Environnement de développement ContrACT: Tutoriel

Robin Passama

Environnement de développement ContrACT: Tutoriel

Auteur : Robin Passama
email : passama@lirmm.fr
Rapport LIRMM n°: RR-10026

1) Introduction

Ce document fournit à l'utilisateur de la méthodologie ContrACT une explication détaillée de l'utilisation de l'environnement de développement associé à la méthodologie. Les questions relatives à l'élaboration de la méthodologie, aussi bien pour ses aspect conceptuels que logiciels, trouvent leur réponses dans le document dédié à cet effet.

Ce document est décomposé en plusieurs sections. La section suivante rappelle les concepts de bases détaillée dans le document de conception de la méthodologie, nécessaires à la compréhension du présent document. La section 3 fournit une présentation générale de l'outil. La section 4 présente la façon d'éditer une architecture de contrôle ContrACT dans l'environnement de développement. La section 5 détaille le processus de génération de code à partir de la description architecturale et présente également les différentes possibilités de personnalisation du code. Enfin la section 6 conclu ce document, la section 7 répertorie les références utilisées dans ce document et la section 8 contient la description de l'exemple utilisé tout au long de ce document.

2) Concepts de bases

Une architecture de contrôle ContrACT est « construite » à partir d'entité logicielles appelées modules. Un module (cf. fig.1) est une entité logicielle qui interagit avec d'autres modules à travers son interface, son comportement interne n'étant pas directement accessible ou modifiable depuis l'extérieur du module.

Un module possède comme caractéristiques principales (cf. fig.1) :

- Un **port de requête**, c'est-à-dire un point d'entrée de messages qui permettent à d'autres modules de contrôler son activité (démarrage, arrêt) et sa configuration (fixation d'un paramètre, abonnement de ses flux).
- Un ensemble de **ports de paramètres**, c'est-à-dire un ensemble de points d'entrée qui permettent à d'autres modules de fixer la valeur d'un de ses paramètres « publics ».
- Un ensemble de **ports d'entrée de données**, qui correspondent chacun à un flux de données entrant dans le module. Ces ports peuvent être connectés à des ports de sortie de données d'autres modules.
- Un ensemble de **ports de sortie de données**, qui correspondent chacun à un flux de données sortant du (générés par) module. Ces ports peuvent être connectés à des ports d'entrée de données d'autres modules.
- Un ensemble de **ports d'entrée d'événements**, qui correspondent chacun à un flux de d'événements entrant dans le module. Ces ports peuvent être connectés à des ports de sortie de d'événements d'autres modules.
- Un ensemble de **ports de sortie de d'événements**, qui correspondent chacun à un flux de d'événements sortant du (générés par) module. Ces ports peuvent être connectés à des ports d'entrée d'événements d'autres modules.

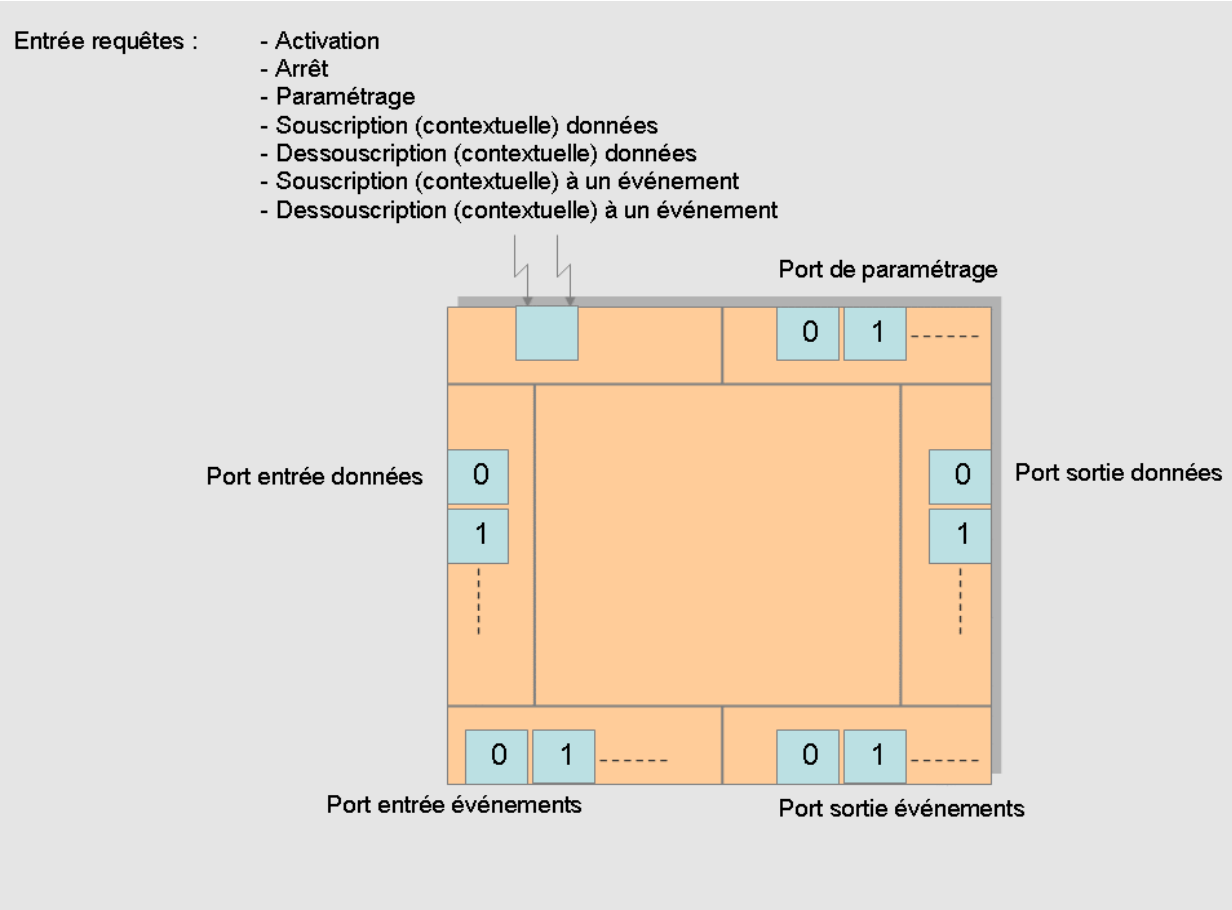

Figure 1 : Modèle de module

Excepté le port de requête qui est le point d'entrée global du module tous les autres ports sont associé à une **donnée** qui elle même respecte un **type de donnée**. Les différences entre les types de ports présentés sont les suivantes:

- Le **port de requête**, les **ports de paramètres** et les **ports d'entrée d'événements** sont des ports déclencheurs, ce qui signifie que l'arrivée d'un message va potentiellement engendrer une réaction du module.
- Par opposition, les **ports d'entrée de données** sont non déclencheurs, l'arrivée d'un message (i.e. une donnée) n'engendrera pas de réaction de la part du module. Une fois activé par ailleurs, le module pourra seulement venir lire son port d'entrée de donnée pour savoir s'il contient une donnée.
- Les **ports de sortie de données** (respectivement **d'événement**) ne peuvent générer des données que vers les **ports d'entrées** (respectivement **d'événement**) auxquels ils sont **connectés**. C'est le comportement interne du module qui décide quand produire des données et des événements en mettant à jour les données associés aux ports de sortie correspondant. Le comportement interne du module n'a pas connaissance des modules vers lesquels ces données et événements produits seront acheminés.
- Les ports d'**entrée/sortie** de **données/événements** peuvent être connectés soit par le comportement interne du module qui aurait alors une connaissance particulière de ses relations avec les autres modules, soit plus couramment par un module tier. Ainsi sont créés

des flux d'événement et de données au sein du système.

- La connexion et la déconnexion des ports d'**entrée/sortie** de **données/événements** des modules se fait sur demande au module producteur (celui qui produit la donnée ou l'événement) via des **requêtes de configuration**. Ainsi l'ensemble des connexions au sein de l'architecture logicielle sont dynamiques (i.e. susceptibles de changer durant l'exécution).
- Les **connexions entre ports de données** sont **continues** (il faut un désabonnement explicite auprès du module producteur pour que le flux doit interrompu) et **cycliques** (un module reçoit les données produites que toutes les x activations du module producteur, avec x compris entre 1 et l'infini).
- Les **connexions entre ports d'événements** sont soit **continues**, soit **éphémères** (le désabonnement du consommateur se fait dès lors que l'événement a été produit et diffusé).
- Les ports de **sortie** de **données** (respectivement **événements**) peuvent être connectés à un nombre quelconque de ports d'entrée de **données** (respectivement **événements**), mais un port d'entrée de **données** (respectivement **événements**) ne peut être connecté qu'à un seul port de **sortie** de **données** (respectivement **événements**).
- Les **ports de paramètres** servent à recevoir des données correspondant aux nouvelles valeurs des paramètres modifiés. Il n'y a pas de notion de flux et de connexion pour les ports de paramètre, ils sont donc utilisables uniquement dans une relation un-à-un de type appel de procédure entre deux modules.

Un module est implémenté comme une **tâche temps-réel**, c'est-à-dire l'équivalent d'un processus pour un système d'exploitation temps-réel avec ordonnancement à priorités fixes (les priorités ne peuvent changer que sur requête de l'application temps-réel elle-même et non du fait de l'ordonnanceur de l'OS par exemple). Ceci implique qu'une implémentation correcte ne peut se faire que sur un système d'exploitation assumant cette caractéristique, ce qui est le cas de la plupart des systèmes temps-réel existants. En effet, cette caractéristique est essentielle à un fonctionnement conforme à notre spécification d'une architecture conçue suivant la décomposition logicielle proposée. Nous supposons par ailleurs que les modules sont implémentés au sein de l'OS comme des **tâches asynchrones**, c'est-à-dire qu'elle ne seront pas réveillées périodiquement par l'ordonnanceur de l'OS à la différence de tâches périodiques. Grâce à ces deux caractéristiques nous pouvons (et nous devons) contrôler précisément les **moments d'activation** des modules (si nécessaire) et leurs **relations de préemption**. Au final cela nous permet d'avoir un contrôle complet et **déterministe** du système.

Une architecture de contrôle logicielle générique construite à partir de modules a été définie (cf. figure 2). Elle répartie les modules sur deux couches conceptuelles : la couche **décisionnelle** et la couche **exécutive**. Dans la couche décisionnelle on retrouve l'ensemble des superviseurs servant à décomposer le mécanisme de prise de décision du robot. Les superviseurs suivent une relation hiérarchique depuis le **superviseur global** (unique point d'entrée du système chargé de gérer globalement l'application) jusqu'aux s**uperviseurs locaux** les plus proches (en terme de priorité) de la couche exécutive, cette relation hiérarchique déterminant leur **niveau de priorité** (au sens système d'exploitation et en même temps au sens décisionnel). Ces superviseurs, quel que soit leur niveau hiérarchique, peuvent interagir directement avec les superviseurs hiérarchiquement inférieurs ou avec les modules de la couche exécutive (voir plus loin). Les interactions entre superviseurs sont de deux types : événementielle et par requêtes de configuration (envoi de paramètres). Les **interactions événementielles** permettent à un module superviseur de notifier le ou

les superviseurs de niveau hiérarchique supérieur de l'occurrence d'un événement qu'il a détecté. Quel que soit son niveau, un superviseur **reçoit des paramètres qui correspondent aux objectifs** qu'il doit réaliser ou abandonner et **produit des paramètres qui correspondent à des sous-objectifs** qu'il souhaite voir réaliser ou abandonner. Cette organisation hiérarchique peut notamment servir à la gestion des modes : le superviseur global se charge de la **commutation des modes d'autonomie** alors que les superviseurs locaux peuvent représenter chacun **un mode d'autonomie particulier** (e.g. téléopération directe, télé-programmation, autonome, mixte, coopération multi-robots, etc.). Cette vision peut être raffinée autant de fois que nécessaire, par exemple en créant des modes plus ou moins dégradés des modes d'autonomie considérés en rajoutant un niveau hiérarchique de superviseurs locaux. La gestion de la préemption est simple : dès lors qu'un module superviseur est activé (sur réception d'un événement ou d'un paramètre) il préemptera tous les modules superviseurs de niveaux hiérarchiques (et donc de priorités, cf. fig. 4) inférieurs. Ceci se justifie par le fait que les décisions prises à un niveau décisionnel donné sont plus importantes (notamment parce qu'elles peuvent les remettre totalement en cause -i.e. déclencher l'abandon des objectifs) que les décisions prises dans les niveaux inférieurs et doivent dont être plus prioritaires.

Dans la **couche exécutive**, on retrouve des modules se situant dans deux mondes différents un **monde continu** (en bas à gauche fig. 2) et un **monde événementiel** (en bas à droite fig. 2). Pour gérer le **monde continu**, la décomposition logicielle définit qu'il y a un module **ordonnanceur** et des **modules synchrones**, ordonnancés par celui-ci. Rappelons que tous ces modules sont asynchrones au sens système, mais : le module ordonnanceur est **pseudo-périodique** (génère des timers pour se réveiller périodiquement, si besoin est en fonction du contexte) et les modules synchrones sont activés périodiquement par le module ordonnanceur qui leur envoi des requêtes d'activation (ils sont donc périodiques du point de vue applicatif). L'ordonnanceur est capable de notifier des événements relatifs à des problèmes d'ordonnancement (retard considérés comme critiques) vers les superviseurs. Ces derniers activent le module ordonnanceur en lui demandant **d'activer ou de désactiver l'ordonnancement d'un ou plusieurs schémas** via l'envoi de paramètres. Les **modules synchrones** servent à implémenter les algorithmes et les entrée/sorties utilisés pour implanter les lois de commandes, observateurs, ou plus généralement, tout type de boucle de calcul et d'interaction périodique. Ils offrent la possibilité de décomposer à grain fin ces boucles afin de mieux en réutiliser différentes parties dans différents schémas. Ils peuvent donc interagir avec des entrées / sorties et/ou réaliser des calculs à partir de données en entrée et produisant des données en sortie. Pour gérer le **monde événementiel**, la décomposition définit deux types de modules : des **modules d'interaction** et des **modules temps-restant**. Ces modules sont des modules asynchrones, c'est-à-dire **non périodiques** et donc **non-ordonnancés par le module ordonnanceur** applicatif. Le principale contrainte sur ces modules synchrones est qu'ils ne peuvent communiquer via des flux de données (i.e. ils n'ont pas de port d'entrée / sortie de donnée). Les **modules d'interaction** sont utilisés pour implanter des interactions avec des systèmes externes via des entrées / sorties qui ne sont pas temporellement prédictibles. Certains de leurs ports de sortie d'événements sont utilisés par les superviseurs pour se voir notifier des événement considérés comme pertinents et leur ports de paramètres peuvent servir à paramétrer leur exécution « en ligne » et éventuellement à déclencher certaines réactions. Les **modules temps restants** sont utilisés pour implanter des **calculs « longs »** que doivent effectués les superviseurs, comme par exemple des planifications de trajectoire, de chemin ou des planification de mission.

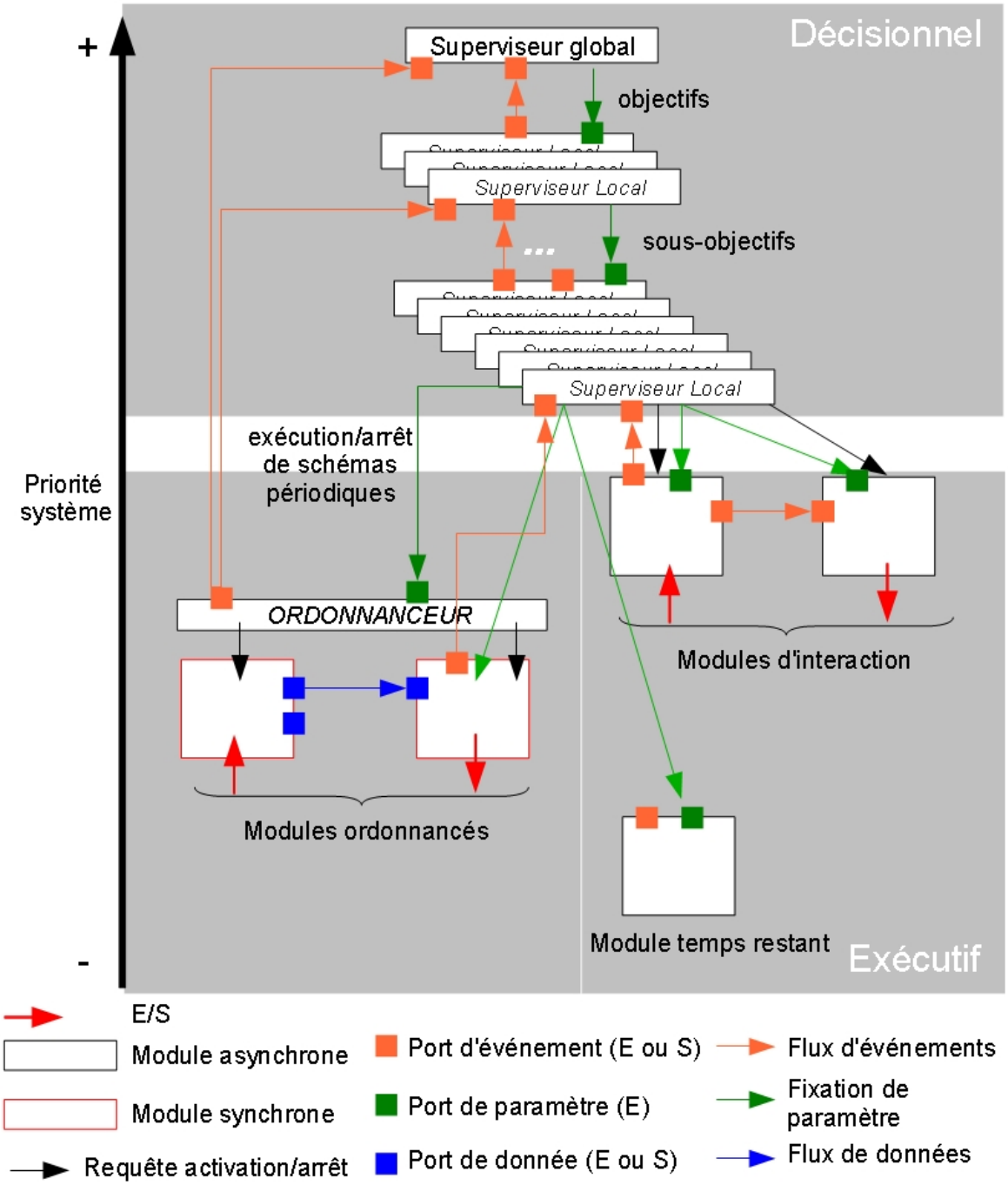

Figure 2 : Schéma résumant la décomposition logicielle proposée

Des explications détaillées sur l'architecture génériques sont fournies dans le document ***[Passama 2010]***. L'outil logiciel qui permet de créer des architectures de contrôle logicielles suivant l'architecture générique présentée est développé dans les section suivantes

3) Vue globale sur l'outil

L'environnement de développement ContrACT permet de créer des projets d'architecture de contrôle, chaque projet correspondant à une architecture de contrôle liée à une application robotique particulière. Dans l'outil un projet est structuré en deux sous-dossiers. Un dossier contenant le code source, c'est-à-dire les fichiers écrits dans le langage dédié, et l'autre dossier contenant le code généré, c'est-à-dire des fichiers source et d'entête C, une arborescence de répertoires et des fichiers de compilation (Makefile), comme présenté dans la figure 3.

Tous les fichiers sources écrit dans le langage dédié **Cactal** (pour ContrACT Advanced Language) possèdent la même extension (*.contract*) mais sont différenciés en fonction de leur contenu:

- *les fichiers de définition de type*. Ces fichiers contiennent la définition des types de données utilisés dans les échanges entre les modules utilisés dans l'architecture. L'existence des types de données échangés au niveau du langage Cactal se justifie par une volonté de rendre le développement plus fiable en faisant du test de type lors de la connexion des modules: on peut ainsi savoir si des modules échange des données de même type et si ce n'est pas le cas reporter une erreur. A partir de ces types de données, les structures C équivalente et les différentes fonctions servant à leur manipulation dans le middleware sont automatiquement générées. Les fichiers de définition de type peuvent également contenir des déclarations de fonctions que l'utilisateur devra implémenter en C. Nous verrons par la suite comment utiliser ces fonctions.

- *Les fichiers de définition d'interface de module fonctionnel*. Ces fichiers servent à définir les interfaces et propriétés essentielles des modules dit « fonctionnels », c'est-à-dire les modules contenant les fonctions de base programmées (en C) par l'utilisateur. Chaque fichier correspond donc à l'interface d'un module **synchrone** ou **asynchrone (temps-restant ou d'interaction)**. Une interface contient le nom du module, la définition de ses ports de paramètres, d'entrée / sortie de données (module synchrone uniquement) ou d'événements (module asynchrone uniquement pour les ports d'entrée d'événements) de façon conforme à la méthodologie. Les modules synchrones possèdent en plus une spécification de leur paramètre de **temps d'exécution nominal,** utile à l'ordonnanceur. A partir de ces informations des squelettes de code complets des modules fonctionnels sont générés et l'utilisateur n'a plus qu'à remplir les fonctions implémentant leur comportement (initialisation, terminaison, exécution nominale, réaction arrivée de paramètres, réaction aux événements).
- *Les fichiers de définition des schémas*. Ces fichiers servent à définir les **schémas périodiques ou événementiels** utilisés dans l'architecture, c'est-à-dire les assemblages de modules (via leur ports de données pour les schéma périodiques ou d'événement pour les schéma événementiels) et leurs paramétrage, ainsi que, pour les schémas périodiques les propriétés nécessaires à leur ordonnancement (période, délai critique et graphe de précédence). Chaque fichier correspond à un schéma, mais ne génèrera pas de module particulier: il sert à générer du code à plusieurs endroits dans l'architecture logicielle, dans l'ordonnanceur (pour les schémas périodiques) et dans les superviseurs utilisant ces schémas.

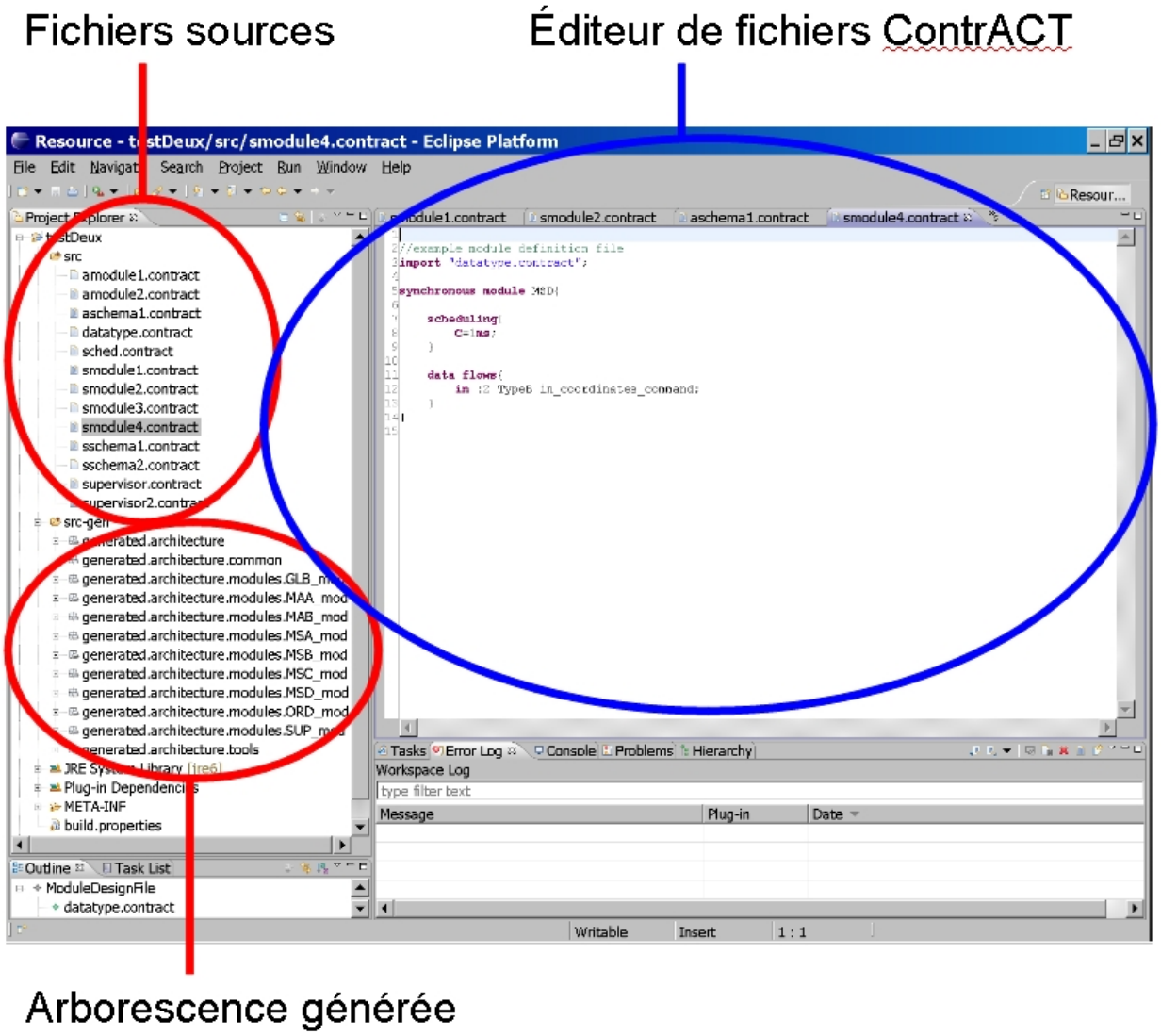

Figure 3: Vue globale sur l'organisation d'un projet dans l'outil de développement

- *Le fichier de définition de la configuration de l'ordonnanceur*. Ce fichier unique permet de répertorier l'ensemble des schémas périodiques et des modules synchrones connus par l'ordonnanceur. Il sert à la globalisation des informations nécessaires à la génération du module ordonnanceur.

- *Les fichiers de définition des superviseurs*. Chaque fichier correspondant à la définition d'un **module superviseur** soit : son interface, constituée de ses ports d'événement en entrée et en sortie ; son comportement constitué d'un ensemble de fonctions de supervision définissant l'ensemble des objectifs qu'il est capable de réaliser. L'utilisation du langage *Cactal* pour décrire les superviseurs est détaillée dans la sous-section suivante. La génération de code utilisera les informations contenues dans un tel fichier pour **générer le code complet des modules superviseurs**.

Les sections suivantes présentes en détail l'aspect édition et l'aspect génération de code de l'outil. Les questions relatives à la conception de l'outil en lui-même trouvent leurs réponses dans le rapport présentant la méthodologie ***[Passama 2010]***.

4) Édition d'une architecture

L'édition d'une architecture de contrôle passe en premier lieu par la création d'un projet. Ce projet est créé de la même manière que n'importe quel projet ECLIPSE en utilisant la commande **File > New > Project...** et en sélectionnant le type de projet **Contract Project** (cf. fig 4.) dans la catégorie **Contract**. Une boite de dialogue demande à l'utilisateur d'entrer le nom du projet et créé ensuite le dossier du projet avec des fichiers. Ces fichiers correspondent à du code source dans le langage Cactal et servent d'exemple pour l'utilisateur:

- **datatype.contract** est un exemple de fichier de définition de type.
- **module1.contract** et **module2.contract** sont des exemple de définition de modules synchrones.
- **sched.contract** est un fichier exemple d'une configuration de l'ordonnanceur.
- **schema.contract** est un fichier exemple de définition d'un schéma périodique.
- **supervisor.contract** est un fichier exemple de définition d'un superviseur.

Pour créer de nouveaux fichiers, il faut simplement créer un nouveau fichier via ECLIPSE (**File > New > File**) et lui associer l'extension `.contract`. Dans les sous-sections suivantes nous détaillons chaque types de définition possible dans l'outil.

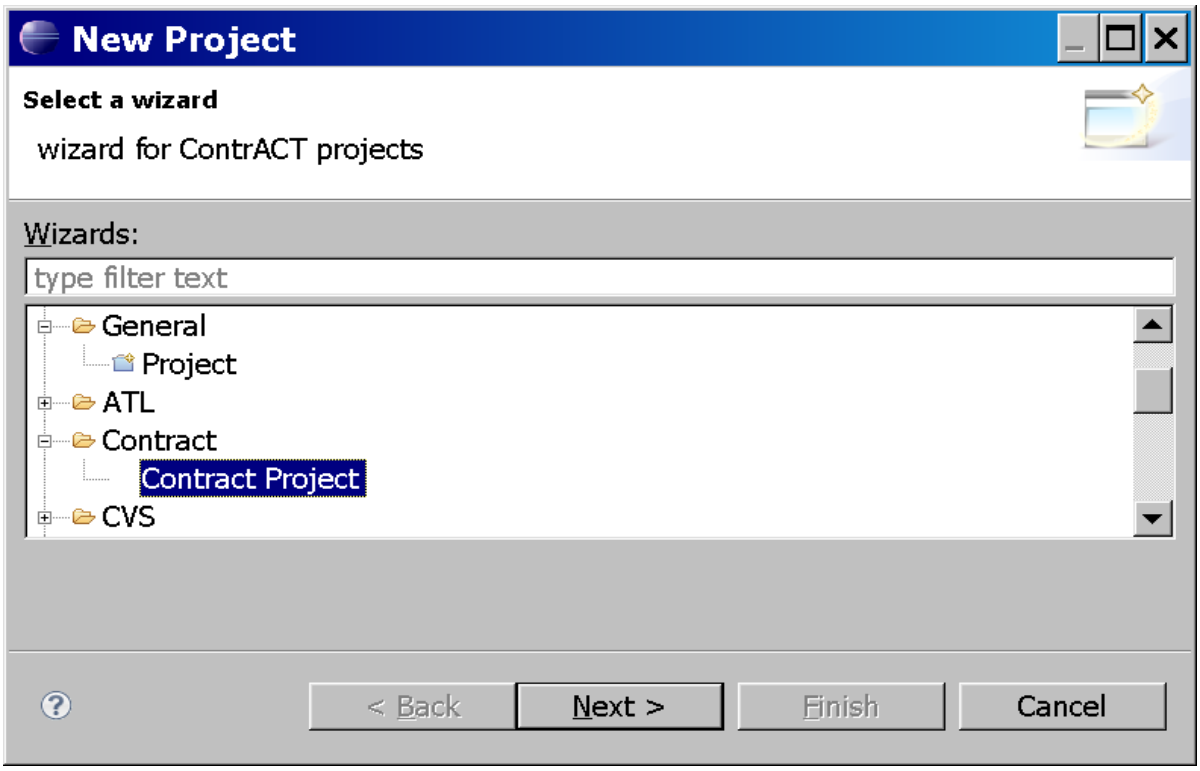

Figure 4 : Création d'un projet ContrACT

4.1) Définition de type de données et de prototypes

Les fichiers de définitions de données servent à décrire les types de données échangés entre les modules de l'architecture, à travers les ports de données, d'événements ou de paramètres. Ces fichiers peuvent également contenir la déclaration de prototypes de fonctions qui pourront être utilisées au niveau des superviseurs. Voici un exemple de code d'un fichier de déclaration de type:

```
//example datatype definition file
datatype TypeA {float i = 0.0, float  j =0.0, float  k=0.0};
datatype TypeB {TypeA coord = {1.0,1.0,1.0}, int  frfrrrf=1};
datatype TypeC {string message, int  messagetype=0, TypeA value };
datatype TypeD {string dataname, int datavalue};
datatype TypeE {TypeC message, TypeD data_ };
datatype TypeF {int fufu, int juju};
//example function prototypes
function bool testit(string arg1, TypeA arg2);
function void doit(string argfirst, float argsecond);
```

Exemple 1 : code de déclaration de types de donnée

Dans cet exemple, l'utilisateur déclare des types de données nommés `TypeA`, `TypeB`, etc. Ces types de données sont construit à partir de types primitifs (actuellement **`int`**, **`float`** et **`string`** sont disponibles) et/ou à partir de type construits préalablement définis. Ces derniers peuvent être soit défini dans le même fichier (exemple `TypeB` défini à partir de `TypeA`), soit défini dans un autre fichier de définition de type qui aura été importé en utilisant le mot-clef **`import`** en début de fichier:

```
import "othertypes.contract";
```

Comme le montre l'exemple de code précédent les variables internes d'un type de données peuvent être initialisées en utilisant l'opérateur =. Les variables de types primitifs peuvent être initialisées via l'utilisation de valeur correspondantes (une valeur entière pour un **`int`**, une valeur réelle pour un **`float`** et une chaine de caractère entre guillemet pour une **`string`**). Les variables de type construit peuvent être initialisées via l'utilisation de valeur « structurées » (utilisant les accolades `{}` pour délimiter la structure). Une vérification des valeurs a lieu au regard du type de la variable pour déterminer si la valeur est conforme au type et l'outil reportera dans sa fenêtre **Problems** les erreurs de type rencontrées. Notons qu'une variable non initialisée prendra la valeur par défaut de son type: 0 pour les **`int`**, 0.0 pour les **`float`** et la chaine vide pour les **`string`**, la valeur par défaut définie dans le type de donnée structuré pour les types définis par l'utilisateur. Dans le code présenté ci-dessus, on voit que la variable `coord` de type `TypeA` voit l'ensemble de ses champs internes initialisés par l'utilisateur, alors que la variable `message` de `TypeC` prend la valeur par défaut associée à son type soit la chaine vide.

Notons que pour le type de donnée **`string`**, la taille maximale est de **256 cractères**.

Dans l'exemple on voit que deux prototypes de fonctions sont définies, qui correspondent aux deux cas possibles :

- Les fonctions de calcul et d'affectation (retournant **`void`**) sont utilisées par les superviseurs pour effectuer des calcul et/ou faire des affectations directement au niveau des superviseurs.

- Les fonctions de test (retournant **bool**) sont utilisées par les superviseurs pour effectuer des tests sur des données au moment de la réception d'événements.

Ces fonctions doivent recevoir une implémentation en C après que le prototype de fonction C correspondant ait été généré.

L'utilisateur peut définir autant de fichiers de définition de type de données qu'il le souhaite. Une fois ceci fait, il peut utiliser ces définitions pour la description des modules.

4.2) Définition d'un module synchrone

L'environnement ContrACT permet, via l'utilisation du langage Cactal, de décrire l'interface des modules utilisés et notamment des modules synchrones, comme le montre l'exemple de code suivant.

```
//example module definition file
import "datatype.contract";

synchronous module MSC{
          parameters{
                    TypeA param = {2.1,6.98,5.34} :0;
          }
          scheduling{
                    C=1ms;
          }
          data flows{
                    in :0 TypeA in_coordinates;
                    out :1 TypeB out_coordinates_commmand;
          }
          event flows{
                    out :1 TypeD failing_data;
          }
}
```

Exemple 2 : code de définition d'une interface de module synchrone

Un module synchrone est défini via le mots clef **synchronous modules** suivi du nom du module. Une contrainte à respecter est que ce nom est créé à partir de **trois caractères majuscule** exactement. La définition de l'interface du module est précédée d'un ensemble d'import pour importer les définitions des types de données structurés utilisés pour définir le module.

La description de l'interface du module consiste avant tout à spécifier l'ensemble des interactions possibles du module avec le reste de l'architecture au travers de la description de ses différentes ports:

- **Ports de paramètres** décrits dans le bloc **parameters**. Chaque paramètre public (i.e. modifiable depuis l'extérieur) du module est décrit par un type , un nom et un index de port de paramètre, dans cet ordre (e.g. `TypeA param :0;`). Les index de ports de paramètres sont nécessairement **uniques pour l'ensemble des paramètres du module**. L'utilisateur peut également définir une initialisation des paramètres (appliquée à l'initialisation du module) qui suit la même syntaxe que pour l'initialisation des variables d'un type de données.
- **Ports de données** décrits dans le bloc **data flows**. Chaque port de donnée est décrit syntaxiquement par une direction (**in** pour un port de réception de donnée en entrée ou **out** pour un port d'émission de données en sortie du module), un index de port de données, un

type et un nom de variable (e.g. `out :1 TypeB out_coordinates_commmand`). Les index de ports de données **`in`** (respectivement **`out`**) sont nécessairement **uniques pour l'ensemble des ports de données `in` (respectivement `out`) du module**. L'utilisateur peut définir une initialisation des variables associées aux ports de données (appliquée à l'initialisation du module) qui suit la même syntaxe que pour l'initialisation des variables d'un type de données.

- **Ports d'événements** décrits dans le bloc **`event flows`**. Chaque port d'événement est décrit syntaxiquement par une direction (**`out`** est la seule direction autorisée pour les modules synchrones), un index de port de données, un type et un nom de variable (e.g. `out :1 TypeD failing_data`). Les index de ports d'événements sont nécessairement **uniques pour l'ensemble des ports d'événements du module**. L'utilisateur peut définir une initialisation des variables associées aux ports d'événements (appliquée à l'initialisation du module) qui suit la même syntaxe que pour l'initialisation des variables d'un type de données.

Notons que l'ensemble des noms des variables définies dans l'interface d'un module sont **nécessairement uniques pour tout le module**. Notons également que l'ordre de définition des blocs est imposé : **`parameters`**, **`scheduling`**, **`data flows`**, **`event flows`**.

Les modules synchrones requièrent également que l'utilisateur fixe leurs propriétés liées à l'ordonnancement. Il s'agit notamment de définir la durée nominale d'exécution des modules. Pour cela il faut fixer le paramètre **`C`** définit dans le bloc **`scheduling`**. Dans l'exemple **`C`**`=1`**`ms`**`;` .indique que la durée nominale d'exécution est de une milliseconde par cycle d'exécution. A l'heure actuelle, c'est le seul paramètre qui peut être fixé. Dans les prochaines évolutions de l'outil et du middleware, d'autre paramètres liés à l'ordonnancement seront disponibles pour contrôler de manière différente l'ordonnancement des modules.

4.3) Définition d'un module asynchrone

Un module asynchrone est défini via le mots clef **`asynchronous modules`** suivi du nom du module, comme pour les modules synchrones.

```
//example module definition file
import "datatype.contract";

asynchronous module MAA{
          parameters{
                    TypeA param = {22.12,6.98,897.34} :0;
          }
          event flows{
                    in :0 string controlevent;
                    out :0 TypeC message_type_notification;
                    out :1 TypeE received_message;
                    out :2 string connection_status;
          }
}
```

Exemple 3 : code de définition d'une interface de module asynchrone

La définition de l'interface d'un module asynchrone suit la même logique et les mêmes contraintes que celle d'un module module synchrone, aux exceptions suivantes près:

- un module asynchrone ne peut pas définir de ports de données (pas de bloc **`data flows`**).
- un module asynchrone n'étant pas ordonnancé, il ne possède pas de paramètres lié à l'ordonnancement (pas de bloc **`scheduling`**).
- un module asynchrone peut définir des ports d'événements en entrée (**`in`**) et en sortie (**`out`**). Les index de ports d'événements **`in`** (respectivement **`out`**) sont nécessairement **uniques pour l'ensemble des ports d'événements `in` (respectivement `out`) du module**.

4.4) Définition d'un schéma périodique

Les schémas servent à définir des assemblages de modules. Les schémas périodiques sont utilisé pour définir des schémas dont les modules contenus ont pour vocation à être ordonnancés, c'est-à-dire un assemblage de module synchrones. Ils permettent par exemple d'implémenter des asservissements, des boucles d'observation ou des boucles de diagnostic, en assemblant les différents modules implémentant le recueil de données perceptives depuis les capteurs, l'application de commandes aux actionneurs ainsi que tous les algorithmes (e.g. traitement du signal, loi de commande, estimateurs, etc.) mis en jeu dans le schéma.

Les schémas périodiques sont déclarés via le mot clef **`periodic schema`** suivit par le nom du schéma (**unique pour l'ensemble des schémas** définis dans l'architecture) et la liste de ses paramètres, comme le montre l'exemple 4.

Le fichier doit comporter en entête l'ensemble des **`import`** nécessaires, c'est-à-dire:

- Les import des fichiers contenant la définition des modules synchrones utilisés dans le schéma.
- Les import des types de données utilisés dans le schéma.

Les noms des paramètres du schéma sont unique pour le schéma et ils peuvent être initialisés comme c'est le cas pour les variables des modules (e.g. `consigne = {0.0, 0.0, 0.0}`). Chaque fois que le schéma est exécuté, chaque paramètre prendra sa valeur d'initialisation (i.e. par défaut) si l'appel demandant son exécution ne fixe pas de valeur à ce paramètre.

Un schéma périodique déclare un ensemble de blocs qui doivent suivre l'ordre suivant : **`realtime`**, **`modules`**, **`scheduling graph`**, **`communications`**.

Le bloc **`realtime`** contient l'ensemble des paramètres nécessaires à l'ordonnancement temps-réel du schéma périodique, à savoir:

- sa période de répétition (**`PERIOD`**)

son délai critique (**`CRITICAL_DELAY`**) qui définit le délai maximal laissé pour que l'ensemble des modules contenus dans le schéma aient terminé leur exécution sur un cycle d'ordonnancement.

```
import "datatype.contract";
import "smodule1.contract";
import "smodule3.contract";
import "smodule4.contract";

periodic schema schemaCommande1(TypeA consigne = {0.0, 0.0, 0.0}){
          realtime{
                    PERIOD=1s;
                    CRITICAL_DELAY=1s;
          }
          modules{
                    MSA ();
                    MSC (consigne:0);
                    MSD ();
          }
          scheduling graph{
                    MSA -> MSC;
                    MSC -> MSD;
          }
          communications{
                    MSA:0 -> MSC:0 * 1;//data flow communication
                    MSC:1 -> MSD :2 * 2;//data flow communication
          }
}
```

Exemple 4 : code de définition d'un schéma périodique

Le bloc **`realtime`** pourra dans le futur contenir des informations additionnelles, notamment en relation avec la politique de réactions aux fautes temps-réel ou de nouveaux paramètres si un un algorithme d'ordonnancement alternatif est possible.

Le bloc **`modules`** contient des références vers tous les modules synchrones utilisés dans le schéma. A chaque module, le schéma peut également définir son paramétrage, c'est-à-dire les valeurs fixées pour tout ou partie de ses paramètres appliqués avant l'ordonnancement du schéma. Le paramétrage consiste à fixer une valeur (valeur constante ou **issue d'un paramètre du schéma**) pour un port de paramètre donné du module concerné (e.g. `MSC(consigne:0);`). L'exemple de code ci-dessus défini par exemple que le paramètre de port `0` du module `MSC` prend la valeur du paramètre `consigne` du schéma.

Le bloc **`scheduling graph`** définit le **graphe de précédence** du schéma périodique, c'est-à-dire une contrainte sur l'ordre d'exécution des modules sur une période. Ce graphe est un graphe orienté obligatoirement **non-cyclique**, définit à partir d'un ensemble de règle de précédence entre module (e.g. `MSA -> MSC` impliquant que le module `MSA` s'exécute avant le module `MSB`). Le graphe de précédence. Ce graphe est utilisé par l'ordonnanceur pour produire un ordonnancement correct des modules du schéma.

Le bloc **`communications`** définit l'ensemble des liens de communication entre les modules synchrones. Pour un schéma périodique, les liens de communication sont des **flux de données uniquement**. Un lien de communication décrit la mise en relation d'un port de donnée en sortie d'un module avec un port de donnée en entrée d'un autre module. Par exemple `MSA:0 -> MSC:0 * 1` décrit la mise en relation du port producteur (sortie) `0` du module `MSA` avec le port consommateur (entrée) `0` du module `MSC`. A chaque lien de communication est associée une périodicité (e.g. « `*`

`1` » pour le lien précédent) qui définit tous les combien de cycles la donnée sera produite par le producteur vers le consommateur. Pour le lien `MSA:0 -> MSC:0 * 1` la donnée sera produite tous les cycles. Pour le lien `MSC:1 -> MSD :2 * 2` la donnée sera produite un cycle sur deux, etc.

4.5) Définition d'un schéma événementiel

Les **schémas événementiels** sont utilisés pour décrire des compositions de modules asynchrones **interagissant uniquement à travers leurs ports d'événements**. Ces modules (dits d'interaction) servent par exemple à interagir avec des systèmes externes via un réseau (e.g. poste opérateur autre robot) ou via des primitives système ou middleware (e.g. interface graphique, entrée/sorties événementielles comme certains périphériques utilisateur).

Les schémas événementiels sont déclarés via le mot clef **`event schema`** suivit par le nom du schéma (**unique pour l'ensemble des schémas périodiques ou événementiels** définis dans l'architecture) et la liste de ses paramètres, comme le montre l'exemple 5.

Le fichier doit comporter en entête l'ensemble des **`import`** nécessaires, c'est-à-dire:

- Les import des fichiers contenant la définition des modules asynchrones utilisés dans le schéma.
- Les import des types de données utilisés dans le schéma.

```
import "datatype.contract";
import "amodule1.contract";
import "amodule2.contract";

event schema schemaReceptionMessage(TypeA param = {27.12,0.0,0.04}){

        modules{
                MAA (param :0);
                MAB ();
        }

        communications{
                MAA:0 =* MAB:0;//continuous event flow communication
        }
}
```

Exemple 5 : code de définition d'un schéma événementiel

Globalement un schéma événementiel possède les mêmes propriétés (paramètres, organisation des blocs) la même structure qu'un schéma périodique hormis les différences suivantes:

- il ne peut pas déclarer de bloc **`realtime`** et **`scheduling graph`**, puisqu'il n'est pas ordonnancé.
- Les liens de communications sont des flux événementiels et pas des flux de données. Les index des ports référencés correspondent aux **index des ports d'événements** disponibles. Les flux d'événements ne sont pas décrits de la même manière que les flux de données :

l'utilisateur ne doit plus indiquer la périodicité de production mais le mode de connexion entre les deux couples modules-ports. Le mode « un coup » (symbole => entre les deux couples) indique que la connexion de ports sera interrompue immédiatement (et automatiquement) après notification de l'événement. Le mode « continu » (symbole =* entre les deux couples, cf. exemple 5) indique que la notification se fera tant qu'une déconnexion explicite n'a pas été demandée.

4.6) Définition de la configuration de l'ordonnanceur

Le fichier de configuration de l'ordonnanceur est celui qui contient toutes les informations pour configurer l'ordonnanceur, c'est-à-dire principalement les références vers l'ensemble de modules synchrones et vers l'ensemble de schémas périodiques qu'il est susceptible d'ordonnancer.

La configuration de l'ordonnanceur est déclarée dans un fichier unique, via le mot clef **`scheduler configuration`**. comme le montre l'exemple 6.

```
import "datatype.contract";
import "smodule1.contract";
import "smodule2.contract";
import "smodule3.contract";
import "smodule4.contract";
import "sschema1.contract";
import "sschema2.contract";

scheduler configuration {

          modules{
                    MSA;
                    MSB;
                    MSC;
                    MSD;
          }

          schemas{
                    schemaCommande1;
                    schemaCommande2;
          }
}
```

Exemple 6 : code de configuration de l'ordonnanceur

A l'heure actuelle la configuration contient deux blocs:

- le bloc **`modules`** contient les références vers tous les modules synchrones connus par l'ordonnanceur et utilisés au sein des schéma périodiques.
- Le bloc **`schemas`** contient les références vers tous les schémas périodiques que l'ordonnanceur est capable d'ordonnancer.

Ce fichier sert uniquement à la centralisation des informations nécessaires à la génération du code de l'ordonnanceur.

4.7) Définition d'un superviseur

```
import "datatype.contract";import "smodule1.contract";
import "smodule2.contract";import "smodule3.contract";
import "smodule4.contract";
import "amodule1.contract";
import "amodule2.contract";
import "sschema1.contract";
import "sschema2.contract";
import "aschema1.contract";
import "sched.contract";

supervisor SUP{
event flows{
          in :1 TypeC message_received;
          in :2 string connection_status;
          in :3 TypeD failure;
          out :1 string failure_message_type;
}
action(TypeA consigne){
          local{
          TypeC temp_message;
          string temp_mess_type;
          TypeD failure_data;
          TypeA temp_consigne = consigne;
          }
          rules{
          FIRST:[elapsed=1ms]
                    subscribe MAA:2 =* :2;subscribe MAA:0 =* :1;
                    activate schema schemaReceptionMessage()
                [MAA:2<connection_status == "disconnected">]

          SECOND:[(MAA:2<(connection_status == "disconnected"
                    & testit("dedede",temp_consigne))>
                  and started(FIRST){infinite})]
                    subscribe MSC:1=>:3;
                    activate schema schemaCommande1(consigne = temp_consigne)
                  [MSC:1(failure_data = failure, temp_mess_type = "commutation")]

          THIRD:[endof(SECOND)]
                    notify :1(failure_message_type=temp_mess_type)
                [started(THIRD)]

          FOURTH:[endof(SECOND)]
                    compute doit("1.24",temp_consigne.k);
                    activate schema schemaCommande2(consigne = temp_consigne)
                  [endof(FIRST)]


          RECEPTION:[MAA:0(temp_message=message_received)]
                       parameterize MSC(temp_message.value :0);
                       activate resttime module MAA
                    [started(RECEPTION)]

          }
}
//other supervision functions
}
```

Exemple 7 : code d'un superviseur

L'environnement logiciel permet de définir à la fois l'interface (i.e. ports d'événements et prototype des fonctions de supervision) et le comportement (i.e. implémentation des fonctions de supervision), comme le montre l'exemple 7. Chaque superviseur est définit dans un fichier séparé via l'utilisation du mot clef **`supervisor`** suivit du nom du module, qui est lui-même constitué de tris caractère majuscule exactement (même contrainte que pour les modules). En entête le fichier contient tous les **`import`** nécessaires au superviseur, c'est-à-dire les types de données manipulés, les schémas, modules et superviseur utilisés (cf. exemple 7).

L'interface d'un superviseur est définie par:

- Un ensemble de ports d'événements en entrée et en sortie déclarés dans le bloc **`event flows`**, de la même manière que pour les modules asynchrones.
- Un ensemble de fonction de supervision, déclarées à la suite du bloc **`event flows`**. Chaque fonction de supervision est définie par un nom unique pour l'ensemble du superviseur et un ensemble (potentiellement vide) de paramètres formels nommés de façon unique et typés. Notons que le nom des paramètres fait partie intégrante de l'interface de la fonction. En effet au moment de l'appel de la fonction, l'appelant peut fixer des valeurs à tout ou partie des paramètres en désignant leur nom. Dans l'exemple 7 on voit que la fonction de supervision `action` a un seul paramètre formel nommé `consigne` qui prend la valeur par défaut associé à son type si sa valeur n'a pas été fixée au moment de l'appel. L'utilisateur peut tout comme pour les paramètres des modules et schémas, fixer une valeur d'initialisation à ces paramètres.

L'apport majeur du langage Cactal est de permettre la description complète des comportements des superviseurs via une syntaxe et une sémantique claire. Les détails sur la manière dont est exécuté le comportement des superviseurs est explicité dans ***[Passama 2010]***. Le présent document décrit essentiellement la syntaxe et la sémantique du langage de description des comportements, c'est-à-dire la manière de définir les fonctions de supervision.

L'implémentation d'un fonction de supervision se fait en deux blocs:

- le bloc **`local`** définit l'ensemble des variables locales utilisées pour stocker des valeurs « intermédiaires ». Chaque variable locale est définit par un type et un nom unique (i.e. différents des autres variables et paramètres de la fonction, et différents des variables associées aux ports d'événements). Une variable locale peut être initialisée à partir de valeurs constantes ou à partir de valeurs des paramètres (e.g. `temp_consigne =consigne`) ou une combinaison des deux pour les variables de type structuré.
- Le bloc **`rules`** définit l'ensemble des règles qui implémentent effectivement le comportement de la fonction. Ces règles peuvent utiliser uniquement les variables locales de la fonction et pas ses paramètres.

Nous détaillons maintenant la description des règles. Chaque règle est structurée de la manière suivante: *label: [précondition] séquence_actions [postcondition]*.

- Le **label** est un nom unique pour l'ensemble des règles d'une fonction de supervision qui permet d'identifier la règle.
- La **précondition** est le test à effectuer pour savoir si une règle inactive doit devenir active.
- La **postcondition** est le test à effectuer pour savoir si une règle active doit devenir inactive. Les pré et post conditions suivent exactement la même syntaxe.
- La **séquence d'actions** est la séquence d'action primitives à effectuer (respectivement à

annuler, si possible) lorsque la règle devient active (respectivement inactive).

Nous détallons maintenant la syntaxe et la sémantique des **conditions** et des **actions primitives**.

La description des conditions suit syntaxiquement la structure définit par la grammaire BNF simplifiée suivante:

```
Condition: Clause ('or' Clause)* ;

Clause: Event | '(' Event ('and' Event)+ ')';

Event: RuleEvent | AbsoluteTimeEvent | ModuleEvent;

RuleEvent: 'started''('RuleIndex')' ('{' Persistancedescripion'}')?
|
'endof''('RuleIndex')' ('{'Persistancedescripion'}')?
|
'duration''(' RuleIndex ')''=' time=Time ('{'Persistancedescripion'}')?;

AbsoluteTimeEvent: 'elapsed''=' Time ('{'Persistancedescripion'}')?;

ModuleEvent: BasicEvent ('<' EventConditions '>')?
('{'Persistancedescripion'}')? ('(' EventExtractions ')')?;

EventConditions: EventClause ('|' EventClause)*;

EventClause: EventCondition | '(' EventCondition ('&' EventCondition)+ ')';

EventCondition: CallArgument ConditionOperator CallArgument| TestFunctionCall;

EventExtractions: VariableExtraction (',' VariableExtraction)*;

Persistancedescripion: Time | 'infinite';

...
```

Exemple 8 : extrait simplifié de la BNF du langage Cactal

Chaque **condition** est constituée d'une disjonction de clause d'événements. La disjonction (OU logique) est déterminée en utilisant le mot clef **`or`** entre chaque clause (s'il y a plus d'une seule clause dans la condition, cf. règle **`Condition`** fig. 8). Une clause (ET logique) est elle-même une conjonction d'événements décrite via l'utilisation du mot clef **`and`** (s'il y a plus d'un événement dans la clause, cf. règle **`Clause`** fig. 8) et entourée de parenthèses si plus d'un événement appartiennent à la clause (cf. règle **`Clause`** fig. 8). Un événement représente l'**occurrence d'un phénomène d'intérêt** pour le superviseur, il peut être de différents types (cf. règle **`Event`** fig. 8):

- Les **événements de module** (cf. règle **`ModuleEvent`**, fig 8.) correspondent aux événements produits par d'autres modules (y compris des superviseurs). Un cas spécifique sont les événements produits par le module ordonnanceur, auxquels tous les superviseurs sont automatiquement abonnés.

- Les **événements de temps absolu** (cf. règle **`AbsoluteTimeEvent`**, fig 8.) correspondent à l'écoulement d'une quantité de temps donnée depuis l'activation de la fonction (**`elapsed=`**).

- Les **événements de règle** (cf. règle **`RuleEvent`**, fig 8.) correspondent à des événements internes au superviseur qui apparaissent dès lors qu'une règle devient active (**`started`**) ou inactive (**`endof`**) ou que le temps d'exécution .de cette règle a atteint une quantité donnée (**`duration`**). La règle ciblée est identifiée par son **label**.

A chaque événement peut être associé un temps de persistance (cf. règle **`Persistancedescripion`**, fig. 8) qui peut être infini. Ce temps permet de spécifier la durée pendant laquelle l'événement, une fois notifié, reste pertinent du point de vue du superviseur. Dès lors qu'une clause multi-événements existe, tous ses événements (sauf éventuellement un seul) doivent définir une durée de persistance, sans quoi la clause ne pourrait jamais être vraie puisqu'il n'existe pas de simultanéité dans un monde purement événementiel.

Un événement de module (**`ModuleEvent`**) est défini à partir d'un nom de module suivit d'un numéro de port (e.g. règle `FIRST`, événement `MAA:2` dans la postcondition, fig. 7). Les événements de modules peuvent se voir associer des opérations supplémentaires:

- un **test** (cf. règle **`EventConditions`**, fig. 8) peut être effectué au moment de la réception de l'événement pour déterminer si l'événement est **significatif** ou pas. Si l'événement n'est pas significatif il sera considéré comme « non produit » et ne pourra donc pas être enregistré pour sa persistance. Par exemple l'expression `MAA:2<connection_status == "disconnected">` définit que l'événement issu du port d'événement en sortie d'index `2` du module `MAA` sera considéré comme significatif seulement si sa donnée associée (reçue dans la variable associée au port d'événement en entrée d'index `3` du superviseur, à savoir `connection_status`) satisfait le test (dans l'exemple que la valeur de la string soit `"disconnected"`). Le test associé à un événement est décomposé logiquement en une disjonction de clauses d'expression booléennes, via l'utilisation des mots-clef **`&`** (ET logique) et **`|`** (OU logique) de manière similaire que pour les conditions. Les expressions booléennes atomiques (cf. règle **`EventCondition`**, fig. 8) sont construites, soit à partir de l'utilisation d'opérateurs de bases (**`ConditionOperator`**) disponibles sur les types primitifs et les types structurés (==, =, <=,>=,<,>) sur des variables ou constantes (**`CallArgument`**), soit via l'utilisation de fonctions de tests (**`TestFunctionCall`**) définie dans un des fichiers de définition de type (cf. section 4.1), qui permettent d'implémenter des tests plus complexes.

- une **extraction** (cf. règle **`EventExtractions`**) peut-être réalisée **au moment ou une clause devient vraie** pour affecter aux variables locales des valeurs extraites **des événements constituant la clause**. Cette extraction est vue comme une séquence d'extraction de variables (**`VariableExtraction`**) qui chacune correspond à l'affectation de la valeur d'une variable locale (ou un de ses champs si c'est une variable de type structuré) à partir d'une valeur constante, d'une valeur extraite de la donnée associé à l'événement ou d'une combinaison des deux pour affecter une valeur à un type structuré. Par exemple, l'expression `MSC:1(failure_data = failure, temp_mess_type = "commutation")`,de la règle `SECOND` indique que la variable locale `failure_data` voit sa valeur affecté à celle de la variable associée au port d'événement en entrée 3 (à savoir `failure`) et que la variable locale `temp_mess_type` voit sa valeur affecté à la valeur

constante `"commutation"`. Le mécanisme d'extraction est essentiel au transfert de données entre les informations perceptives du superviseur (événements) et ses sorties (événements, activation de schémas, etc. - voir plus loin).

Une fois qu'une règle a été rendue active par sa précondition, elle exécute sa séquence d'**actions primitives**, l'opérateur `;` servant à décrire séparer les instruction et à les séquencer (action à gauche de l'opérateur s'exécute avant l'action à sa droite). Si elle est rendue inactive par sa postcondition elle annule (si possible) chaque action de sa séquence dans le même ordre imposé par l'opérateur `;`. Les actions primitives existantes sont les suivantes:

- Lancement (activation de règle) ou annulation (désactivation de règle) d'un **schéma périodique ou événementiel**. , via l'utilisation du mot clef **`activate schema`** suivit du nom du schéma considéré et de la liste des paramètres de ce schéma que le superviseur décide d'initialiser. Par exemple, l'expression « **`activate schema`** `schemaCommande1(consigne = temp_consigne)` » définit l'activation ou l'annulation du schéma périodique de nom `schemaCommande1` dont le paramètre `consigne` voit sa valeur initialisée avec la valeur de la variable locale `temp_consigne`. Du point de vue du langage il n'y a pas de distinction entre les schémas périodiques et événementiels bien que leur fonctionnement soit fondamentalement différents.

- **Paramétrage en ligne** d'un module synchrone ou d'un module asynchrone (ne peut être annulé dans le cas de la désactivation de règle), via l'utilisation du mot clef **`parameterize`** suivit du nom du module cible et de la liste des paramètres affectés. L'affectation d'une valeur à un paramètre revient à indiquer la valeur (constante ou variable) à affecter et un index de port de paramètre du module dont le type est concordant. Par exemple, l'expression « **`parameterize`** `MSC(temp_message.value :0)` » définit le paramétrage du paramètre de port `0` du module `MSC` qui prend la valeur du champ `value` de la variable locale `temp_message`.

- Lancement (activation de règle) ou annulation (désactivation de règle) d'une **fonction de supervision**, via l'utilisation du mot clef **`activate supervisor`** suivit du nom du module, du nom de la fonction et (éventuellement) de l'affectation de valeurs aux paramètres de la fonction correspondante. Par exemple, l'expression « **`activate supervisor`** `SUP.action(consigne=consigne_courante)` » (cf. section 8.1 décrivant l'exemple complet) définit l'activation ou l'annulation de la fonction `action` du module superviseur `SUP` dont le paramètre `consigne` prend la valeur de la variable locale `consigne_courante`.

- Lancement (activation de règle) ou annulation (désactivation de règle) d'un **calcul en parallèle**, via l'utilisation des mots clefs **`activate resttime module`** ou **`activate critical module`**, suivit du nom du module **asynchrone** concerné. Le calcul en parallèle est supporté par le module concerné qui est ainsi lancé ou arrêté par le superviseur. La différence entre **`resttime`** et **`critical`** consiste simplement dans la priorité conférée au module : avec **`resttime`** le module est exécuté avec la priorité la plus faible

du système ce qui revient à définir le fait qu'il s'exécute en temps restant sans remettre en cause la stabilité des éventuels asservissements (schémas périodiques exécutés), avec **`critical`** le module est exécuté avec une priorité plus haute que tout module synchrone ce qui suppose qu'il effectue un traitement critique qui doit être effectué quitte à remettre en cause la stabilité des éventuels asservissements.

- Exécution d'un **appel critique**, via l'utilisation du mot clef **`compute`** suivit du nom de la fonction de calcul (cf. section 4.1). Cette action revient pour le superviseur a effectuer un appel de fonction local prenant en paramètres certaines de ses **variables locales**. Par exemple , l'expression « **`compute`** `doit("1.24",temp_consigne.k)` » définit un appel de la fonction `doit`, l'appel se faisant de la même manière qu'en C. Cette action ne peut être annulée dans le cas de la désactivation de règle puis qu'elle bloque le fonctionnement du superviseur jusqu'à ce quelle se termine. Elle doit donc être utilisée avec précaution, pour effectuer des calculs « rapides », car pendant son exécution le superviseur n'est plus réactif aux événements.

- Abonnement (activation de règle) ou désabonnement (désactivation de règle) à un **flux de données d'événements**, via l'utilisation du mot clef **`subscribe`**. Cette action consiste pour le superviseur à établir une connexion (respectivement à détruire une connexion pour la désactivation de règle) entre un de ses ports d'entrée d'événements et un port de sortie d'événement d'un module de la couche exécutive ou d'un autre module superviseur. Par exemple, «**`subscribe`** `MAA:2 =* :2`» dans la règle `FIRST` établit une connexion entre le port d'événement en entrée du superviseur d'index `2` avec le port d'événements en sortie d'index `2` du module `MAA` suivant un mode de diffusion en continu (utilisation de l'opérateur de connexion =*). «**`subscribe`** `MSC:1=>:3`» dans la règle `SECOND` établit une connexion entre le port d'événement en entrée du superviseur d'index `3` avec le port d'événements en sortie d'index `1` du module `MSC` suivant un mode de diffusion « un coup » (utilisation de l'opérateur de connexion =>). Tant que le superviseur n'est pas abonné à un événement d'un module, toutes les clauses d'événements (dans les pré ou post conditions) incluant cet événement ne pourront être **vraies**. Dès que des ports d'événements sont connectés, la variable associé au port d'entrée du superviseur se verra affecter la valeur de la donnée associé à l'événement **au moment de sa notification**.

- **Notification d'un événement**, via l'utilisation du mot clef **`notify`**. Cette action consiste pour le superviseur à notifier, via un de ses ports de sortie d'événements, l'occurrence d'un événement qu'il a détecté. Par exemple, l'expression « **`notify`** `1(failure_message_type=temp_mess_type)` » dans la règle `THIRD`, spécifie que le superviseur notifie un événement sur son port de sortie d'événement d'index `1`. Elle spécifie également que la variable associée à l'événement, à savoir `failure_message_type`, se voit affectée la valeur de la variable locale `temp_mess_type`. Cette affectation peut être décomposée en une séquence d'affectations des champs de la variable d'événement. Cette action ne peut être annulée dans le cas d'une désactivation de règle car la notification est instantanée et unique dès lors que la règle devient active.

Notons qu'il y a certaines bonnes pratiques dans la définition des règles:

- L'événement de règle **`endof`** ne doit pas être utilisé dans la **postcondition** de la règle qu'il référence, car il ne pourra jamais être vrai. En effet, puisque la postcondition définit le contexte de désactivation de la règle, elle ne peut se baser sur un événement consécutif à cette désactivation. Exemple incorrect:
  `REGLEA:[...]actions[endof(REGLEA)].`

- De manière symétrique, les événements de règle **`started`** et **`duration`** ne doivent pas être utilisés dans la **précondition** de la règle qu'il référence, car il ne pourra jamais être vrai. En effet, puisque la précondition définit le contexte d'activation de la règle, elle ne peut se baser sur un événement consécutif (immédiatement pour **`started`**, après une durée pour **`duration`**) à cette activation. Exemple incorrect:
  `REGLEB:[started(REGLEB)]actions[...].`

- L'utilisation du temps de persistance pour les événements d'une clause peut être évitée pour **un seul événement uniquement** appartenant à la clause. Implicitement, cela signifie que cet événement sera nécessairement **le dernier notifié au superviseur** dans le cas ou la clause devient vraie. En effet, n'ayant pas de temps de persistance, si la clause est fausse après réception de cet événement, alors celui-ci sera « oublié ». De fait, la notification d'un des autres événements de la clause ne pourra être à l'origine de sa validation, puisque l'événement non persistent n'est jamais enregistré mais traité immédiatement puis oublié.

4.8) Conclusion sur l'éditeur

L'éditeur de code du langage Cactal ayant été créé à partir des technologies MDE d'ECLIPSE, il réutilise une grande partie des fonctionnalités existantes dans ECLIPSE pour la manipulation de projets de développement logiciel, dont certaines ont été adaptées au langage. La figure 4 fournit une vue de l'éditeur et de ces fonctionnalités.

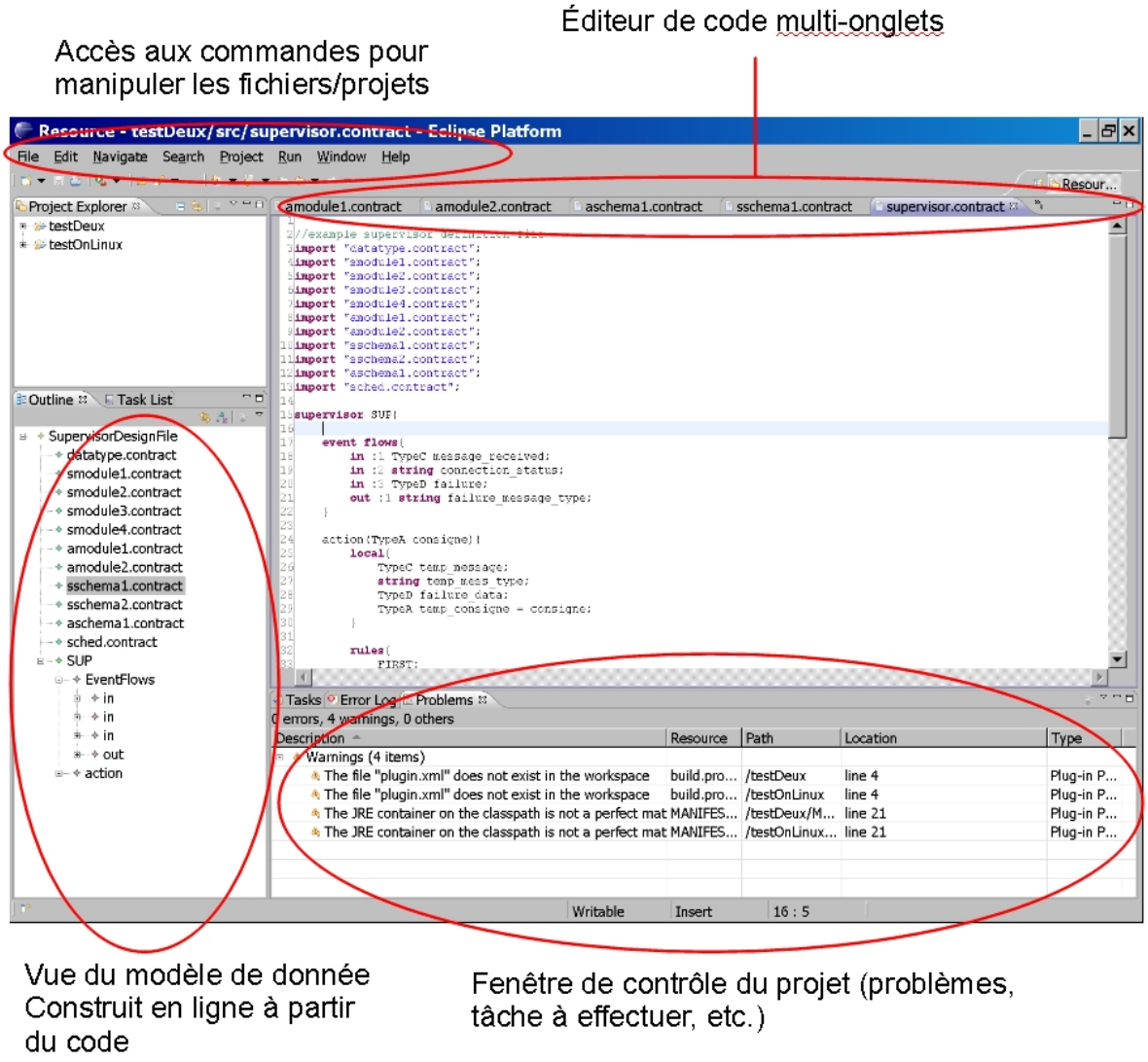

Figure 5: Fonctionnalités de l'éditeur de code

L'éditeur fournit tout d'abord toutes les fonctionnalités de manipulation (via la barre des menus ou des raccourcis clavier) de fichier et de dossiers: annulation des dernières actions utilisateur, création / sauvegarde de fichiers et de projets, etc.. Il fournit un **explorateur de projets** qui permet de visualiser les ressources de chaque projet de manière claire et un éditeur de code multi-onglet qui permet d'accéder et de naviguer facilement dans les contenus des fichiers de code source. Enfin comme tous les éditeurs ECLIPSE il fournit des vues spécialisées pour : contrôler différents aspects du projet (erreur internes, problèmes rencontrés - notamment les erreurs syntaxique et sémantiques dans le code, tâches à effectuer, etc.) et visualiser le modèle de données construit en ligne à partir du code .(fenêtre *Outline*). L'éditeur possède donc toutes les fonctionnalités de base qu'on peut attendre d'un éditeur de code.

5) Génération et personnalisation du code temps-réel

Une fois que l'utilisateur a défini l'architecture de contrôle de son application robotique, il peut générer le code source C correspondant. Pour cela l'éditeur propose une action particulière qui peut être accédée en fait : clic droit sur le projet à générer, puis en sélectionnant l'action « Generate All » (cf. figure 6). Cette action va générer l'intégralité des fichiers source C, des fichiers de compilation, auquels l'utilisateur pourra apporter une personnalisation.

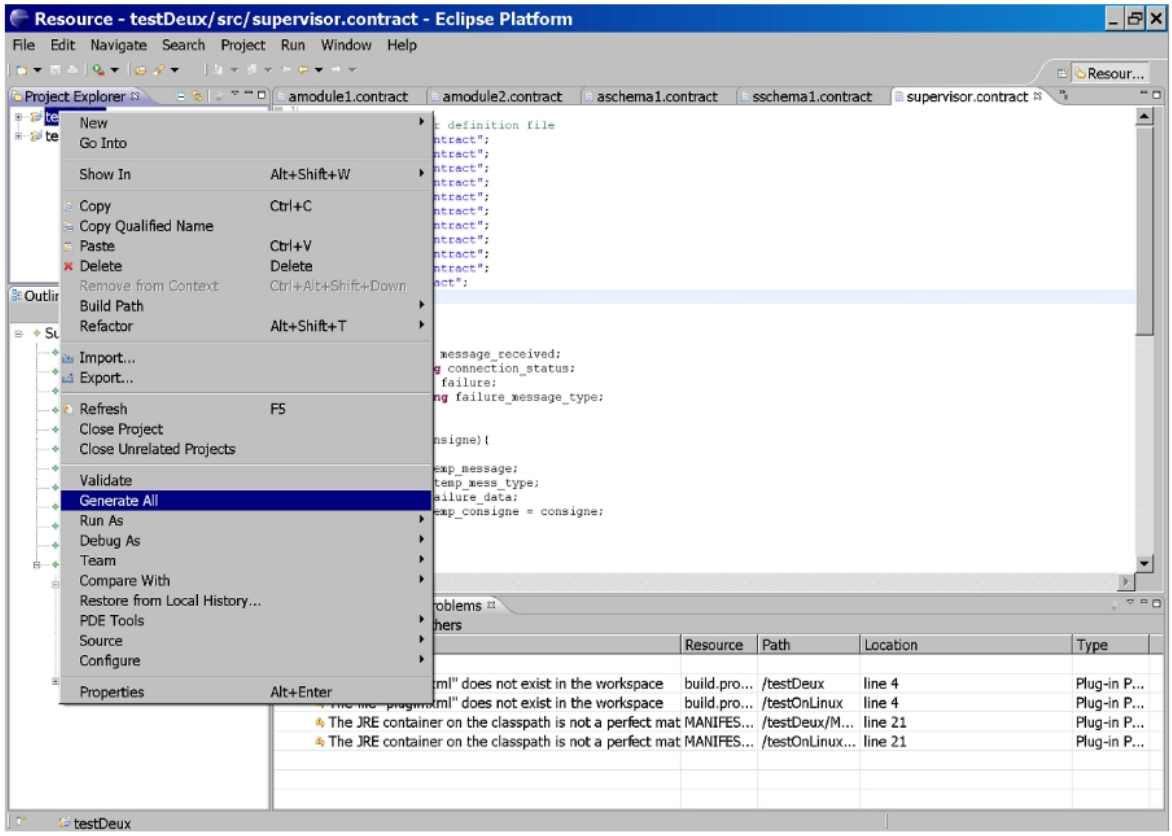

Figure 6: Génération du code source

La génération de code va créer une arborescence dans le dossier src-gen du projet concerné. Cette arborescence suit la structure suivante (cf. fig. 7):

- le sous-dossier `generated/architecture/` contient le fichier de compilation (`Makefile`) global de l'architecture qui permet de compiler et nettoyer les sources de l'architecture dans sa globalité, ainsi que d'exécuter et de terminer les exécutables des modules de l'architecture.

- le sous-dossier `generated/architecture/common/` contient l'ensemble des fichiers relatifs aux types de données partagés par les modules (qui suivent la convention de nommage `Shared<nom du type de donnée>.h`) et aux fonctions utilisateurs (qui suivent la convention de nommage `SharedFunction<nom de la fonction>.h`).

- le sous-dossier `generated/architecture/modules/ORD_mod/` contient le code source complet du module ordonnanceur, c'est-à-dire le code du module (fichier `module.c`) contenant notamment l'algorithme générique d'ordonnancement, la définition des types de données et constantes utilisées par le code générique (fichier `schedtypes.h`), un fichier de définition de fonction auxiliaires utilisées par les modules (fichier `tools.h`), le fichier définissant les types de données échangés entre le module ordonnanceur et les modules superviseurs (fichier `schedmessages.h`), le code de configuration de l'ordonnanceur spécifique à l'application robotique qui est généré à partir de la description de l'architecture (fichier `vocabulary.c`) et le fichier de compilation (`Makefile`) permettant de compiler / nettoyer le code source et d'exécuter ou terminer l'exécutable correspondant au module ordonnanceur.

Figure 7: Arborescence générée

- les sous-dossiers suivant la convention de nommage `generated/architecture/modules/<nom module>_mod/` qui contiennent chacun la définition d'un module de la couche exécutive. Chaque dossier contient le fichier source du module concerné (fichier `module.c`) ainsi que le fichier de compilation (`Makefile`) permettant de compiler / nettoyer le code source et d'exécuter ou terminer l'exécutable correspondant au module.

- les sous-dossiers suivant la convention de nommage `generated/architecture/modules/<nom module superviseur>_mod/` qui contiennent chacun la définition d'un module superviseur. Chaque dossier contient le fichier source du module superviseur concerné (fichier `module.c`) qui contient notamment les algorithmes génériques du mécanisme de supervision, le fichier de compilation (`Makefile`) permettant de compiler / nettoyer le code source et d'exécuter ou terminer l'exécutable correspondant au module superviseur, le fichier générique de définition des types, variables et constantes utilisées en interne dans le module (fichier `suptypes.h`) et le fichier de configuration du superviseur qui est généré à partir de la description du superviseur (fichier `vocabulary.c`).

- le sous-dossier `generated/architecture/tools/` qui contient l'ensemble des commandes utilisateur générée les modules de l'architecture. Chaque commande correspond à un fichier source C (qui suit la convention de nommage `control<nom module>.c`) définissant un exécutable et à un fichier de compilation dédié (qui suit la convention de nommage `makecontrol<nom module>`), soit deux fichiers par module. Un fichier de compilation global (fichier `Makefile`) permet de gérer la compilation globale des commandes. Ces commandes utilisateur permettent de contrôler l'exécution des modules dans une optique de test ou pour démarrer l'architecture (utilisation de la commande de contrôle du superviseur de plus haut niveau hiérarchique).

Pour l'utilisation de l'environnement ContrACT, il n'est pas nécessaire de comprendre l'intégralité du code généré. Il faut par contre comprendre quelles sont les possibilités de personnalisation du code source généré et quelles contraintes s'appliquent à cette personnalisation. Nous détaillons dans les sections suivantes les possibilités de personnalisation. Notons qu'aucune personnalisation de code ne peut être réalisée dans le code source des modules superviseurs et du module ordonnanceur, puisque le code de ces derniers est **généré intégralement** par l'outil à partir de la description de l'architecture.

5.1) Personnalisation des types de données

Chaque type de donné défini via le langage Cactal est traduit en un fichier d'entête C. L'exemple 9 montre le code généré pour le type de donnée `TypeA`. La génération de code a créé un structure de donnée C équivalente au type de donnée. Notons que les **`float`** en Cactal correspondent à des **`double`** en C, que les **`int`** en Cactal correspondent à des **`long`** en C et que

les **`string`** en Cactal correspondent à des tableau de 256 caractère (**`char`** `[256]`).

La génération de code a également généré un ensemble de fonctions nécessaires à la gestion de ces types de données au sein des superviseurs:

- Fonctions pour sérialiser (`serialize_TypeA`) et dé-sérialiser (`deserialize_TypeA`) la structure de donnée.
- Fonctions de test correspondant à chacun des opérateurs de base (`==`, `!=`, `>`, `<`, `>=`, `<=`) utilisables dans les superviseurs pour tester les variables.
- Fonction d'affectation (`set_TypeA`) correspondant à l'opérateur d'affectation (=) utilisé dans les superviseurs.
- Fonction d'impression (`print_TypeA`) de la variable du type considéré sur la sortie standard.
- Fonction d'extraction (`get_TypeA`) de la valeur d'une variable depuis une chaine de caractère, utilisé pour le passage d'argument depuis les commandes utilisateur.

```
#ifndef SHARED_TYPEA_H
#define SHARED_TYPEA_H
#include "tools.h"

//from here you can make new inclusions        and definitions
/*PROTECTED REGION ID(protect_additionnal_import_TypeA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/


typedef struct TypeA{
          double i;
          double j;
          double k;
} TypeA;


unsigned int serialize_TypeA(TypeA * arg, unsigned char * buffer);
unsigned int deserialize_TypeA(TypeA * arg, unsigned char* buffer);
int diff_TypeA(TypeA * arg1, TypeA * arg2);
int eq_TypeA(TypeA * arg1, TypeA * arg2);
int inf_TypeA(TypeA * arg1, TypeA * arg2);
int sup_TypeA(TypeA * arg1, TypeA * arg2);
int infeq_TypeA(TypeA * arg1, TypeA * arg2);
int supeq_TypeA(TypeA * arg1, TypeA * arg2);
void set_TypeA(TypeA * arg1, TypeA * arg2);
int get_TypeA(TypeA * arg, char* description);
void print_TypeA(TypeA * arg);

//from here you can declare new functions
/*PROTECTED REGION ID(protect_additionnal_decl_TypeA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/
...
```

Exemple 9 : extrait du code du fichier SharedTypeA.h généré à partir du type TypeA

La première possibilité laissé à l'utilisateur est celle de déclarer de nouvelles fonctions associées à ce type de donnée. Pour cela, l'utilisateur doit placer le code de déclaration de nouvelles fonctions entre les balises `/*PROTECTED REGION ID(protect_additionnal_decl_TypeA) ENABLED START*/` et `/*PROTECTED REGION END*/`. Ces balises servent à définir une région

protégée pour la génération : le générateur n'écrasera pas ce code contenu entre les balises lors des générations de code suivantes, à moins que le paramètre `ENABLED` n'ait été retiré de la première balise. Notons que, bien sur, l'utilisateur ne doit pas toucher aux balises elles mêmes hormis cette manipulation. Ainsi l'utilisateur peut modifier la définition des types de données sans avoir à réécrire complètement l'intégralité du code spécifique qu'il a voulu rajouter.

La déclaration de nouvelles fonctions, peut amener la nécessité d'utiliser des bibliothèques non incluse par défaut, par exemple des bibliothèques définissant des types de données spécifiques qui seront utilisé en paramètres. Pour cela le code généré propose une région protégée en début de fichier délimité par les balises `/*PROTECTED REGION ID(protect_additionnal_import_TypeA) ENABLED START*/` et `/*PROTECTED REGION END*/`.dans laquelle il pourra faire les inclusions nécessaires (**`#include`**), d'éventuelles définitions (**`#define`**) ou déclaration de variables globales ou de nouvelles structures de données.

A la suite de l'exemple 9 le fichier contient l'implémentation de toutes les fonctions par défaut (cf. section 8.2 pour l'exemple complet). Parmi elles, **seules les fonctions de test sont personnalisable**. L'exemple 10 décrit la fin du code généré pour le type de donnée `TypeA`.

...

```
int infeq_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_infeq_TypeA) ENABLED START*/
        if (inf_TypeA(arg1,arg2) == 1 || eq_TypeA(arg1,arg2) == 1)
                return 1;
        return 0;
        /*PROTECTED REGION END*/
}

int supeq_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_supeq_TypeA) ENABLED START*/
        if (sup_TypeA(arg1,arg2) == 1 || eq_TypeA(arg1,arg2) == 1)
                return 1;
        return 0;
        /*PROTECTED REGION END*/
}

//from here you can define new functions for your newly defined type

/*PROTECTED REGION ID(protect_additionnal_impl_TypeA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/

#endif
```

Exemple 10: extrait du code du fichier SharedTypeA.h (fin du fichier)

L'ensemble des fonctions de test par défaut d'un type de donnée (e.g. `diff_TypeA`, `eq_TypeA`, `inf_TypeA`, `sup_TypeA`, `infeq_TypeA`, `supeq_TypeA`, pour le type `TypeA`) reçoivent une implémentation par défaut, générée automatiquement, de manière à être fonctionnelle si l'utilisateur ne fournit pas d'implémentation propre. Néanmoins dans de nombreux cas cette implémentation par défaut a peu de sens (hormis peut-être pour celles correspondant aux opérateurs `==` et `!=`), c'est pourquoi l'utilisateur a la possibilité de personnaliser le code de ces fonctions.

Cette personnalisation se fait simplement en modifiant le code de ces fonctions contenues

entre les balises `/*PROTECTED REGION ID(<nom fonction>) ENABLED START*/` et `/*PROTECTED REGION END*/` définissant la région protégée propre à la fonction concernée. De cette manière, lors d'une nouvelle génération de code, l'implémentation redéfinie par l'utilisateur ne sera pas modifiée.

La dernière région protégée du fichier de définition de type permet d'implémenter toutes les nouvelles fonctions, propres au type, déclarée précédemment par l'utilisateur. L'implémentation se fait comme pour n'importe quelle fonction C.

Notons que nous avons choisi d'utiliser cette structuration des régions protégées :

- Afin que l'utilisateur puisse implémenter les fonctions qu'il désire utiliser au sein des modules (de la couche exécutive) qui exploitent ces types de données.
- Afin de limiter par ailleurs les erreurs de l'utilisateur en clarifiant les endroits où il peut effectuer des modifications sans compromettre le fonctionnement des modules.
- Afin de s'assurer que chaque génération de code produira le code conforme à la description architecturale sans pour autant « écraser » le code utilisateur.

5.2) Personnalisation des fonctions

Chaque fonction (de test ou de calcul) définie via le langage Cactal est traduite en un fichier d'entête C. L'exemple 11 montre le code généré pour la fonction `testit`. La personnalisation suit exactement la même technique que pour les types de données, à savoir l'implémentation du code utilisateur à l'intérieur de régions protégées.

```
#ifndef SHARED_FUNCTION_TESTIT_H
#define SHARED_FUNCTION_TESTIT_H
#include "SharedTypeA.h"

//from here you can make inclusions   and definitions
/*PROTECTED REGION ID(protect_additionnal_import_function_testit) ENABLED
START*/
//TODO
/*PROTECTED REGION END*/

int testit(char * arg1,TypeA * arg2);

int testit(char * arg1,TypeA * arg2){
        /*PROTECTED REGION ID(protect_function_impl_testit) ENABLED START*/
        //TODO
        return 0;//return 0 when result is false otherwise must return 1
        /*PROTECTED REGION END*/
}

#endif
```

Exemple 11: code du fichier SharedFunctiontestit.h

Chaque fichier de code source d'une fonction utilisateur contient une région protégée en début de fichier dans laquelle l'utilisateur peut faire différentes opération d'inclusion, de définition, de déclaration de variable ou de types structurés, exactement de la même manière et pour les mêmes raisons que pour les types de données.

La différence majeure avec les types de données, est que **la personnalisation est obligatoire** pour les fonctions, car il n'y a aucune implémentation par défaut générée par l'outil.

5.3) Personnalisation des modules de la couche exécutive

Chaque module de la couche exécutive défini via le langage Cactal est traduit en un fichier source C (fichier `module.c`).qui contient :

- le **code d'interface** du module, c'est à dire en premier lieu l'implémentation de ses différents ports. L'exemple 12 présente de code d'interface généré à partir du module `MAA` dont la description en Cactal a été fournit par l'exemple 3.
- Le **squelette de comportement** du module qui correspond à sa boucle d'attente passive de réaction aux requêtes (cf. ***[Passama 2010]***). Le squelette de comportement du module MAA est présenté de façon complète dans la section 8.2.
- Le **code personnalisable**, qui correspond à des fonctions spécifiques qui sont appelées dans le squelette de comportement et qui implémentent le comportement métier du module. La personnalisation des modules se fait de la même manière que pour les fonctions et types de données, à savoir en exploitant des régions protégées dans le code qui ont été générées par l'outil au moment de la génération de code.

```
#include <libmodule/libmodule.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "tools.h"
#include "SharedTypeC.h"
#include "SharedTypeE.h"
#include "SharedTypeA.h"


//from here you can make new imports or definitions
/*PROTECTED REGION ID(protect_additionnal_import_MAA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/

MODULE_DESCRIPTION(/*PROTECTED REGION ID(protect_description_MAA) ENABLED
START*/"test asynchronous module A for Cactal generation testing"/*PROTECTED
REGION END*/);
MODULE_AUTHOR(/*PROTECTED REGION ID(protect_author_MAA) ENABLED START*/"Robin
Passama"/*PROTECTED REGION END*/);

//Exchanges Management Variables
char controlevent[256] = "\0";
TypeC message_type_notification = { "\0", 0, { 0.0, 0.0, 0.0 } };
TypeE received_message = { { "\0", 0, { 0.0, 0.0, 0.0 } }, { "\0", 0 } };
char connection_status[256] = "\0";
TypeA param = {22.12, 6.98, 897.34};
```

```
// module name
char MODULE_NAME[] = "MAA";
//module priority
int MODULE_PRIORITY = 37;
// input data flows : example = { &Variable, PortReception, Taille, Periodicity
(important if module name is set), "Name of emitter module"(can be unused by
setting value to "---"), PortModuleEmetteur} ou { &Variable, PortReception,
Taille, Periodicite, "---", 0}
ModuleUse IUSE[] = { IUSE_TERM };

// output data flows : example = { &Variable, PortEmision, Size}
ModuleProduce IPRODUCE[] = { IPRODUCE_TERM };

//input event flows : example = { &Variable, PortReception, size, "module name"
(can be unused by setting value to "---"), PortEmision, oneshot}
ModuleReact IREACT [] = { { &controlevent, 0, sizeof(char)*256, "---", 0, 0},
IREACT_TERM };

//output event flows : example = { &variable, PortEmision, size }
ModuleDetect IDETECT [] = { { &message_type_notification, 0, sizeof(TypeC)},
{ &received_message, 1, sizeof(TypeE)}, { &connection_status, 2,
sizeof(char)*256}, IDETECT_TERM };

// size of parameters that can be received in the request mailbox:
size_t PARAM_SIZE_IN[] = { sizeof(TypeA), PARAM_SIZE_TERM };

// size of parameters or events that can be sent in request mailboxes of other
modules (user has to add sizes of sent param requests, if any):

size_t PARAM_SIZE_OUT[] = {/*PROTECTED REGION ID(protect_param_out_MAA) ENABLED
START*/ /*PROTECTED REGION END*/PARAM_SIZE_TERM };
```

Exemple 12 : code d'interface du module MAA

Le **code d'interface** du module utilise la librairie système que nous avons implémenté pour définir l'ensemble des ports et plus généralement de toutes les informations et structures de données nécessaires à la communication inter-modules. Dans l'exemple 12, le code d'interface généré pour le module `MAA` est présenté. Dans ce code on retrouve:

- Les **informations non fonctionnelles** sur le module, comme le ou les auteurs de la personnalisation du code du module (macro `MODULE_DESCRIPTION`), et la description synthétique du module (macro `MODULE_DESCRIPTION`). Notons que le contenu de ces deux macros est personnalisable (présence des balises de la région protégée de chaque macro).
- Les **informations liées à la tâche temps-réel** correspondant au module, c'est-à-dire le nom du module (variable globale `MODULE_NAME`) à partir duquel sera généré l'identifiant de la tâche et la priorité du module (variable globale `MODULE_PRIORITY`). Ces informations ne sont **pas personnalisables**.
- Les **informations liées à la définition et aux paramétrage des ports**. Le port de requête du module est créé par défaut (sans besoin de déclaration spécifique) et il est configuré automatiquement en fonction des tailles des données associées à certaines requêtes qu'il est susceptible de recevoir (variable `PARAM_SIZE_IN`, et tailles des variables utilisées dans la variable `IREACT`) qui sont déduite de l'interface du module décrite dans l'architecture. Le port de requête n'est donc **pas personnalisable**. Les autres ports sont définis à partir de

variables spécifiques (`IUSE` pour les port d'entrée de données, `IPRODUCE` pour les ports de sortie de donnée, `IREACT` pour les port d'entrée d'événements, `IDETECT` pour les ports de sortie d'événements) dont le contenu est automatiquement généré à partir de la description de l'interface du module : ces ports ne sont donc **pas personnalisables**. Les seules informations qui **doivent être personnalisées** sont les tailles des paramètres émis par le module : la variable `PARAM_SIZE_OUT` doit contenir la taille de chaque type de donnée (e.g. `sizeof(double)`) potentiellement émis par un module via l'utilisation d'une requête de configuration. Dans la plupart des cas néanmoins l'utilisateur n'a rien à faire puisque les modules synchrones ne peuvent émettre directement des paramètres et que cette possibilité est laissée aux modules asynchrones de manière marginale (i.e. devrait être évité par l'utilisateur autant que possible).

Comme le montre l'exemple 12, des variables ont été automatiquement générées et sont utilisées dans la définition des ports. Ces variables possèdent exactement le même nom que celui donné par l'utilisateur dans la description de l'interface du module afin que l'utilisateur puisse facilement identifier les variables associées à chaque port. Par exemple, la variable `received_message` a été générée automatiquement et a été associée au port de sortie d'événement d'index 1 (cf. deuxième élément du tableau `IDETECT`), comme spécifié dans l'interface du module `MAA` (cf. exemple 3). L'utilisateur pourra utiliser ces variables dans des appels à la librairie développée pour implémenter les modules, afin de coder le comportement du module.

Par ailleurs, il existe une région protégée en début de fichier (délimitée par les balises `/*PROTECTED REGION ID(protect_additionnal_import_<nom module>) ENABLED START*/`et `/*PROTECTED REGION END*/`) dans laquelle l'utilisateur peut ajouter des inclusions de bibliothèques, des définitions de macro ou de constantes et des déclarations de types de données, comme pour les autres fichiers personnalisables. Comme le montre l'exemple complet du module `MAA` dans la section 8.2, il existe également des régions protégées pour définir de nouvelles variables globales au module (délimitée par les balises `/*PROTECTED REGION ID(protect_internal_vars_<nom module>) ENABLED START*/` et `/*PROTECTED REGION END*/`) et de nouveaux prototypes de fonctions (délimitée par les balises `/*PROTECTED REGION ID(protect_internal_functions_decl_<nom module>) ENABLED START*/` et `/*PROTECTED REGION END*/`). Ces données et fonctions auxiliaires supplémentaires sont utilisées pour implémenter l'ensemble des fonctions qui doivent ou peuvent être personnalisées par l'utilisateur.

```
int _init_module(void){//==0 false, != 0 true
        /*PROTECTED REGION ID(protect_init_MAA) ENABLED START*/
        printf("initialization of MAA\n");
        _connected = 0;
        return 1;
        /*PROTECTED REGION END*/
}

void _exit_module(void){
        /*PROTECTED REGION ID(protect_exit_MAA) ENABLED START*/
        printf("termination of MAA\n");
        /*PROTECTED REGION END*/
}
```

Exemple 13: code personnalisable d'initialisation et de terminaison du module

Les fonctions qui peuvent être personnalisées par l'utilisateur sont en premier lieu les fonction d'initialisation (`_init_module`) et de terminaison (`_exit_module`) du module. Le code de l'exemple 13 définit l'implémentation simple de ces fonctions pour le module `MAA`. C'est dans ces fonction que le module gèrera par exemple un lien de communication avec des capteurs ou actionneurs embarqué (ouverture et fermeture du lien de communication). La fonction `_init_module` est appelée au moment de l'initialisation du module, c'est-à-dire juste après que la tâche temps réel exécutant le module ait été créée et ses structures internes (i.e. gestion des ports) initialisée, et la fonction `_exit_module` est appelée au moment de sa terminaison, c'est-à-dire juste avant que les structure internes soient dértuites et avant que la tâche temps réel exécutant le module cesse d'exister (cf. Squelette de comportement du module `MAA`, section 8.2).

Pour les modules synchrones la seule et unique autre fonction **qui doit être personnalisée** est celle qui implémente le comportement nominal du module, nommée `_exec_module_state`. Pour les modules asyncrhones il s'agit de la seule fonction qui **doit obligatoirement être personnalisée**, mais d'autres fonctions (que nous verrons par la suite) peuvent l'être. L'exemple 14 présente la personnalisation apportée au module `MAA`.: le module est chargé de simuler la réception de message (qui viendraient dans un cas réel du réseau) à partir d'entrées utilisateur et de générer en conséquence des événements.

La fonction `_exec_module_state` s'implémente de la manière suivante:

- l'utilisateur définit un ou plusieurs états du comportement, chaque état correspondant à une étape de l'aglorithme global contenu dans la fonction. L'état courant est représenté par le paramètre `stateindex`, dont la valeur pointée est `0` (i.e. état initial) lors du le premier appel à la fonction. La décomposition de l'algorithme en plusieurs états n'est pas obligatoire mais l'utilisateur doit faire attention au fait que le module n'est plus réactif à des requêtes lorsqu'il exécute la fonction. Le fait de décomposer les calculs en différentes étapes va entrainer l'arrêt momentanné des calculs (i.e. de la fonction) afin que le module surveille l'arrivée de requêtes (notamment des requêtes d'arrêt d'exécution du comportement nominal) dans le but de rester réactif. Une fois cette surveillance et d'éventuelles réactions effectuées, la fonction est réappellée avec le **nouvel état courant comme paramètre**.
- En fonction de l'état courant passé en paramètre, le code du module peut sélectionner l'étape à exécuter, par exemple via l'utilisation d'un bloc **`switch`** testant l'état courant. L'exemple 14 définit deux états pour le module `MAA`, l'état `0` dans lequel il réalise son comportement et l'état `1` dans lequel il ne fait rien (état terminal).
- Il faut que l'utilisateur puisse **signaler** au squelette de comportement qui exécute la fonction (cf. exemple complet du code généré pour `MAA`, section 8.2) **que l'algorithme global a terminé son exécution** (i.e. toutes les étapes de l'algorithme ont été réalisées). Pour cela il lui suffit de fixer la valeur pointée de la variable `stateindex` à `-1`.(cf. `*stateindex = -1` dans l'état `1`).

```
void _exec_module_state(int * stateindex){//state from 0 to x with x > 0, if
stateindex is -1 at end of execution then run is finished
/*PROTECTED REGION ID(protect_exec_MAA) ENABLED START*/
char input[256] = "";
switch(*stateindex){
case 0:printf("MAA : cycle on state 0\n");
printf("USER : enter a command : %s\n", (_connected==1?"1)disconnection 2)
receiving message 3)exitting (and disconnecting) 4)Loop":"1)connection 2)
exitting 3)Loop"));scanf("%s", &input[0]);
if (_connected==1){
          if (strcmp(input,"1")==0){
                    strcpy(connection_status,"disconnected");
                    PostEvent(&connection_status[0]);
                    _connected = 0;printf("disconnection => posting event\n");
          }
          else if(strcmp(input,"2")==0){
                    strcpy(received_message.message.message , "FUFU");
                    received_message.message.messagetype = 1;
                    received_message.message.value.i =
received_message.message.value.i + _increment;
                    received_message.message.value.j =
received_message.message.value.j + _increment;
                    received_message.message.value.k =
received_message.message.value.k + _increment;
                    strcpy(&received_message.data_.dataname[0], "0");
                    received_message.data_.datavalue = 678;
                    PostEvent(&received_message);
                    printf("posting the whole message is done !\n");
                    set_TypeC(&message_type_notification,
&received_message.message);
                    PostEvent(&message_type_notification);
                    printf("posting the message type is done !\n");
          }
          else if(strcmp(input,"3")==0){printf("exitting\n");
                    strcpy(connection_status,"disconnected");
                    PostEvent(&connection_status[0]);
                    *stateindex = 1;
                    _connected = 0;printf("disconnection => posting event\n");
          }
          else if(strcmp(input,"4")==0){printf("loop\n");}
          else{printf("BAD INPUT : retry\n");}
}
else{     if (strcmp(input,"1")==0){printf("connection => posting event\n");
                    strcpy(connection_status,"connected");
                    PostEvent(&connection_status[0]);
                    _connected = 1;printf("connection => event posted\n");
          }
          else if(strcmp(input,"2")==0){printf("exitting\n");*stateindex = 1;}
          else if(strcmp(input,"3")==0){printf("loop\n");}
          else{printf("BAD INPUT : retry\n");}
}break;
case 1:   printf("MAA : state 1 => terminal state (do nothing)\n");
          *stateindex = -1;
          break;
default:  break;
}
/*PROTECTED REGION END*/
}
```

Exemple 14 : code personnalisable de l'activité nominale du module

Les modules asynchrones peuvent également personnaliser d'autres fonctions qui sont relatives à leur réaction immédiates à des requêtes de configuration et à des événements reçus. Ces fonctions ne sont générées que si le module asynchrone est suceptible de recevoir respectivement au moins un paramètre et au moins un événement. L'exemples 15 fournit le code de la fonction à personnaliser pour réagir à des événements reçus (fonction `_react_to_event`) et l'exemple 16 ournit le code de la fonction à personnaliser pour réagir à des requêtes de configuration (fonction `_react_to_parameterizing`).

```
void _react_to_event(int receiverport , int senderport, char * sendermodule,
RTIME * eventProductionTime, RTIME * eventReceptionTime){
          /*PROTECTED REGION ID(protect_event_MAA) ENABLED START*/
          //TODO : implement the code for reaction to event notifications
          /* example pattern for implementing this function
          if (strlen(sendermodule) != 0){//event incoming in a port of the
module
                    switch(receiverport){
                    case 0:
                              ...
                              break;
                    case 1:
                              ...
                              break;
                    }
          }
          else{//event coming from the specified port of the sender module
                    ...
          }

          */
          /*PROTECTED REGION END*/
}
```

Exemple 15 : code personnalisable de la réaction aux événements

```
void _react_to_parameterizing(int parameterport, RTIME * productionTime, RTIME *
receptionTime){
          /*PROTECTED REGION ID(protect_param_MAA) ENABLED START*/
          //TODO : implement the code for reaction to parameterizing requests
          /* example pattern for implementing this function
          switch(parameterport){
          case 0:   ...
                    break;
          case 1:   ...
                    break;
          case 2:   ...
                    break;
          }
          */
          /*PROTECTED REGION END*/
}
//Specific functions (from here you have to implement new internal functions
declared before main)
/*PROTECTED REGION ID(protect_internal_functions_impl_MAA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/
```

Exemple 16 : code personnalisable de la réaction au paramétrage du module

Ces deux fonctions peuvent être personnalisées suivant la même logique: en fonction du port du module ayant reçu l'événement (`receiverport`) ou le paramètre(`parameterport`), l'utilisateur peut déclencher des réactions appropriées. C'est pourquoi l'utilisation de bloc **`switch`** testant le port concerné est préconisée: chaque **`case`** représente alors un des ports d'entrée d'événements (cf. variable `IREACT`) ou un des ports de paramètres défini pour le module. Notons que les variables associées aux port d'entrée d'événements (cf. variables dans le tableau `IREACT`) ou aux ports de paramètres (variables globales du module non référencées par ailleurs dans une structure de donnée particulière comme pour les ports d'entrée/sortie) **ont été mises à jour avant l'appel à ces fonctions**, leur valeur est donc exploitable, car cohérente relativement à la gestion de la requête.

L'appel à ces deux fonctions se fait automatiquement dans le squellete de comportement du module. L'appel fixe la valeur des paramètres `productionTime` et `receptionTime`, qui peuvent être utilisés au sein des fonctions pour connaitre la date ou l'événement ou le paramètre a respectivement été produit par le module émetteur et il a été reçu par le module receveur (i.e. Le module implémentant ces fonctions).

L'implémentation des fonctions personnalisables peut être réalisée en utilisant des fonctions auxiliaires qui sont elles mêmes définies dans la région protégée contenue dans les balises `/*PROTECTED REGION ID(protect_internal_functions_impl_<nom module>) ENABLED START*/` et `/*PROTECTED REGION END*/`. Pour implémenter ces fonctions, l'utilisateur peut utiliser différentes primitives fournies par la librairie temps-réel, et qui servent notamment à exploiter les ports des modules :

- `GetData(void*,RTIME*productiondate,RTIME*receptiondate)` est utilisée pour **consomer la première donnée** disponible dans un port d'entrée de données. Le paramètre d'appel est l'adresse de la variable associée au port d'entrée de données concerné (cf. variable `IUSE`). Une fois l'appel effectué, la variable passé en paramètre contiendra la valeur de cette donnée. Les deux paramètre temporel permettent de récupérer la date de production et la date de réception de la variable. Pour cela l'utilisateur doit utiliser par défaut les deux variables globale a disposition dans le module (productionTime et receptionTime) ou des variables qu'il aura lui-même défini.
- `GetLatestData(void*,RTIME*productiondate,RTIME*receptiondate)` est utilisée pour **consomer toutes les données** disponibles dans un port d'entrée de données. Le paramètre d'appel est l'adresse de la variable associée au port d'entrée de données concerné (cf. variable `IUSE`) concerné. Une fois l'appel effectué, la variable passé en paramètre contiendra la valeur de la dernière donnée disponible (i.e. la donnée la plus à jour).
- `PostData(void*)` est utilisée pour **produire** une **donnée** via un port de sortie de données. Le paramètre d'appel est l'adresse de la variable associée au port de sortie de données concerné (cf. variable `IPRODUCE`).
- `PostEvent(void*)` est utilisée pour **produire** un **événement** via un port de sortie d'événement. Le paramètre d'appel est l'adresse de la variable associée au port de sortie d'événement concerné (cf. variable `IDETECT`).
- `SetModuleParam(char* ModuleName, unsigned char port, void* Data, unsigned long size)` est la fonction qui peut être utilisée pour envoyer une requête de configuration à un module. Comme cela a été expliqué précédemment l'utilisation de cette fonction devrait se faire avec parcimonie et **uniquement au sein des modules**

asynchrones. Par ailleurs, dès que cette fonction est utilisé il faut ajouter la taille de la variable émise au tableau `PARAM_SIZE_OUT`. Les arguments de la fonction sont dans l'ordre : le nom du module cible, l'identifiant du port de paramètre considéré pour ce module, l'adresse de la variable qui contient la nouvelle valeur du paramètre et la taille en mémoire de cette variable. L'utilisateur sdoit donc prendre garde à ce que le port de paramètre ciblé existe bien dans le module receveur et que le type de la variable émise soit bien le même que le type du paramètre concerné.

6) Conclusion

Le présent tutoriel a expliqué en détail les spécificité de l'environnement ContrACT, depuis la création de projet et l'édition de la description d'architectures de contrôle en utilisant le langage Cactal, jusqu'à la génération et la personnalisation du code temps-réel C.

La section 8 fournit une description plus détaillée de l'exemple développé tout au long de ce document, soit l'intégralité du code de description de l'architecture de contrôle et des extrait du code de modules synchrone (`MSC`) et asynchrones (`MAA`) personnalisés.

7) Références

[Passama 2010] Passama Robin. *ContrACT : Une méthodologie de conception et de développement d'architectures de contrôle de robots*. Rapport de recherche LIRMM RR10025, Université Montpellier II, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), 2010.

8) Exemple

8.1) code complet de l'exemple en langage Cactal

Fichier *datatype.contract*

```
//synchronous
datatype TypeA {float i = 0.0, float  j =0.0, float  k=0.0};
datatype TypeB {TypeA coord = {1.0,1.0,1.0}, int  frfrrrf=1};

//asynchronous
datatype TypeC {string message, int  messagetype=0, TypeA value };
datatype TypeD {string dataname, int datavalue};
datatype TypeE {TypeC message, TypeD data_ };
datatype TypeF {int fufu, int juju};
```

```
function bool testit(string arg1, TypeA arg2);
function void doit(string argfirst, float argsecond);
```

Fichier *amodule1.contract*

```
import "datatype.contract";

asynchronous module MAA{

          parameters{
                    TypeA param = {22.12,6.98,897.34} :0;
          }

          event flows{
                    in :0 string controlevent;
                    out :0 TypeC message_type_notification;
                    out :1 TypeE received_message;
                    out :2 string connection_status;
          }
}
```

Fichier *amodule2.contract*

```
import "datatype.contract";

asynchronous module MAB{

          event flows{
                    in :0 TypeC message_type_notification;
          }
}
```

Fichier *aschema1.contract*

```
import "datatype.contract";
import "amodule1.contract";
import "amodule2.contract";

event schema schemaReceptionMessage(TypeA param = {27.12,0.0,0.04}){

          modules{
                    MAA (param :0);
                    MAB ();
          }

          communications{
                    MAA:0 =* MAB:0;//continuous event flow communication
          }
}
```

Fichier s*module1.contract*

```
import "datatype.contract";
```

```
synchronous module MSA{

          scheduling{
                    C=500us;
          }

          data flows{
                    out :0 TypeA coordinates;
          }
}
```

Fichier s*module2.contract*

```
import "datatype.contract";

synchronous module MSB{

          scheduling{
                    C=500us;
          }

          data flows{
                    out :0 TypeA coordinates;
          }
}
```

Fichier s*module3.contract*

```
import "datatype.contract";

synchronous module MSC{

          parameters{
                    TypeA param = {2.1,6.98,5.34} :0;
          }

          scheduling{
                    C=1ms;
          }

          data flows{
                    in :0 TypeA in_coordinates;
                    out :1 TypeB out_coordinates_commmand;
          }

          event flows{
                    out :1 TypeD failing_data;
          }
}
```

Fichier s*module4.contract*

```
import "datatype.contract";

synchronous module MSD{

          scheduling{
```

```
                C=1ms;
        }

        data flows{
                in :2 TypeB in_coordinates_command;
        }
}
```

Fichier *sschema1.contract*

```
import "datatype.contract";
import "smodule1.contract";
import "smodule3.contract";
import "smodule4.contract";

periodic schema schemaCommande1(TypeA consigne = {0.0, 0.0, 0.0}){

        realtime{
                PERIOD=1s;
                CRITICAL_DELAY=1s;
        }

        modules{
                MSA ();
                MSC (consigne:0);
                MSD ();
        }


        scheduling graph{
                MSA -> MSC;
                MSC -> MSD;
        }

        communications{
                MSA:0 -> MSC:0 * 1;//data flow communication
                MSC:1 -> MSD :2 * 2;//data flow communication
        }
}
```

Fichier *sschema2.contract*

```
import "datatype.contract";
import "smodule2.contract";
import "smodule3.contract";
import "smodule4.contract";

periodic schema schemaCommande2(TypeA consigne){

        realtime{
                PERIOD=1s;
                CRITICAL_DELAY=1s;
        }
```

```
        modules{
                MSB ();
                MSC (consigne:0);
                MSD ();
        }

        scheduling graph{
                MSB -> MSC;
                MSC -> MSD;
        }

        communications{
                MSB:0 -> MSC:0 * 1;//data flow communication
                MSC:1 -> MSD :2 * 1;//data flow communication
        }
}
```

Fichier *sched.contract*

```
import "datatype.contract";
import "smodule1.contract";
import "smodule2.contract";
import "smodule3.contract";
import "smodule4.contract";
import "sschema1.contract";
import "sschema2.contract";

scheduler configuration {

        modules{
                MSA;
                MSB;
                MSC;
                MSD;
        }

        schemas{
                schemaCommande1;
                schemaCommande2;
        }
}
```

Fichier *supervisor.contract*

```
import "datatype.contract";
import "smodule1.contract";
import "smodule2.contract";
import "smodule3.contract";
import "smodule4.contract";
import "amodule1.contract";
import "amodule2.contract";
import "sschema1.contract";
import "sschema2.contract";
import "aschema1.contract";
import "sched.contract";
```

```
supervisor SUP{

event flows{
          in :1 TypeC message_received;
          in :2 string connection_status;
          in :3 TypeD failure;
          out :1 string failure_message_type;
}

action(TypeA consigne){
          local{
          TypeC temp_message;
          string temp_mess_type;
          TypeD failure_data;
          TypeA temp_consigne = consigne;
          }

          rules{
          FIRST:[elapsed=1ms]
                    subscribe MAA:2 =* :2;subscribe MAA:0 =* :1;
                    activate schema schemaReceptionMessage()
                 [MAA:2<connection_status == "disconnected">]

          SECOND:[(MAA:2<(connection_status == "disconnected" & testit
("dedede", temp_consigne))> and started(FIRST){infinite})]
                    subscribe MSC:1=>:3;
                    activate schema schemaCommande1(consigne = temp_consigne)
                  [MSC:1(failure_data = failure, temp_mess_type = "commutation")]

          THIRD:[endof(SECOND)]
                    notify :1(failure_message_type=temp_mess_type)
                 [started(THIRD)]

          FOURTH:[endof(SECOND)]
                    compute doit("1.24",temp_consigne.k);
                    activate schema schemaCommande2(consigne = temp_consigne)
                  [endof(FIRST)]


          RECEPTION:[MAA:0(temp_message=message_received)]
                       parameterize MSC(temp_message.value :0);
                       activate resttime module MAA
                    [started(RECEPTION)]

          }
}
//other supervision functions
}
```

Fichier *supervisor2.contract*

```
import "datatype.contract";
import "supervisor.contract";

supervisor GLB{
```

```
event flows{
          in :0 string gdyegdyegdyge;
}


mission(TypeA consigne){
          local{
          TypeA consigne_courante = consigne;
          }
          rules{

          SINGLE:[elapsed=1ms]
                    subscribe SUP:1 => :0;
                    activate supervisor SUP.action(consigne=consigne_courante)
                  [SUP:1<gdyegdyegdyge=="commutation">]
          }
}
}
```

8.2) Extraits de code source C personnalisable généré par l'environnement ContrACT

Fichier `SharedTypeA.h` généré à partir de la définition du type `TypeA`

```
#ifndef SHARED_TYPEA_H
#define SHARED_TYPEA_H
#include "tools.h"

//from here you can make new inclusions        and definitions
/*PROTECTED REGION ID(protect_additionnal_import_TypeA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/


typedef struct TypeA{
          double i;
          double j;
          double k;
} TypeA;


unsigned int serialize_TypeA(TypeA * arg, unsigned char * buffer);
unsigned int deserialize_TypeA(TypeA * arg, unsigned char* buffer);
int diff_TypeA(TypeA * arg1, TypeA * arg2);
int eq_TypeA(TypeA * arg1, TypeA * arg2);
int inf_TypeA(TypeA * arg1, TypeA * arg2);
int sup_TypeA(TypeA * arg1, TypeA * arg2);
int infeq_TypeA(TypeA * arg1, TypeA * arg2);
int supeq_TypeA(TypeA * arg1, TypeA * arg2);
void set_TypeA(TypeA * arg1, TypeA * arg2);
int get_TypeA(TypeA * arg, char* description);
void print_TypeA(TypeA * arg);

//from here you can declare new functions
/*PROTECTED REGION ID(protect_additionnal_decl_TypeA) ENABLED START*/
//TODO
```

```
/*PROTECTED REGION END*/

unsigned int serialize_TypeA(TypeA * arg, unsigned char * buffer){
          unsigned int size = 0;
          size += extract_float(buffer, size, arg->i);
          size += extract_float(buffer, size, arg->j);
          size += extract_float(buffer, size, arg->k);
          return size;
}


unsigned int deserialize_TypeA(TypeA * arg, unsigned char* buffer){
          unsigned int size = 0;
          size += fill_float(&arg->i, size, buffer);
          size += fill_float(&arg->j, size, buffer);
          size += fill_float(&arg->k, size, buffer);
          return size;
}

void set_TypeA(TypeA * arg1, TypeA * arg2){
          arg1->i=arg2->i;
          arg1->j=arg2->j;
          arg1->k=arg2->k;
}

int get_TypeA(TypeA * arg, char* description){
          int sizeread=0, res=0, nbfields=3;
          char symbol = '{';
          if (arg == NULL) return -1;
          if ((res=increment_until_markerfound(&description[sizeread], &symbol))
== -1) return -1;
          sizeread+=res;
          if((res=get_float(&arg->i, &description[sizeread])) == -1) return -1;
          nbfields--;
          sizeread += res;
          if (nbfields != 0) symbol = ',';
          else symbol = '}';
          if ((res=increment_until_markerfound(&description[sizeread], &symbol))
== -1){
                    return -1;
          }
          sizeread+=res;
          if((res=get_float(&arg->j, &description[sizeread])) == -1) return -1;
          nbfields--;
          sizeread += res;
          if (nbfields != 0) symbol = ',';
          else symbol = '}';
          if ((res=increment_until_markerfound(&description[sizeread], &symbol))
== -1){
                    return -1;
          }
          sizeread+=res;
          if((res=get_float(&arg->k, &description[sizeread])) == -1) return -1;
          nbfields--;
          sizeread += res;
          if (nbfields != 0) symbol = ',';
          else symbol = '}';
          if ((res=increment_until_markerfound(&description[sizeread], &symbol))
== -1){
```

```
                return -1;
        }
        sizeread+=res;
        return sizeread;
}

void print_TypeA(TypeA * arg){
        printf("(object type %s : ","TypeA");
        printf("%s : %lf ; ","i", arg->i);
        printf("%s : %lf ; ","j", arg->j);
        printf("%s : %lf ; ","k", arg->k);
        printf(")");
}

//functions that can be redefined by the user (let the comment markers unchanged
by default, simply remove the ENABLED flag if you want to regenerate code)

int eq_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_eq_TypeA) ENABLED START*/
        if ((arg1->i==arg2->i?1:0) == 0) return 0;
        if ((arg1->j==arg2->j?1:0) == 0) return 0;
        if ((arg1->k==arg2->k?1:0) == 0) return 0;
        return 1;
        /*PROTECTED REGION END*/
}

int diff_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_diff_TypeA) ENABLED START*/
        if (eq_TypeA(arg1,arg2) == 0) return 1;
        return 0;
        /*PROTECTED REGION END*/
}

int inf_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_inf_TypeA) ENABLED START*/
        if ((arg1->i<arg2->i?1:0) == 0) return 0;
        if ((arg1->j<arg2->j?1:0) == 0) return 0;
        if ((arg1->k<arg2->k?1:0) == 0) return 0;
        return 1;
        /*PROTECTED REGION END*/
}

int sup_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_sup_TypeA) ENABLED START*/
        if (inf_TypeA(arg1,arg2) == 0 && eq_TypeA(arg1,arg2) == 0)
                return 1;
        return 0;
        /*PROTECTED REGION END*/
}

int infeq_TypeA(TypeA * arg1, TypeA * arg2){
        /*PROTECTED REGION ID(protect_infeq_TypeA) ENABLED START*/
        if (inf_TypeA(arg1,arg2) == 1 || eq_TypeA(arg1,arg2) == 1)
                return 1;
        return 0;
        /*PROTECTED REGION END*/
}

int supeq_TypeA(TypeA * arg1, TypeA * arg2){
```

```
          /*PROTECTED REGION ID(protect_supeq_TypeA) ENABLED START*/
          if (sup_TypeA(arg1,arg2) == 1 || eq_TypeA(arg1,arg2) == 1)
                    return 1;
          return 0;
          /*PROTECTED REGION END*/
}

//from here you can define new functions for your newly defined type

/*PROTECTED REGION ID(protect_additionnal_impl_TypeA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/

#endif
```

Fichier `SharedFunctiontestit.h` généré à partir de la définition du prototype `testit`

```
#ifndef SHARED_FUNCTION_TESTIT_H
#define SHARED_FUNCTION_TESTIT_H
#include "SharedTypeA.h"

//from here you can make inclusions   and definitions
/*PROTECTED REGION ID(protect_additionnal_import_function_testit) ENABLED
START*/
//TODO
/*PROTECTED REGION END*/

int testit(char * arg1,TypeA * arg2);

int testit(char * arg1,TypeA * arg2){
          /*PROTECTED REGION ID(protect_function_impl_testit) ENABLED START*/
          //TODO
          return 0;//return 0 when result is false otherwise must return 1
          /*PROTECTED REGION END*/
}

#endif
```

Fichier `module.c` du module MAA

```
//                module source file
//                MODULE: MAA
//                Libraries includes
#include <libmodule/libmodule.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "tools.h"
#include "SharedTypeC.h"
#include "SharedTypeE.h"
#include "SharedTypeA.h"


//from here you can make new imports or definitions
```

```
/*PROTECTED REGION ID(protect_additionnal_import_MAA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/

MODULE_DESCRIPTION(/*PROTECTED REGION ID(protect_description_MAA) ENABLED
START*/"test asynchronous module A for Cactal generation testing"/*PROTECTED
REGION END*/);
MODULE_AUTHOR(/*PROTECTED REGION ID(protect_author_MAA) ENABLED START*/"Robin
Passama"/*PROTECTED REGION END*/);

//Exchanges Management Variables
char controlevent[256] = "\0";
TypeC message_type_notification = { "\0", 0, { 0.0, 0.0, 0.0 } };
TypeE received_message = { { "\0", 0, { 0.0, 0.0, 0.0 } }, { "\0", 0 } };
char connection_status[256] = "\0";
TypeA param = {22.12, 6.98, 897.34};

// module name
char MODULE_NAME[] = "MAA";
//module priority
int MODULE_PRIORITY = 37;
// input data flows : example = { &Variable, PortReception, Taille, Periodicity
(important if module name is set), "Name of emitter module"(can be unused by
setting value to "---"), PortModuleEmetteur} ou { &Variable, PortReception,
Taille, Periodicite, "---", 0}
ModuleUse IUSE[] = { IUSE_TERM };

// output data flows : example = { &Variable, PortEmision, Size}
ModuleProduce IPRODUCE[] = { IPRODUCE_TERM };

//input event flows : example = { &Variable, PortReception, size, "module name"
(can be unused by setting value to "---"), PortEmision, oneshot}
ModuleReact IREACT [] = { { &controlevent, 0, sizeof(char)*256, "---", 0, 0},
IREACT_TERM };

//output event flows : example = { &variable, PortEmision, size }
ModuleDetect IDETECT [] = { { &message_type_notification, 0, sizeof(TypeC)},
{ &received_message, 1, sizeof(TypeE)}, { &connection_status, 2,
sizeof(char)*256}, IDETECT_TERM };

// size of parameters that can be received in the request mailbox:
size_t PARAM_SIZE_IN[] = { sizeof(TypeA), PARAM_SIZE_TERM };

// size of parameters or events that can be sent in request mailboxes of other
modules (user has to add sizes of sent param requests, if any):

size_t PARAM_SIZE_OUT[] = {/*PROTECTED REGION ID(protect_param_out_MAA) ENABLED
START*/ /*PROTECTED REGION END*/PARAM_SIZE_TERM };


//global variables
FILE* logFile;
RTIME productionTime, receptionTime;

//from here you can define new internal variables
/*PROTECTED REGION ID(protect_internal_vars_MAA) ENABLED START*/
int _connected = 0;
double _increment = 1.0;
/*PROTECTED REGION END*/
```

```
// Generic Prototypes (to be implemented by the user)
int _init_module(void);//==0 false, != 0 true
void _exit_module(void);
void _exec_module_state(int * stateindex);//state from 0 to x with x > 0, if
stateindex is -1 at end of execution then run is finished
void _react_to_event(int eventid ,  int senderport, char * sender, RTIME *
eventProductionTime, RTIME * eventReceptionTime);//to implement to react to some
events
void _react_to_parameterizing(int parameterport, RTIME * productionTime, RTIME *
receptionTime);//to implement to code reaction on parameterizing actions
//Specific prototypes (from here you can declare new internal functions)
/*PROTECTED REGION ID(protect_internal_functions_decl_MAA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/

int ModuleMain(int argc, const char * argv[])
{
RequestMessage receivedMessage, registered;
int subscription = 0;
int end=0;
int state=0;
int interact,cycleRunning=0;
//log file initialization
time_t log_date; time(&log_date); char* date = ctime(&log_date) ;
char date_jj[5], date_mm[5]; int date_j,date_h,date_m,date_s,date_a;
sscanf(date, "%s %s %d %d:%d:%d %d", date_jj, date_mm, &date_j, &date_h,
&date_m, &date_s,&date_a);
char filePath[100];
sprintf(filePath,"./%s_%d-%s-%d-%dh%d %ds.csv", MODULE_NAME,date_j, date_mm,
date_a, date_h,date_m,date_s);
logFile = fopen(filePath,"at");
if(logFile == NULL) {printf("ERR:%s opening log file(%s)\n",MODULE_NAME,
filePath);}

if( _init_module() == 0 )  return(-1);
receivedMessage.Type=REQ_UNKNOWN;
while(!end)
{
          switch(receivedMessage.Type)
          {
          case REQ_UNKNOWN :
          if (cycleRunning){
                    if (state != -1){ //the execution has not been interrupted
                    //filling the received message with old START request
information
                              receivedMessage.Type=REQ_START;
                              if ((registered.Param.Start.WantedAck&ACK_NOTHING)
== 0)//do not try to copy module names if an ACK is not required
                                        strcpy( receivedMessage.Param.Start.ModAck,
registered.Param.Start.ModAck);
                              receivedMessage.Param.Start.WantedAck=
registered.Param.Start.WantedAck;
                              receivedMessage.Param.Start.AckID =
registered.Param.Start.AckID;
                    }
                    else {//the execution has been interrupted, waiting again
                              state = 0;
                              //sending an acknowledge to the module that has
```

```
started this module
                                if ((registered.Param.Start.WantedAck &ACK_STOPPED)
!= 0){
                                        SendAck( registered.Param.Start.ModAck,
registered.Param.Start.WantedAck,  ACK_STOPPED, registered.Param.Start.AckID);
                                }
                                //reinitializing registered
                                registered.Type=REQ_UNKNOWN;
                                registered.Param.Start.WantedAck = ACK_NOTHING;
                                registered.Param.Start.AckID = -1;
                                //waiting a new request
                                WaitForRequest(&receivedMessage);
                                break;
                        }
                }
                else {
                        WaitForRequest(&receivedMessage);
                        break;
                }

                case REQ_START:
                if (cycleRunning == 0){//if execution cycle does not run
                        if ((receivedMessage.Param.Start.WantedAck&ACK_START) != 0)
//if an ACK is required by the caller
                        SendAck( receivedMessage.Param.Start.ModAck,
receivedMessage.Param.Start.WantedAck,  ACK_START,
receivedMessage.Param.Start.AckID);
                }
                if (subscription == 0){
                        SubscribeVariables();
                        SubscribeEvent();
                        subscription = 1;
                }
                interact = 0;
                cycleRunning = 1;
                while(!interact){
                        _exec_module_state(&state);
                        if(state==-1){//termination of the run
                                state = 0;
                                cycleRunning = 0;
                                interact = 1;
                                receivedMessage.Type=REQ_UNKNOWN;
                                if((receivedMessage.Param.Start.WantedAck
                                &ACK_FINISHED) != 0) //if an ACK is required by the
caller
                                SendAck( receivedMessage.Param.Start.ModAck,
receivedMessage.Param.Start.WantedAck,  ACK_FINISHED,
receivedMessage.Param.Start.AckID);
                        }
                        if(cycleRunning){
                                //registering the previous request
                                registered.Type=receivedMessage.Type;
                                registered.Param.Start.WantedAck=
receivedMessage.Param.Start.WantedAck;
                                registered.Param.Start.AckID =
receivedMessage.Param.Start.AckID;
                                if ((registered.Param.Start.WantedAck&ACK_NOTHING)
== 0)//do not try tocopy module names if an ACK is not required
                                        strcpy(registered.Param.Start.ModAck,rece
```

```
ivedMessage.Param.Start.ModAck);

                                // polling on message to be reactive
                                if (LookForRequest(&receivedMessage)!= NULL)
                                        interact=1;
                        }
                }
                break;

                case REQ_STOP:
                if (cycleRunning) {
                        state = -1;
                        //sending an acknowledge
                        if ((receivedMessage.Param.Stop.WantedAck&ACK_STOPPED) !=
0)//if an ACK is required
                        SendAck( receivedMessage.Param.Stop.ModAck,
receivedMessage.Param.Stop.WantedAck,  ACK_STOPPED,
receivedMessage.Param.Stop.AckID);
                }
                else {
                        state = 0;
                        if ((receivedMessage.Param.Stop.WantedAck&ACK_STOPPED) != 0)
{//if an ACK is required
                        SendAck( receivedMessage.Param.Stop.ModAck,
receivedMessage.Param.Stop.WantedAck,  ACK_NOTDONE,
receivedMessage.Param.Stop.AckID);
                        }
                }
                receivedMessage.Type=REQ_UNKNOWN;
                break;

                case REQ_KILL:
                        end = 1;
                        if ((receivedMessage.Param.Kil.WantedAck&ACK_KILLED) != 0)
//if an ACK is required
                        SendAck( receivedMessage.Param.Kil.ModAck,
receivedMessage.Param.Kil.WantedAck,  ACK_KILLED,
receivedMessage.Param.Kil.AckID);
                        receivedMessage.Type=REQ_UNKNOWN;
                break;

                case REQ_EVENT:
                _react_to_event(GetEventId(&productionTime,&receptionTime),
receivedMessage.Param.Event.sourcePort,
&receivedMessage.Param.Event.sourceModule[0], &productionTime, &receptionTime);
                receivedMessage.Type=REQ_UNKNOWN;
                break;

                case REQ_SETPARAM:

                switch(receivedMessage.Param.SetParam.Port)
                case 0 :
                GetDataFromSetParamRequest(&receivedMessage, &param, sizeof(TypeA),
&productionTime, &receptionTime);
                _react_to_parameterizing(0,&productionTime, &receptionTime);
                break;

                default:
                printf("Unknown param port:%d\n",receivedMessage.Param.SetParam.Port);
```

```
          break;
          }
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_SUBSCRIBE:
          SubscribeUser(receivedMessage.Param.Subscribe.SourcePort,
receivedMessage.Param.Subscribe.TargetModule,
receivedMessage.Param.Subscribe.TargetPort,
receivedMessage.Param.Subscribe.Periodicity,receivedMessage.Param.Subscribe.isCo
ntextual);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_UNSUBSCRIBE:
          UnsubscribeUser(receivedMessage.Param.Unsubscribe.SourcePort,
receivedMessage.Param.Unsubscribe.TargetModule,
receivedMessage.Param.Unsubscribe.TargetPort);
          receivedMessage.Type=REQ_UNKNOWN;
          break;
          case REQ_SUB_EVENT:
           SubscribeUserToEvent(receivedMessage.Param.SubscribeEvent.targetModule
, receivedMessage.Param.SubscribeEvent.targetPort,
receivedMessage.Param.SubscribeEvent.sourcePort,
receivedMessage.Param.SubscribeEvent.mode);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_UNSUB_EVENT:
           UnsubscribeUserToEvent(receivedMessage.Param.UnsubscribeEvent.targetMo
dule, receivedMessage.Param.UnsubscribeEvent.targetPort,
receivedMessage.Param.UnsubscribeEvent.sourcePort);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_PURGE_MBX:
          PurgeData(NULL);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          default:
          receivedMessage.Type=REQ_UNKNOWN;
          break;
          }
}
UnsubscribeVariables();
UnsubscribeEvent();
_exit_module();
fclose(logFile);
return 0;
}


int _init_module(void){//==0 false, != 0 true
          /*PROTECTED REGION ID(protect_init_MAA) ENABLED START*/
          printf("initialization of MAA\n");
          _connected = 0;
          return 1;
          /*PROTECTED REGION END*/
```

```
}

void _exit_module(void){
          /*PROTECTED REGION ID(protect_exit_MAA) ENABLED START*/
          printf("termination of MAA\n");
          /*PROTECTED REGION END*/
}

void _exec_module_state(int * stateindex){//state from 0 to x with x > 0, if
stateindex is -1 at end of execution then run is finished
          /*PROTECTED REGION ID(protect_exec_MAA) ENABLED START*/
          char input[256] = "";

                    switch(*stateindex){
                    case 0:
                              printf("MAA : cycle on state 0\n");
                              printf("USER : enter a command : %s\n",
(_connected==1?"1)disconnection 2) receiving message 3)exitting (and
disconnecting) 4)Loop":"1)connection 2) exitting 3)Loop"));
                              scanf("%s", &input[0]);

                              if (_connected==1){
                                        if (strcmp(input,"1")==0){

          strcpy(connection_status,"disconnected");
                                                  PostEvent(&connection_status[0]);
                                                  printf("disconnection => posting
event\n");
                                                  _connected = 0;

                                        }
                                        else if(strcmp(input,"2")==0){


          strcpy(received_message.message.message , "FUFU");

          received_message.message.messagetype = 1;
                                                  received_message.message.value.i
= received_message.message.value.i + _increment;
                                                  received_message.message.value.j
= received_message.message.value.j + _increment;
                                                  received_message.message.value.k
= received_message.message.value.k + _increment;

          strcpy(&received_message.data_.dataname[0], "0");
                                                  received_message.data_.datavalue
= 678;
                                                  PostEvent(&received_message);
                                                  printf("posting the whole message
is done !\n");

          set_TypeC(&message_type_notification,&received_message.message);

          PostEvent(&message_type_notification);
                                                  printf("posting the message type
is done !\n");
                                        }
                                        else if(strcmp(input,"3")==0){
                                                  printf("exitting\n");
```

```
		strcpy(connection_status,"disconnected");
                                                PostEvent(&connection_status[0]);
                                                printf("disconnection => posting
event\n");
                                                *stateindex = 1;
                                                _connected = 0;
                                        }
                                        else if(strcmp(input,"4")==0){
                                                printf("loop\n");
                                        }
                                        else{
                                                printf("BAD INPUT : retry\n");
                                        }

                                }
                                else{
                                        if (strcmp(input,"1")==0){
                                                printf("connection => posting
event\n");

		strcpy(connection_status,"connected");
                                                PostEvent(&connection_status[0]);
                                                printf("connection => event
posted\n");
                                                _connected = 1;

                                        }
                                        else if(strcmp(input,"2")==0){
                                                printf("exitting\n");
                                                *stateindex = 1;
                                        }
                                        else if(strcmp(input,"3")==0){
                                                printf("loop\n");
                                        }
                                        else{
                                                printf("BAD INPUT : retry\n");
                                        }
                                }
                                break;
                        case 1:
                                printf("MAA : state 1 => terminal state (do
nothing)\n");
                                *stateindex = -1;
                                break;

                        default:
                                break;
                        }
		/*PROTECTED REGION END*/
}
void _react_to_event(int receiverport , int senderport, char * sendermodule,
RTIME * eventProductionTime, RTIME * eventReceptionTime){
		/*PROTECTED REGION ID(protect_event_MAA) ENABLED START*/
		//TODO : implement the code for reaction to event notifications
		/* example pattern for implementing this function
		if (strlen(sendermodule) != 0){//event incoming in a port of the
module
                        switch(receiverport){
```

```
                    case 0:
                              ...
                              break;
                    case 1:
                              ...
                              break;
                    }
          }
          else{//event coming from the specified port of the sender module
                    ...
          }

          */
          /*PROTECTED REGION END*/
}
void _react_to_parameterizing(int parameterport, RTIME * productionTime, RTIME *
receptionTime){
          /*PROTECTED REGION ID(protect_param_MAA) ENABLED START*/
          //TODO : implement the code for reaction to parameterizing requests
          /* example pattern for implementing this function
          switch(parameterport){
          case 0:
                    ...
                    break;
          case 1:
                    ...
                    break;
          case 2:
                    ...
                    break;
          }
          */
          /*PROTECTED REGION END*/
}
//Specific functions (from here you have to implement new internal functions
declared before main)
/*PROTECTED REGION ID(protect_internal_functions_impl_MAA) ENABLED START*/
//TODO
/*PROTECTED REGION END*/
```

Fichier `module.c` du module `MSC`

```
#include <libmodule/libmodule.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include "tools.h"
#include "SharedTypeA.h"
#include "SharedTypeB.h"
#include "SharedTypeD.h"


//from here you can make new imports or definitions
/*PROTECTED REGION ID(protect_additionnal_import_MSC) ENABLED START*/
//TODO
/*PROTECTED REGION END*/
```

```
MODULE_DESCRIPTION(/*PROTECTED REGION ID(protect_description_MSC) ENABLED
START*/"test synchronous module C for Cactal generation testing"/*PROTECTED
REGION END*/);
MODULE_AUTHOR(/*PROTECTED REGION ID(protect_author_MSC) ENABLED START*/"Robin
Passama"/*PROTECTED REGION END*/);

//Exchanges Management Variables
TypeA in_coordinates = { 0.0, 0.0, 0.0 };
TypeB out_coordinates_commmand = { {1.0, 1.0, 1.0}, 1 };
TypeD failing_data = { "\0", 0 };
TypeA param = {2.1, 6.98, 5.34};


// module name
char MODULE_NAME[] = "MSC";
//module priority
int MODULE_PRIORITY = 40;
// input data flows : example = { &Variable, PortReception, Taille, Periodicity
(important if module name is set), "Name of emitter module"(can be unused by
setting value to "---"), PortModuleEmetteur} ou { &Variable, PortReception,
Taille, Periodicite, "---", 0}
ModuleUse IUSE[] = { { &in_coordinates, 0, sizeof(TypeA), "---", 0, 0},
IUSE_TERM };

// output data flows : example = { &Variable, PortEmision, Size}
ModuleProduce IPRODUCE[] = { { &out_coordinates_commmand, 1, sizeof(TypeB)},
IPRODUCE_TERM };

//input event flows : example = { &Variable, PortReception, size, "module name"
(can be unused by setting value to "---"), PortEmision, oneshot}
ModuleReact IREACT [] = { IREACT_TERM };

//output event flows : example = { &variable, PortEmision, size }
ModuleDetect IDETECT [] = { { &failing_data, 1, sizeof(TypeD)}, IDETECT_TERM };

// size of parameters that can be received in the request mailbox:
size_t PARAM_SIZE_IN[] = { sizeof(TypeA), PARAM_SIZE_TERM };

// size of parameters or events that can be sent in request mailboxes of other
modules (user has to add sizes of sent param requests, if any):

size_t PARAM_SIZE_OUT[] = {/*PROTECTED REGION ID(protect_param_out_MSC) ENABLED
START*/ /*PROTECTED REGION END*/PARAM_SIZE_TERM };

//global variables
FILE* logFile;
RTIME productionTime, receptionTime;

//from here you can define new internal variables
/*PROTECTED REGION ID(protect_internal_vars_MSC) ENABLED START*/
//TODO
/*PROTECTED REGION END*/

// Generic Prototypes (to be implemented by the user)
int _init_module(void);//==0 false, != 0 true
void _exit_module(void);
void _exec_module_state(int * stateindex);//state from 0 to x with x > 0, if
stateindex is -1 at end of execution then run is finished
```

```
//Specific prototypes (from here you can declare new internal functions)
/*PROTECTED REGION ID(protect_internal_functions_decl_MSC) ENABLED START*/
//TODO
/*PROTECTED REGION END*/


int ModuleMain(int argc, const char * argv[])
{
RequestMessage receivedMessage, registered;
int subscription = 0;
int end=0;
int state=0;
int interact,cycleRunning=0;
//log file initialization
time_t log_date; time(&log_date); char* date = ctime(&log_date) ;
char date_jj[5], date_mm[5]; int date_j,date_h,date_m,date_s,date_a;
sscanf(date, "%s %s %d %d:%d:%d %d", date_jj, date_mm, &date_j, &date_h,
&date_m, &date_s,&date_a);
char filePath[100];
sprintf(filePath,"./%s_%d-%s-%d-%dh%d %ds.csv", MODULE_NAME,date_j, date_mm,
date_a, date_h,date_m,date_s);
logFile = fopen(filePath,"at");
if(logFile == NULL) {printf("ERR:%s opening log file(%s)\n",MODULE_NAME,
filePath);}

if( _init_module() == 0 )  return(-1);
receivedMessage.Type=REQ_UNKNOWN;
while(!end)
{
          switch(receivedMessage.Type)
          {
          case REQ_UNKNOWN :
          if (cycleRunning){
                    if (state != -1){ //the execution has not been interrupted
                    //filling the received message with old START request
information
                              receivedMessage.Type=REQ_START;
                              if ((registered.Param.Start.WantedAck&ACK_NOTHING)
== 0)//do not try to copy module names if an ACK is not required
                                        strcpy( receivedMessage.Param.Start.ModAck,
registered.Param.Start.ModAck);
                              receivedMessage.Param.Start.WantedAck=
registered.Param.Start.WantedAck;
                              receivedMessage.Param.Start.AckID =
registered.Param.Start.AckID;
                    }
                    else {//the execution has been interrupted, waiting again
                              state = 0;
                              //sending an acknowledge to the module that has
started this module
                              if ((registered.Param.Start.WantedAck &ACK_STOPPED)
!= 0){
                                        SendAck( registered.Param.Start.ModAck,
registered.Param.Start.WantedAck,  ACK_STOPPED, registered.Param.Start.AckID);
                              }
                              //reinitializing registered
                              registered.Type=REQ_UNKNOWN;
                              registered.Param.Start.WantedAck = ACK_NOTHING;
                              registered.Param.Start.AckID = -1;
```

```
                                //waiting a new request
                                WaitForRequest(&receivedMessage);
                                break;
                        }
                }
                else {
                        WaitForRequest(&receivedMessage);
                        break;
                }

                case REQ_START:
                if (cycleRunning == 0){//if execution cycle does not run
                        if ((receivedMessage.Param.Start.WantedAck&ACK_START) != 0)
//if an ACK is required by the caller
                        SendAck( receivedMessage.Param.Start.ModAck,
receivedMessage.Param.Start.WantedAck,  ACK_START,
receivedMessage.Param.Start.AckID);
                }
                if (subscription == 0){
                        SubscribeVariables();
                        SubscribeEvent();
                        subscription = 1;
                }
                interact = 0;
                cycleRunning = 1;
                while(!interact){
                        _exec_module_state(&state);
                        if(state==-1){//termination of the run
                                state = 0;
                                cycleRunning = 0;
                                interact = 1;
                                receivedMessage.Type=REQ_UNKNOWN;
                                if((receivedMessage.Param.Start.WantedAck
                                &ACK_FINISHED) != 0) //if an ACK is required by the
caller
                                SendAck( receivedMessage.Param.Start.ModAck,
receivedMessage.Param.Start.WantedAck,  ACK_FINISHED,
receivedMessage.Param.Start.AckID);
                        }
                        if(cycleRunning){
                                //registering the previous request
                                registered.Type=receivedMessage.Type;
                                registered.Param.Start.WantedAck=
receivedMessage.Param.Start.WantedAck;
                                registered.Param.Start.AckID =
receivedMessage.Param.Start.AckID;
                                if ((registered.Param.Start.WantedAck&ACK_NOTHING)
== 0)//do not try tocopy module names if an ACK is not required
                                        strcpy(registered.Param.Start.ModAck,rece
ivedMessage.Param.Start.ModAck);

                                // polling on message to be reactive
                                if (LookForRequest(&receivedMessage)!= NULL)
                                        interact=1;
                        }
                }
                break;

                case REQ_STOP:
```

```
          if (cycleRunning) {
                    state = -1;
                    //sending an acknowledge
                    if ((receivedMessage.Param.Stop.WantedAck&ACK_STOPPED) !=
0)//if an ACK is required
                    SendAck( receivedMessage.Param.Stop.ModAck,
receivedMessage.Param.Stop.WantedAck,  ACK_STOPPED,
receivedMessage.Param.Stop.AckID);
          }
          else {
                    state = 0;
                    if ((receivedMessage.Param.Stop.WantedAck&ACK_STOPPED) != 0)
{//if an ACK is required
                    SendAck( receivedMessage.Param.Stop.ModAck,
receivedMessage.Param.Stop.WantedAck,  ACK_NOTDONE,
receivedMessage.Param.Stop.AckID);
                    }
          }
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_KILL:
                    end = 1;
                    if ((receivedMessage.Param.Kil.WantedAck&ACK_KILLED) != 0)
//if an ACK is required
                    SendAck( receivedMessage.Param.Kil.ModAck,
receivedMessage.Param.Kil.WantedAck,  ACK_KILLED,
receivedMessage.Param.Kil.AckID);
                    receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_EVENT:
                    receivedMessage.Type=REQ_UNKNOWN;
          break;


          case REQ_SETPARAM:
          switch(receivedMessage.Param.SetParam.Port){
                    case 0 :
                    GetDataFromSetParamRequest(&receivedMessage, &param,
sizeof(TypeA),  &productionTime, &receptionTime);
                    break;

                    default:
                    printf("Unknown param port:%d\n",
                    receivedMessage.Param.SetParam.Port);
                    break;
          }
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_SUBSCRIBE:
          SubscribeUser(receivedMessage.Param.Subscribe.SourcePort,
receivedMessage.Param.Subscribe.TargetModule,
receivedMessage.Param.Subscribe.TargetPort,
receivedMessage.Param.Subscribe.Periodicity,receivedMessage.Param.Subscribe.isCo
ntextual);
          receivedMessage.Type=REQ_UNKNOWN;
          break;
```

```
          case REQ_UNSUBSCRIBE:
          UnsubscribeUser(receivedMessage.Param.Unsubscribe.SourcePort,
receivedMessage.Param.Unsubscribe.TargetModule,
receivedMessage.Param.Unsubscribe.TargetPort);
          receivedMessage.Type=REQ_UNKNOWN;
          break;
          case REQ_SUB_EVENT:
           SubscribeUserToEvent(receivedMessage.Param.SubscribeEvent.targetModule
, receivedMessage.Param.SubscribeEvent.targetPort,
receivedMessage.Param.SubscribeEvent.sourcePort,
receivedMessage.Param.SubscribeEvent.mode);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_UNSUB_EVENT:
           UnsubscribeUserToEvent(receivedMessage.Param.UnsubscribeEvent.targetMo
dule, receivedMessage.Param.UnsubscribeEvent.targetPort,
receivedMessage.Param.UnsubscribeEvent.sourcePort);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          case REQ_PURGE_MBX:
          PurgeData(NULL);
          receivedMessage.Type=REQ_UNKNOWN;
          break;

          default:
          receivedMessage.Type=REQ_UNKNOWN;
          break;
          }
}
UnsubscribeVariables();
UnsubscribeEvent();
_exit_module();
fclose(logFile);
return 0;
}

int _init_module(void){//==0 false, != 0 true
          /*PROTECTED REGION ID(protect_init_MSC) ENABLED START*/
          printf("initialization of MSC\n");
          return 1;
          /*PROTECTED REGION END*/
}

void _exit_module(void){
          /*PROTECTED REGION ID(protect_exit_MSC) ENABLED START*/
          printf("termination of MSC\n");
          /*PROTECTED REGION END*/
}

void _exec_module_state(int * stateindex){//state from 0 to x with x > 0, if
stateindex is -1 at end of execution then run is finished
          /*PROTECTED REGION ID(protect_exec_MSC) ENABLED START*/
          switch(*stateindex){
          case 0:
                    GetLatestData(&in_coordinates, &productionTime,
&receptionTime);
```

```
                    printf("MSC : state 0 => getting coordinates :
");print_TypeA(&in_coordinates);printf("\n");
                    *stateindex = 1;
                    break;
          case 1:

                    //computing
                    out_coordinates_commmand.coord.i = param.i-in_coordinates.i;
                    out_coordinates_commmand.coord.j =  param.j-in_coordinates.j;
                    out_coordinates_commmand.coord.k = param.k -
in_coordinates.k;
                    printf("MSC : state1 => computed out coordinates:
");print_TypeA(&out_coordinates_commmand.coord);printf("\n");
                    printf("MSC : state1 => param coordinates:
");print_TypeA(&param);printf("\n");
                    printf("MSC : state1 => in coordinates:
");print_TypeA(&in_coordinates);printf("\n");
                    //testing the event
                    if
((out_coordinates_commmand.coord.i+out_coordinates_commmand.coord.j+out_coordina
tes_commmand.coord.k)>30.0){
                              strcpy(&failing_data.dataname[0], "problem");
                              failing_data.datavalue = (int) (30.0-
(out_coordinates_commmand.coord.i+out_coordinates_commmand.coord.j+out_coordinat
es_commmand.coord.k));
                              PostEvent(&failing_data);
                              printf("MSC : posting event %s -
%ld\n",&failing_data.dataname[0],failing_data.datavalue);
                    }
                    *stateindex = 2;
                    break;
          case 2:
                    out_coordinates_commmand.frfrrrf = 666;
                    PostData(&out_coordinates_commmand.coord);
                    *stateindex = -1;
                    break;

          default:
                    break;
          }

          /*PROTECTED REGION END*/
}
//Specific functions (from here you have to implement new internal functions
declared before main)
/*PROTECTED REGION ID(protect_internal_functions_impl_MSC) ENABLED START*/
//TODO
/*PROTECTED REGION END*/
```