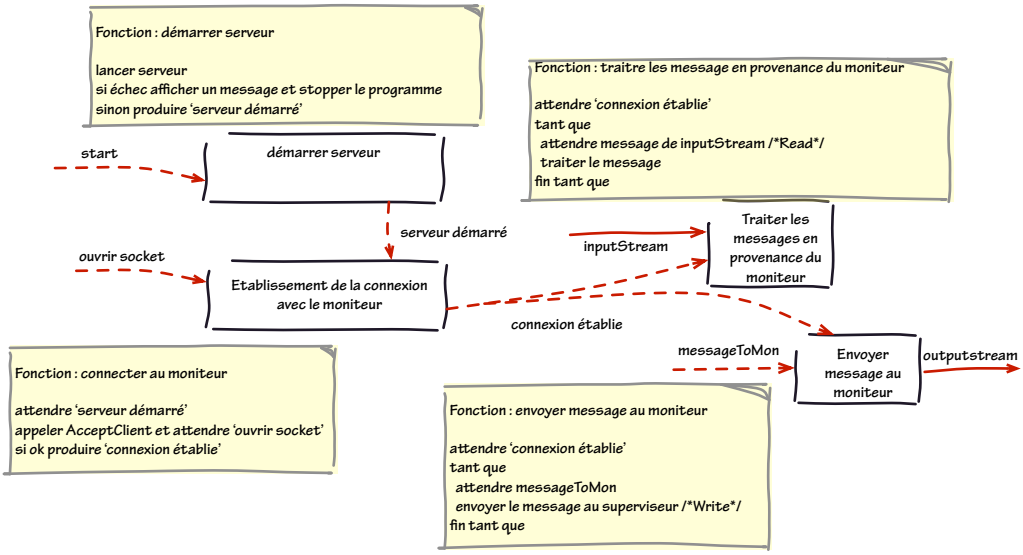
En analysant la fonctionnalité 1, le concepteur produit au brouillon le dessin suivant. Il décrit dans un rectangle blanc la fonction et dans une note sur fond jaune le comportement de la fonction. La flèche en pointillée représente un évènement.



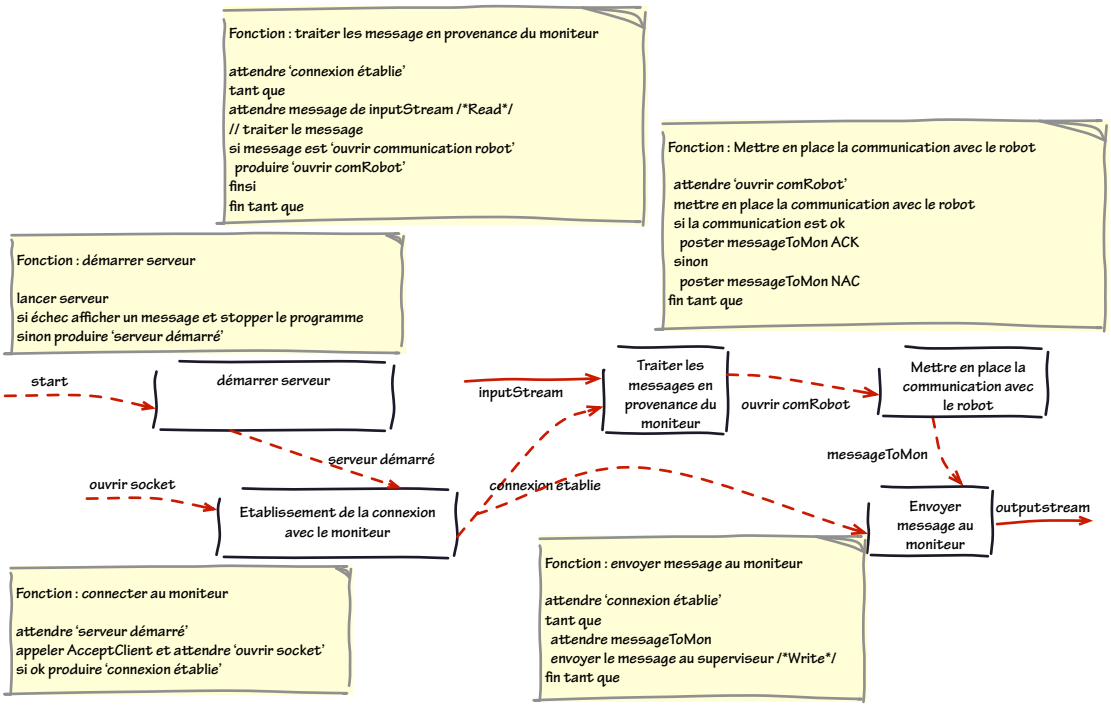
Pour intégrer la fonctionnalité 2, le concepteur complète son brouillon avec une nouvelle fonction. Il modifie aussi la première fonction pour que celle-ci produise l'événement `serveur démarré`. L'évènement `ouvrir socket` est produit par le moniteur lorsque l'utilisateur demande la connexion entre le moniteur et le superviseur.



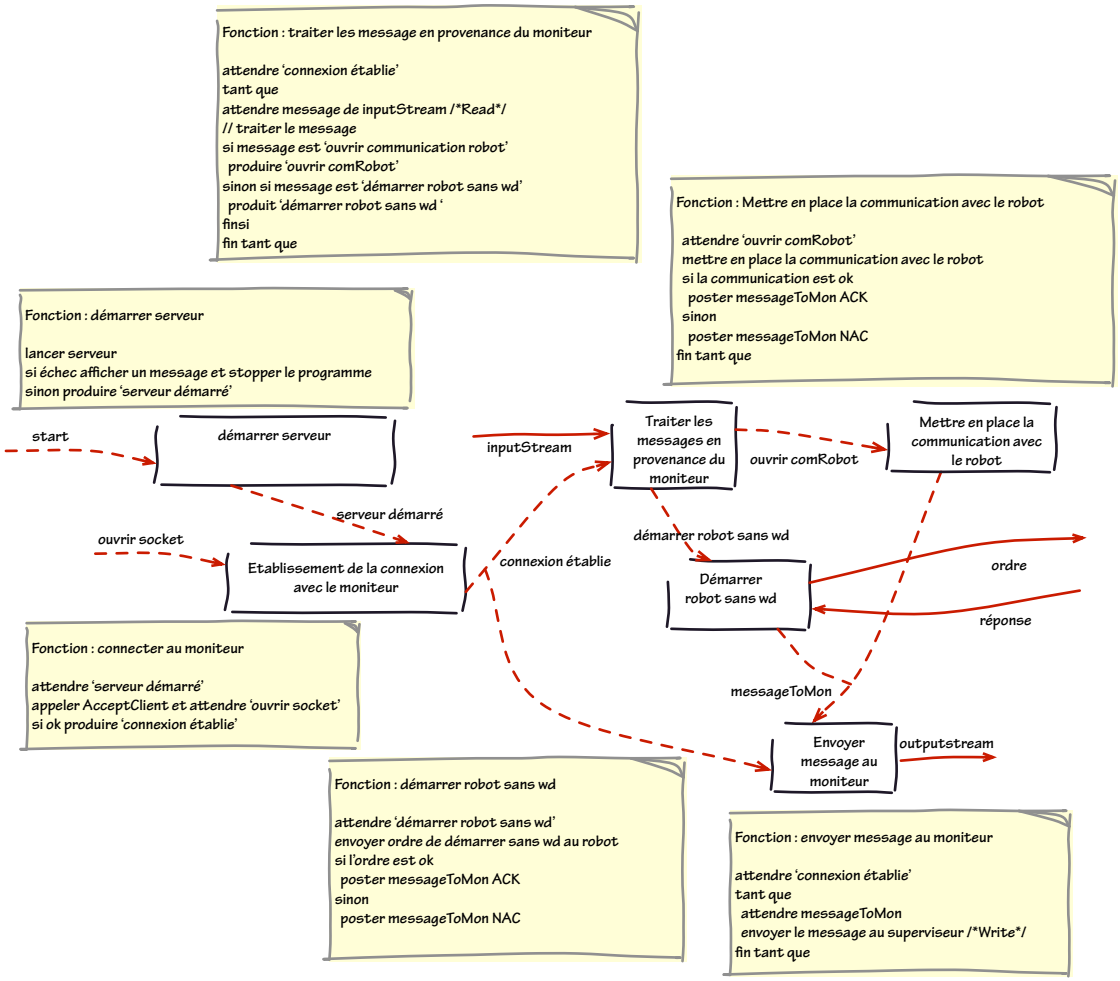
Les fonctionnalités 3 et 4 (voir dessin ci-dessous) induisent l'ajout du nouvel évènement `connexion établie`. Une file de messages `messageToMon` est aussi introduite. C'est dans cette file que seront postés les messages à envoyer au moniteur. On remarque l'apparition des flux de données-événement `inputStream` et `oututStream` en provenance et à destination du moniteur.



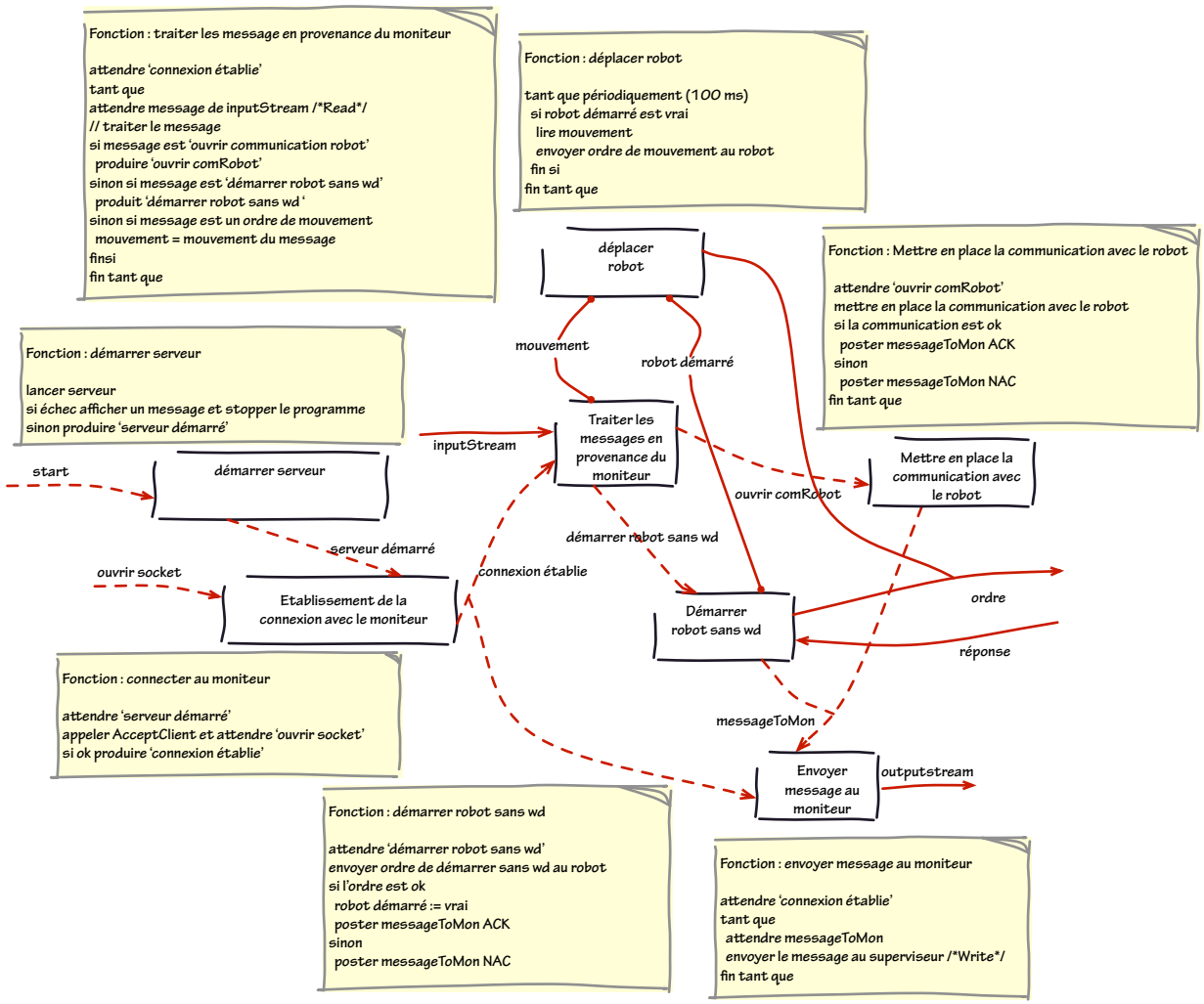
L'ajout de la fonctionnalité 7 conduit à modifier la fonction de réception pour différencier les actions à réaliser en fonction du type des messages reçus. Un nouvel évènement `ouvrir comRobot` est ainsi ajouté pour signaler une demander de mise en service de la communication avec le robot. On remarque aussi que la file de messages `messageToMon` est maintenant utilisée par une fonction pour envoyer au moniteur le message d'acquittement ou d'échec.



La fonction pour démarrer le robot sans watchdog (fonctionnalité 10) suit le même schéma que le fonctionnalité 7. On remarque cette fois que les flux `ordre` et de `réponse` sont utilisés pour communiquer avec le robot. La file de message `messageToMon` a maintenant deux producteurs.



La dernière fonctionnalité qui sera traitée dans cette conception introduit deux variables partagées : `mouvement` et `robot démarré`. Le choix fait par le concepteur pour traiter les mouvements consiste à envoyer périodiquement un message de mouvement au robot. Pour cela la fonction lit le mouvement à réaliser dans la variable `mouvement`. Cette variable est mise à jour lorsqu'un message de mouvement est reçu. Cependant, le message ne doit être envoyé que si le robot est démarré, d'où l'ajout de la variable `robot démarré` qui est mise à jour dans la fonction qui démarre le robot.



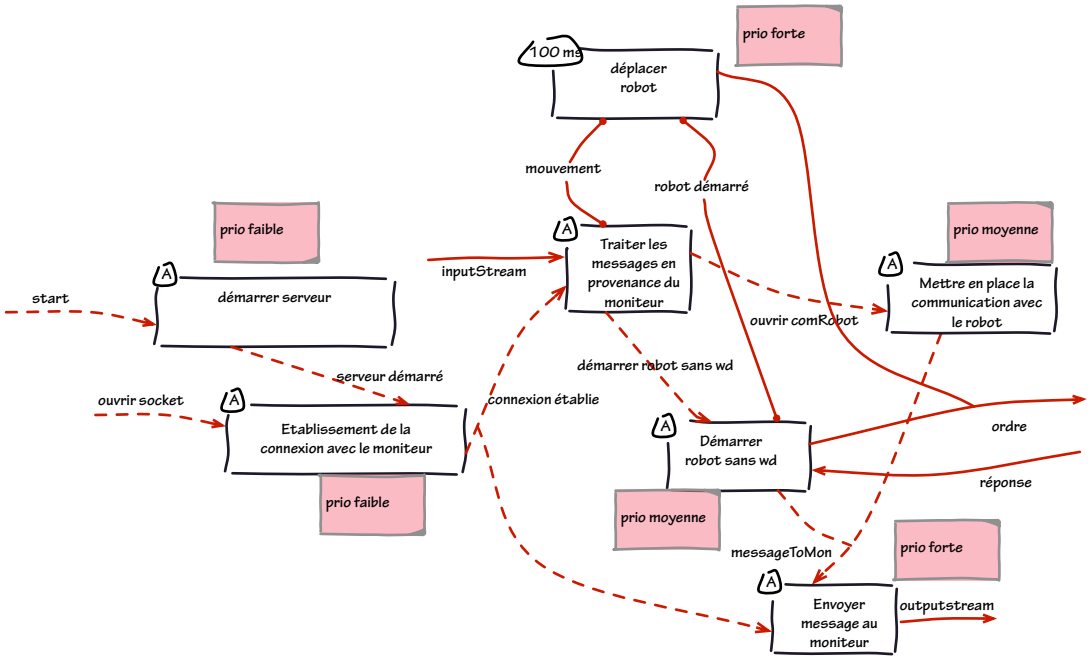
4 Le travail d'un concepteur : la formalisation

4.1 Identification des threads

Après avoir identifier les fonctions de l'application, le concepteur doit préciser quand l'exécution des fonctions se produit. La première étape consiste à considérer chaque fonction comme un thread. Deux questions se posent alors :

— Quel est le type du thread ? Un thread apériodique (noté par un A dans un rond) est un thread dont l'exécution est contrôlé par des événements alors qu'un thread périodique (noté par la valeur de la période dans un rond) aura une exécution qui revient périodiquement.

— Quel est la priorité du thread ? Une règle simple pour les threads périodiques est d'attribuer les priorités par ordre décroissant des périodes, c'est-à-dire qu'un thread sera d'autant plus prioritaire que sa période sera petite. Pour les threads apériodiques, le niveau de priorité va dépendre de la criticité des fonctions.



4.2 Raffinage

Une seconde étape consiste à avoir un regard critique sur sa conception, par exemple en se demandant s'il est possible de réunir des threads ensembles. Ici, le thread `Etablissement de la communication avec le moniteur` ne peut être appelé qu'après `démarrer serveur`, il est donc possible de les réunir en un seul thread.

C'est aussi le moment de se demande si toutes les fonctionnalités sont couvertes. Pour cela on reprend chaque fonctionnalité et on vérifie qu'elles sont bien traitées. Ici tout semble correct, bien qu'un doute subsiste pour la gestion des déplacements du robot...

4.3 Formalisation

Jusqu'ici tout a été réalisé au brouillon par le concepteur, il faut donc maintenant passer à une étape de formalisation pour pouvoir partager le travail réalisé et lever toute ambiguité. Pour cela, l'architecture sera décrite en AADL et le comportement des threads par des diagrammes d'activité UML. Le schéma 2 présente le modèle AADL de l'architecture logicielle.

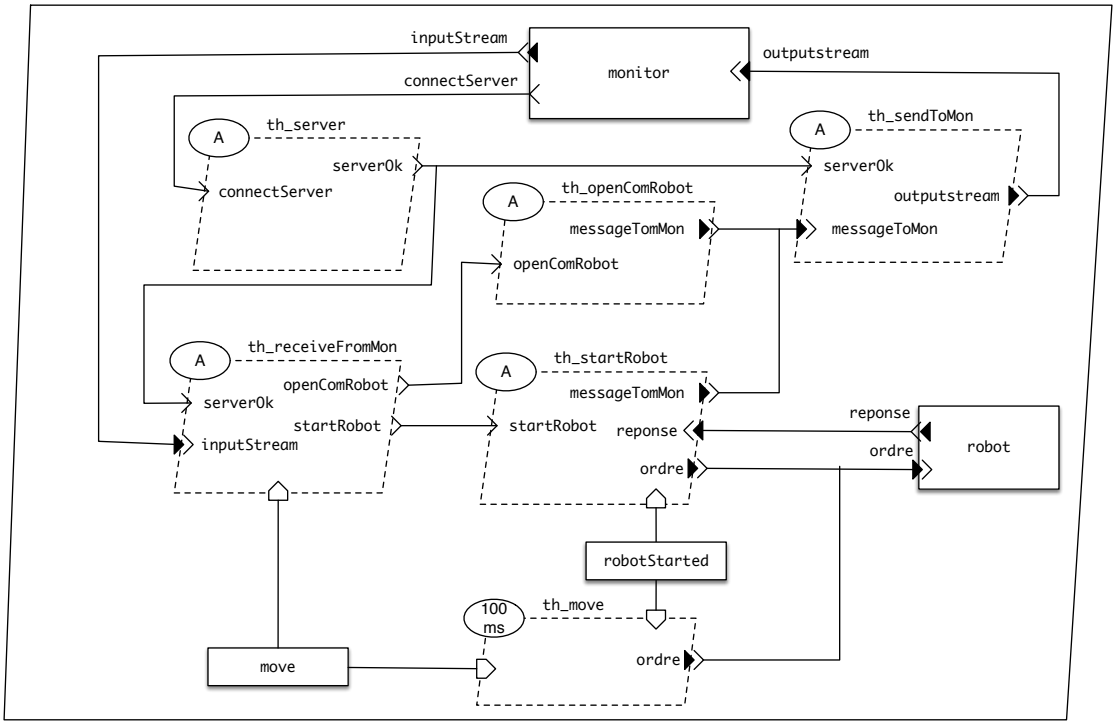


Fig. 2: Modèle AADL de l'architecture de l'application

Ensuite, pour chacun des threads, un diagramme d'activité UML est utilisé pour décrire son comportement. Les diagrammes ont été produits avec https://www.planttext.com et les codes sources sont disponibles en annexe (disponibles numériquement sur la page moodle).

4.4 Thread th_server

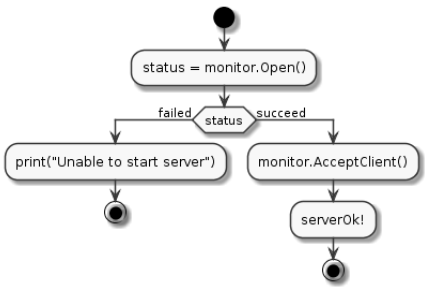


Fig. 3: Diagramme d'activité du thread th_server

4.5 Thread th_sendToMon

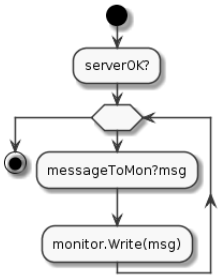


Fig. 4: Diagramme d'activité du thread th_sendToMon

4.6 Thread th_receiveFromMon

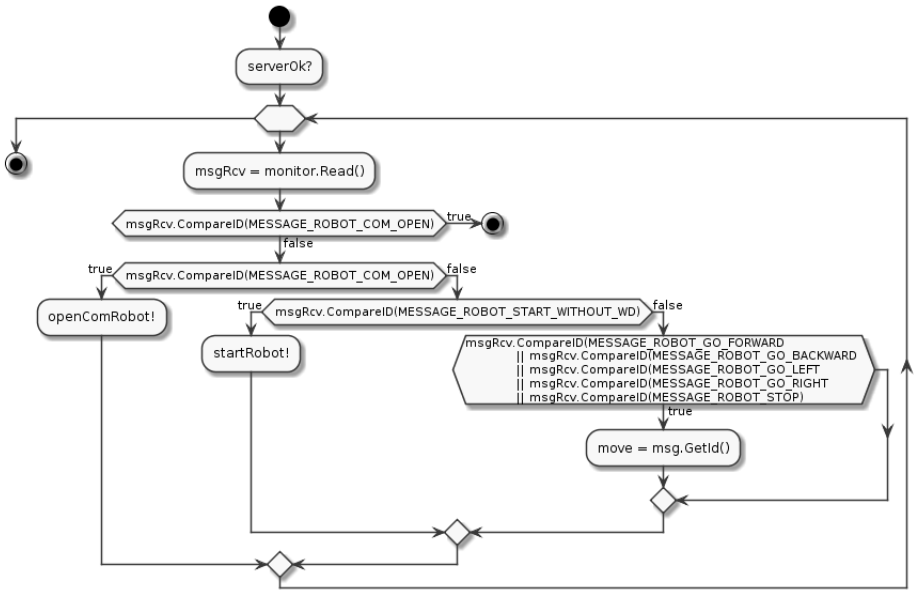


Fig. 5: Diagramme d'activité du thread th_receiveFromMon

4.7 Thread th_openComRobot

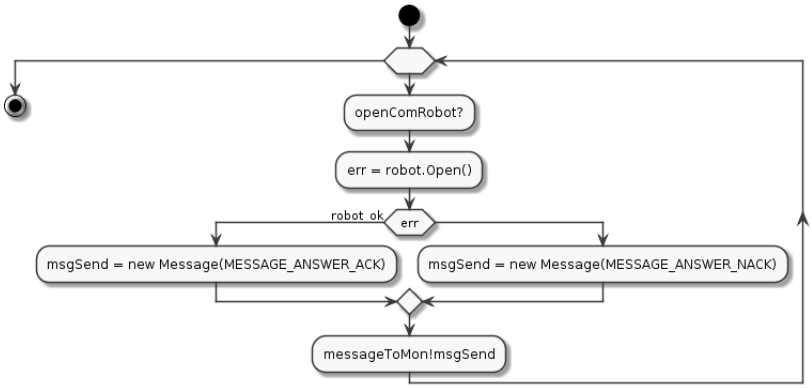


Fig. 6: Diagramme d'activité du thread th_openComRobot

4.8 Thread th_startRobot

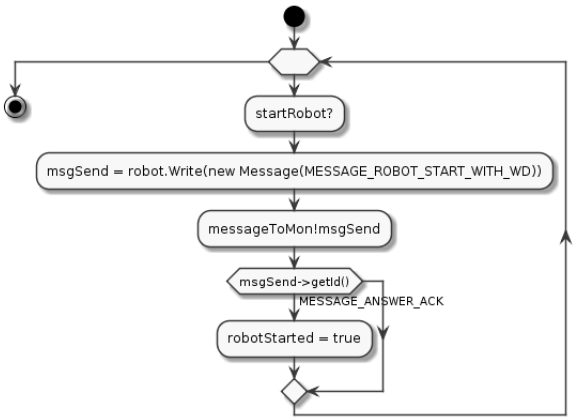


Fig. 7: Diagramme d'activité du thread th_startRobot

4.9 Thread th_move

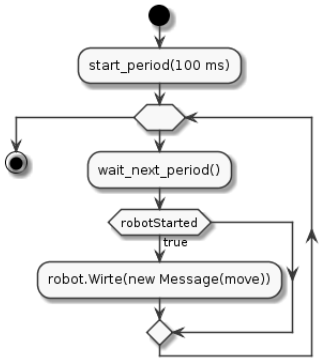


Fig. 8: Diagramme d'activité du thread th_move

Annexes

A Diagramme d'activité

> **Remarque :** Cette présentation du diagramme d'activité est tirée du cours mis en ligne de Laurent Audibert.
> Elle ne se veut pas exhaustive, mais simplement présenter les éléments nécessaire pour l'activité concernée par ce cours. De plus des éléments non standardisés sont introduit pour facilité l'expression des besoins.

Le diagramme d'activité est un diagramme comportemental d'UML, permettant de représenter le déclenchement d'événements en fonction des états du système et de modéliser des comportements parallélisables (multi-threads ou multi-processus).

Les diagrammes d'activités permettent de spécifier des traitements a priori séquentiels et offrent une vision très proche de celle des langages de programmation impératifs comme C++ ou Java. Il serait utilisé ici dans ce but.

A.1 Nœuds d'activité

Une activité modélise un comportement décrit par un séquencement organisé d'unités dont les éléments simples sont les actions. Le flot d'exécution est modélisé par des nœuds reliés par des arcs (transitions). Le flot de contrôle reste dans l'activité jusqu'à ce que les traitements soient terminés.

Un nœud d'activité est un type d'élément abstrait permettant de représenter les étapes le long du flot d'une activité. Il existe plusieurs familles de nœuds d'activités :

— les nœuds d'exécutions,
— et les nœuds de contrôle.

La figure 9 représente les différents types de nœuds d'activité.

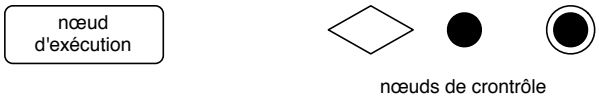


Fig. 9: Représentation graphique des nœuds d'activité

Le passage d'une activité vers une autre est matérialisé par une transition. Elles sont déclenchées dès que l'activité source est terminée et provoquent automatiquement et immédiatement le début de la prochaine activité à déclencher (l'activité cible). Contrairement aux activités, les transitions sont franchies de manière atomique, en principe sans durée perceptible.

Les transitions spécifient l'enchaînement des traitements et définissent le flot de contrôle. Elles sont représentées graphiquement par un arc entre deux nœuds.

A.1.1 Nœuds de contrôle

Un nœud de contrôle est un nœud d'activité abstrait utilisé pour coordonner les flots entre les nœuds d'une activité. Il existe plusieurs types de nœuds de contrôle (voir figure 10) :

— nœud initial : Un nœud initial est un nœud de contrôle à partir duquel le flot débute lorsque l'activité enveloppante est invoquée. Un nœud initial possède un arc sortant et pas d'arc entrant.
— nœud de fin d'activité : Lorsque l'un des arcs d'un nœud de fin d'activité est activé (i.e. lorsqu'un flot d'exécution atteint un nœud de fin d'activité), l'exécution de l'activité enveloppante s'achève et tout nœud ou flot actif au sein de l'activité enveloppante est abandonné.
— nœud de décision : Un nœud de décision est un nœud de contrôle qui permet de faire un choix entre plusieurs flots sortants. Il possède un arc entrant et plusieurs arcs sortants. Ces derniers sont généralement accompagnés de conditions de garde pour conditionner le choix. L'utilisation d'une garde [else] est recommandée après un nœud de décision car elle garantit un modèle bien formé. En effet, la condition de garde [else] est validée si et seulement si toutes les autres gardes des transitions ayant la même source sont fausses.
— nœud de fusion : Un nœud de fusion est un nœud de contrôle qui rassemble plusieurs flots alternatifs entrants en un seul flot sortant. Il n'est pas utilisé pour synchroniser des flots concurrents mais pour accepter un flot parmi plusieurs.

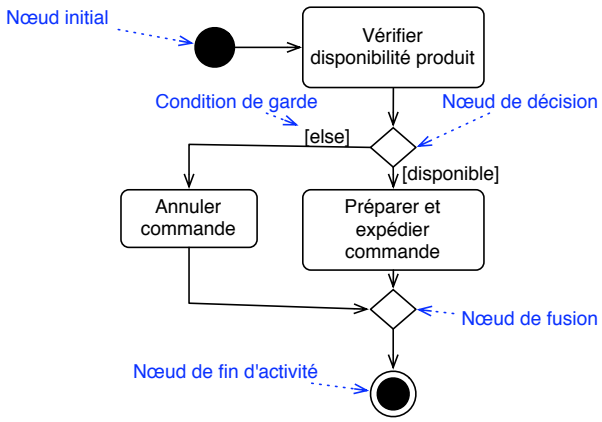


Fig. 10: Exemple de nœuds de contrôle

A.2 Nœuds d'exécution

Un nœud d'exécution est un nœud d'activité exécutable qui constitue l'unité fondamentale de fonctionnalité exécutable dans une activité. L'exécution d'une action représente une transformation ou un calcul quelconque dans le système modélisé. Les actions sont généralement liées à des opérations qui sont directement invoquées. Un nœud d'exécution doit avoir au moins un arc entrant.

Graphiquement, un nœud d'exécution est représenté par un rectangle aux coins arrondis (figure 10) qui contient sa description textuelle. Cette description textuelle peut aller d'un simple nom à une suite d'actions réalisées par l'activité. UML n'impose aucune syntaxe pour cette description textuelle, on peut donc utiliser une syntaxe proche de celle d'un langage de programmation particulier ou du pseudo-code.

Ajout de sémantique Pour faire le lien avec la conception en AADL, nous ajoutons deux actions spécifiques à la gestion des événements. Le caractère « ? » utilisé après le nom d'un événement signifie que l'action est une attente (donc bloquée) sur la réception de cet événement, alors que « ! » signifie sont émission. La figure 11 donne un exemple où une activité est en attente (bloquée) de l'événement `toto` et produit l'événement `tata`.

Pour les événements portant une donnée nous ferons suivre le symbole de synchronisation par une liste de variables qui reçoivent les données. Par exemple, `toto?var` signifie que l'action est en attente de l'événement `toto` et qu'à la réception la variable `var` aura pour valeur celle transmise.

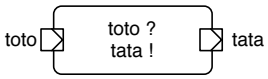


Fig. 11: Exemple de synchronisation

B Le langage AADL

AADL (Architecture Analysis and Design Language) est un langage de description d'architecture pour les systèmes temps réel. Il peut être indifféremment utilisé en avionique, spatial, robotique, etc. Il a été développé suite aux retours d'expérience sur l'utilisation du langage MethaH et a été standardisé sous l'autorité de le division ASD (Avionics System Division) du SAE (International Society for Automotive Engineers).

Une première version du standard AADL (SAE AS5506) a été produite en novembre 2004. Celle courante est la 2.0 [**?**] et date de janvier 2009. Depuis des fournitures de documents annexes et d'outils ont été proposés.

Le langage AADL étant extrêmement riche, il est impossible d'en donner ici une vue exhaustive en quelques pages. Nous ne présentons qu'un sous-ensemble de composants pour ne se focaliser que sur les principaux. Les personnes souhaitant approfondir le sujet peuvent se référer aux notes techniques disponibles sur le site de la SAE ou bien directement accéder au standard.

Le standard [**?**] est construit sur le concept de modélisation par composant. Un composant est une entité logicielle permettant de faire un calcul ou de stocker des données. Il peut représenter aussi bien une simple fonction qu'une application complète. Un composant peut être simple ou composé, il est alors dit composite. On distingue habituellement l'interface du composant qui permet de décrire les services qu'il fournit ou requière, de son implémentation qui décrit son fonctionnement interne.

AADL permet de décrire aussi bien une architecture logicielle que matérielle et de spécifier le moteur d'exécution en terme de t,ches concurrentes, de synchronisation et d'allocation. Le standard SAE AADL offre :

- une spécification du langage avec une syntaxe textuelle ;
- une sémantique et une représentation graphique ;
- un profil UML du SAE AADL ;
- d'une spécification XML/XMI comme format de modèle ;
- et plusieurs annexes détaillant certains points : formalisation du comportement, définition des interfaces avec le C et Ada, extension au modèle d'erreur, etc.

> **Remarque :** La description du langage qui en est donnée ci-après est loin d'être exhaustive et certaines libertés ont été prises pour l'adapter aux besoins pédagogiques. La suite du document est directement inspirée du guide en ligne fourni par la société Axlog.

C Notion de composant et syntaxe

La description d'une architecture en AADL consiste en la description de ses composants et leur composition sous forme d'une arborescence. Cette description pouvant être contenue dans des fichiers, une base de données, etc.

Chaque composant appartient à une **catégorie**. Ces catégories sont prédéfinies et se décomposent en :

- Catégorie matérielle avec les composants : mémoire (*memory*) ; périphérique (*device*) ; processeur (*processor*) ; bus (*bus*).
- Catégorie logicielle avec les composants : donnée (*data*) ; sousprogramme (*subprogram*) ; thread (*thread*) ; groupe de threads (*thread group*) ; processus (†*processs*).

— Catégorie composite avec les composants : système (*system*), composant abstrait (*abstract*).

C.1 Types et implémentation

Chaque composant comprend deux parties. La première, **le type**, correspond à son interface fonctionnelle, c'est-à-dire ce qui est visible pour les autres composants. La seconde, **l'implémentation**, décrit son contenu : sous-composants, propriétés, connexions, etc.

Chaque implémentation est associée à un type. ¿ chaque type est associé aucune, une ou plusieurs implémentations. La figure 12 donne un exemple de deux types de composants dont le premier est associé à deux implémentations de composants (component implementations). Textuellement, le type est introduit par le mot décrivant le type du composant, alors que l'implémentation est décrite par le type suivi du mot *implementation*.

```
system type1
end type1;

system type2
end type2;

system implementation type1.impl1
end type1.impl1;

system implementation type1.impl2
end type1.impl2;
```

Fig. 12: Exemple de description de composants

L'intérêt de scinder la description d'un composant en type et implémentation est de bien séparer ces deux points de vue. Décrire le type permet à spécifier l'interface du composant, c'est-à-dire exprimer à quoi il ressemble depuis l'extérieur. Alors que l'implémentation en représente l'intérieur. Dans la pratique, la description du type et de l'implémentation peuvent être faites par des personnes différentes, chacune ayant en charge une étape dans le raffinement de la description de l'architecture, du plus haut niveau jusqu'aux moindres détails.

C.2 Les propriétés

Chaque composant est caractérisé par des propriétés pouvant prendre des valeurs. Les propriétés sont prédéfinies, c'est-à-dire qu'elles sont identifiées par un nom, un type et la liste des catégories de composants sur lesquelles elles s'appliquent. Par exemple les *threads* disposent de propriétés telles que la période, l'échéance ou la durée d'exécution (voir figure 13).

De nouvelles propriétés et de nouveaux types de propriétés peuvent être définis par l'utilisateur et associés à tout ou partie des catégories de composants. Ce mécanisme de propriétés est un point fort d'AADL en matière d'extensibilité. Gr,ce à lui, toute notion spécifique au besoin de l'utilisateur peut être prise en compte dans sa description.

Syntaxiquement, les propriétés sont introduites par le mot *properties* suivi d'une liste de propriétés séparées par un point virgule. Les valeurs associées à une propriété sont spécifiées à l'aide de "=>".

```
thread thread1
    properties
        Period => 15 ms;
        Deadline => 10 ms;
end thread1;
```

Fig. 13: Exemple d'un *thred* avec des propriétés sur sa période et son échéance

C.3 Ports et connexions

La description des flots de données et de contrôles entre composants se fait par le moyen de **ports** et de **connexions**. Un port est un point d'entrée et de sortie d'un composant, c'est-à-dire une interface par où transitent les données et les événements entre les composants. Une connexion permet de relier deux ports, soit les ports de deux sous-composants, soit le port d'un sous-composant avec le port du composant le contenant. Le type et le sens entre les ports connectés doivent être identiques.

La figure 14 représente graphiquement des ports avec des connexions et la figure 15 en donne sa syntaxe. Le système contient deux *process*, eux-même contenant chacun un *thread*. Une succession de *ports*, représentés par les triangles, et de connexions, représentées par les lignes, établit une communication entre les deux threads.

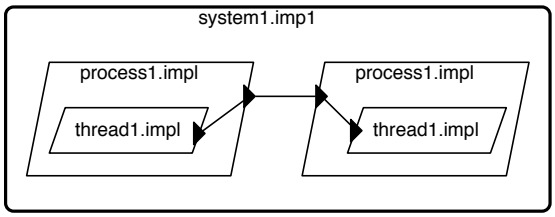


Fig. 14: Représentation d'un système et de ses sous-composants

D Outils

Différentes outils existent pour manipuler le langage AADL. En particulier, l'atelier de développement ouvert TOPCASED [**?**] fournit un éditeur textuel et graphique du langage, ADELE. De même, OSATE est un outil pour vérifier la syntaxe d'une description AADL. Cependant, ces outils ne sont pas très robuste ni « user friendly ».

E Description des principaux composants

> **Remarque :** Les principaux composants et éléments de description AADL utilisés pour la conception dans le cadre des TP sont énumérés dans les sections suivantes. Nous nous limitons volontairement à un sous-ensemble et ne respectons pas certaines règles imposées par le standard afin d'en simplifier son utilisation.
> Nous nous limitons aussi à l'expression graphique du langage. Nous ne ferons donc pas de distinction entre type et implémentation et les propriétés seront portées sous forme d'annotations sur les composants.

```
system system1
end system1;

system implementation system1.impl
    subcomponents
        p1: process process1.impl;
        p2: process process2.impl;
    connections
        cn: data port p1.outport -> p2.inport;
end system1.impl;

process process1
    features
        outport: out data port;
end process1;

process implementation process1.impl
    subcomponents
        t1: thread thread1.impl;
    connections
        cn: data port t1.outport -> outport;
end process1.impl;

process process2
    features
        inport: in data port;
end process2;

process implementation process2.impl
    subcomponents
        t2: thread thread2.impl;
    connections
        cn: data port inport -> t2.inport;
end process2.impl;

thread thread1
    features
        outport: out data port;
end thread1;

thread implementation thread1.impl
end thread1.impl;

thread thread2
    features
        inport: in data port;
end thread2;

thread implementation thread2.impl
end thread2.impl;
```

Fig. 15: Exemple de description de connexions

E.1 System

Définition Un *system* représente l'assemblage des composants logiciels d'une l'application et de sa plate-forme d'exécution.

Règles syntaxiques Un *system* peut contenir les déclaration de *data*, de *port* et de *thread*[1].

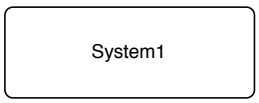


Fig. 16: Représentation graphique d'un système

1. Cela n'est pas vrai dans le standard, un système contient normalement des *processes* qui contiennent des *threads*

E.2 Thread

Définition Un *thread* modélise une activité concurrente, c'est-à-dire une unité ordonnançable qui peut être exécutée en concurrence avec un autre *thread*. Chaque *thread* est représenté par un flot de contrôle séquentiel qui exécute les instructions d'une image binaire produite par un code source. Un thread peut être activé (*dispatch*), c'est-à-dire que son exécution est provoquée par un événement qui peut être asynchrone ou périodique.

Règles syntaxiques Un *thread* peut contenir des déclarations de *port*, de *requires data access*, et contenir des *data*.

Exécution Un *thread* s'exécute à la suite d'un *disptach*. Un tel événement peut se produire périodiquement ou suite à un événement. Dans le cas périodique, le *thread* ré-exécute régulièrement le code qui lui est associé. Dans le cas d'un *dispatch* sur événement, le code peut faire référence à une attente de cet événement.

Quand le *thread* termine une exécution, il passe dans un état d'attente du prochain *dispatch*. Si un événement provoquant un dispatch est en attente, le *thread* commence immédiatement une exécution. Les événements de *dispatch* sur un événement peuvent être en attente.

Propriétés Un *thread* aura une propriété décrivant son type d'activation ainsi qu'un niveau de priorité. Dans le cas périodique, une propriété indiquant la période sera ajoutée.

```
-- Propri\'et\'es liÃ©es \'a l'activation
Dispatch_Protocol: {Periodic | Aperiodic }
Period: time
-- PropriÃ©tÃ© liÃ©e Ã  l'ordonnancement
Priority: integer
```

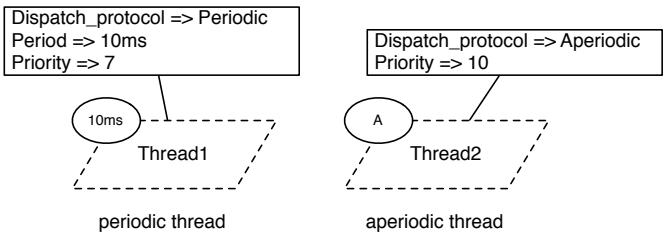


Fig. 17: Représentation graphique d'un *thread*

E.3 Data

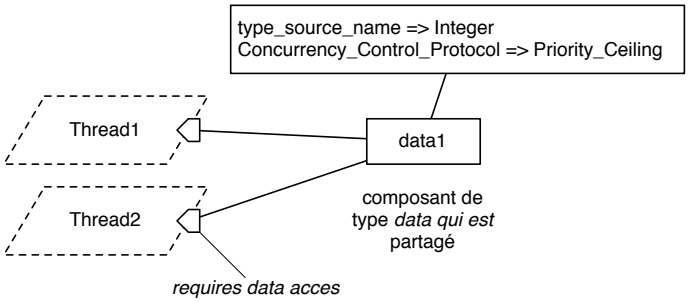
Définition Un composant de type *data* représente une donnée qui peut être accessible et partagée par d'autres composants, ce qui est modélisé par un *requires data access*. L'accès concurrent à la donnée partagée est coordonnée par le protocole de partage spécifié par la propriété `Concurrency_Control_Protocol` du composant de type *data*. Un *thread* est considéré comme étant dans une section critique quand il a accès au composant de type *data*.

Règles syntaxiques Un *data* peut contenir des déclarations de *subprogram access* et les propriétés associées.

Propriétés Un *data* est décrit par son type ainsi que par son protocole d'accès s'il est partagé.

```
- - Type de la donnÃ©e
Type_Source_Name : string
- - Protocole d'accÃ¨s Ã  la donnÃ©e
Concurrency_Control_Protocol : { Maximum_Priority | Priority_Inheritance |
                                 Priority_Ceiling | Spin_Lock | Semaphore}
```

Un ensemble de services standards est fourni pour accéder au *data*. Les services `Read_Data(Data)` et `Write_Data(Data,Value)` représentent des interfaces pour les fonctions qui réalisent une lecture et écriture de la données du *data*. Ces fonctions commencent toujours par une prise de la ressource et se termine par sa libération. La gestion de l'accès concurrent se fait suivant le protocole spécifié. Il est bien sûr possible de définir d'autres services dédiés.



E.4 Ports

Définition Les *ports* sont les points de connexion entre les différents composants qui peuvent être utilisés pour le transfère du contrôle et des données entre eux. Les *ports* sont directionnels, c'est-à-dire qu'un port en sortie (*output*) est connecté à un port en entrée (*input*). Les ports peuvent passés des données, des événements ou les deux. Les données transférées par les ports sont typées. Du point de vue du code source, les données des *ports* sont accesibles comme des variables.

Trois catégories de *ports* sont distinguées :

— Les *event data ports* sont les *ports* à travers lesquels des données sont envoyées et reçues. L'arrivée d'une donnée peut provoquer un événement chez le récepteur. Les données peuvent être mise dans une file. Un *event data port* représente les files de messages.

— Les *data ports* sont des *event data ports* avec une file de taille égale à un pour laquelle seule la dernière valeure est conservée (*blackboard*). Par défaut, l'arrivée d'une donnée ne cause pas d'activité. Les *data ports* représentent les *ports* sans file d'attente qui communiquent des informations, tels que des flux qui sont échantillonnés.

— Les *event ports* sont des *event data ports* sans contenu de message. Les *event ports* représentent des événement discrets dans l'environnement physique, tel que l'appui sur un bouton, une interruption d'horloge ou un événement logique discret comme une alarme.

Les *ports* sont directionnels. Un *out port* représente une sortie produite par un émetteur, et un *in port* représente une entrée requise par un récepteur.

Règles syntaxiques Les *ports* peuvent être déclarés dans les types des *threads* et *system*.

Propriétés À un port est associé au type de données qu'il transporte ainsi que la taille de sa file d'attente et le protocole utilisé pour gérer la file.

```
-- Taille de la file de messages, 1 par dÃ©faut
Queue_Size: aadlinteger 0 .. Max_Queue_Size => 1
```

Les *event* et *event data ports* ont par défaut une file associée avec une taille de 1 qui peut être explicitement changée en modifiant la propriété `Queue_size`. Les propriétés `Queue_Size` et `Queue_Processing_Protocol` spécifient le comportement de la file.

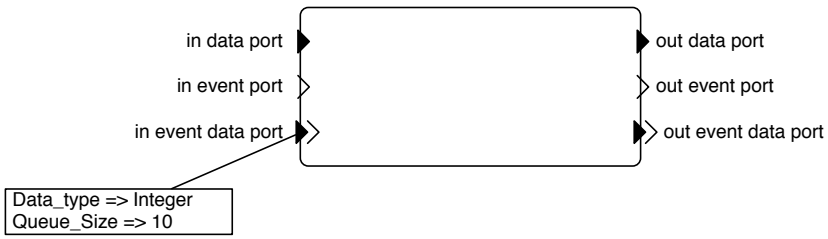


Fig. 18: Représentation graphique des *ports*

E.5 Connexions

Définition Une connexion est un lien orienté entre les *features* de deux composants qui représente les échanges de données et de contrôle entres les composants. Cela peut être la transmission de contrôle et de données entre des ports de différents *threads* ou entre des *threads* et un *data*.

Règles syntaxiques Une connexion doit contenir au moins une source et une destination et respecter le sens de communication des *ports*.

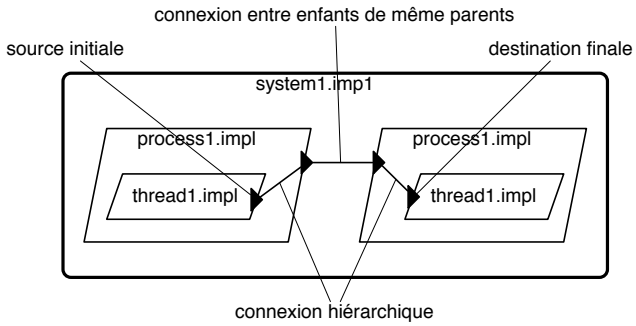


Fig. 19: Représentation graphique des connexions

F Code source des schémas d'activité

F.1 Thread th_server

```
@startuml
skinparam monochrome true
start
:status = monitor.Open();
if (status) then (failed)
    :print("Unable to start server");
stop
else (succeed)
    :monitor.AcceptClient();
    :serverOk!;
    stop
endif
@enduml
```

F.2 Thread th_sendToMon

```
@startuml
skinparam monochrome true
start
:serverOK?;
while ()
  :messageToMon?msg;
  :monitor.Write(msg);
endwhile
stop
@enduml
```

F.3 Thread th_receiveFromMon

```
@startuml
skinparam monochrome true
start
:serverOk?;
while ()
  :msgRcv = monitor.Read();
  if (msgRcv.CompareID(MESSAGE_MONITOR_LOST)) then (true)
    stop
  else (false)
    if (msgRcv.CompareID(MESSAGE_ROBOT_COM_OPEN)) then (true)
      :openComRobot!;
        else (false)
        if (msgRcv.CompareID(MESSAGE_ROBOT_START_WITHOUT_WD)) then (true)
            :startRobot!;
        else (false)
            if (msgRcv.CompareID(MESSAGE_ROBOT_GO_FORWARD
                || msgRcv.CompareID(MESSAGE_ROBOT_GO_BACKWARD
                || msgRcv.CompareID(MESSAGE_ROBOT_GO_LEFT
                || msgRcv.CompareID(MESSAGE_ROBOT_GO_RIGHT
                || msgRcv.CompareID(MESSAGE_ROBOT_STOP)) then (tru
                :move = msg.GetId();
            endif
        endif
    endif
  endif
endwhile
stop
@enduml
```

F.4 Thread th_openComRobot

```
@startuml
skinparam monochrome true
start
while ()
  :openComRobot?;
  :err = robot.Open();
  if (err) then (robot_ok)
    :msgSend = new Message(MESSAGE_ANSWER_ACK);
  else
    :msgSend = new Message(MESSAGE_ANSWER_NACK);
  endif
  :messageToMon!msgSend;
endwhile
stop
@enduml
```

F.5 Thread th_openStartRobot

```
@startuml
skinparam monochrome true
start
while ()
  :startRobot?;
  :msgSend = robot.Write(new Message(MESSAGE_ROBOT_START_WITH_WD));
  :messageToMon!msgSend;
  if (msgSend.GetId()) then (MESSAGE_ANSWER_ACK)
```

```
    :robotStarted = true;
  endif
endwhile
stop
@enduml
```

F.6 Thread th_move

```
@startuml
skinparam monochrome true
start
:start_period(100 ms);
while ()
  :wait_next_period();
  if (robotStarted) then (true)
  :robot.Wirte(new Message(move));
  endif
endwhile
stop
@enduml
```