Bachelor Universitaire technologique - Automatisme pour la robotique S5

CodeSyS – Approche classique

Cours

v1

IUT d'Annecy, 9 rue de l'Arc en Ciel, 74940 Annecy

CodeSyS

Ce cours présente des rappels de l'utilisation du logiciel CodeSys dans une approche dite "Classique". Nous verrons dans un prochain cours une approche orientée objet appliquée à ce logiciel.

Sommaire

Un système automatisé centralisé consiste en une unique unité centrale qui gère l'ensemble des entrées et des sorties. Cette unité centrale est donc reliée à l'ensemble des capteurs et des actionneurs. Dans le domaine qui nous intéresse, l'unité centrale est un automate programmable industriel (API) et pourrait gérer une série d'étapes d'une chaîne de production.

Avantages
- Un automate unique.
- Architecture de données simple.

Inconvénients
- Nombre d'entrées/sorties important.
- Traitement d'automatisation complexe.
- Maintenance et évolutions difficiles.

Activité 1: Architecture centralisée

Question 1 Représenter l'architecture d'un système automatisé centralisé.

Un système automatisé modulaire et réparti consiste, à l'inverse, en plusieurs unités centrales qui gèrent chacune une partie de la chaine et donc des entrées et des sorties. Chaque unité centrale est reliée à un ensemble de capteurs et d'actionneurs. Un protocole de communication est mis en place entre les unités centrales.

Avantages

- Nombre d'entrées/sorties limité par unité centrale.
- Maintenance et évolutions propres à chaque unité.

Inconvénients

- Architecture de données complexe.
  - ◇ Orientation des données.
  - ◇ Perte de la sémantique des données qu'il faut reconstruire (union, chevauchement, ...).
- Protocole de communication à mettre en place.

Activité 2: Architecture modulaire

Question 2 Représenter l'architecture d'un système automatisé modulaire.

Critères de choix Le choix entre une architecture centralisée ou modulaire ainsi que les matériels et logiciels à utiliser se feront en fonction de différents critères :

- Nombre d'entrées/sorties TOR.
- Nombre d'entrées/sorties analogiques.
- Communications à mettre en place entre les unités centrales (protocoles, nombres).
- Maintenance et évolution.
- Interconnexion avec le Cloud (4.0).
- Puissance de calcul nécessaire.
- Boucle de régulation intégrée ou non.
- Coût (matériel, chaîne de développement, ...).
- Langages IEC 61131-3 supportés.
- Les extensions de ces langages pouvant conduire à une programmation orientée objet.

Aussi, le choix de l'environnement de développement (IDE) est important. Il doit permettre de programmer l'ensemble des unités centrales et de gérer les communications entre elles. Il doit aussi permettre de gérer les entrées/sorties et les communications avec les capteurs/actionneurs. Ainsi, sur une architecture complexe, on fera le choix d'un IDE permettant la programmation, dans un même projet, de matériels non homogènes (d'automates de marques et/ou modèles différents).

1.2 Le système d'exploitation

L'automate industriel programmable embarque un système d'exploitation temps réel. Ce système d'exploitation est un logiciel qui permet de gérer les ressources matérielles de l'automate (processeur, mémoire, entrées/sorties, ...) et de fournir des services aux applications (gestion des tâches, gestion des communications, ...).

Pour programmer efficacement et correctement un automate, il est nécessaire de comprendre, au moins sommairement, le fonctionnement du système d'exploitation.

1.2.1 Rappels : Le cycle automate

Un Automate Programmable Industriel (API) fonction en suivant un cycle périodique, appelé cycle automate dont la période est généralement comprise entre 1 ms et 100 ms. Ce cycle est composé de 4 phases répétées en boucle :

1. Traitement interne
   - Mise à jour des bits système
   - Requêtes en provenance de la console
   - horodatage
   - ...
2. Lecture des entrées
   - Le système met à jour l'état des entrées TOR et analogiques.
   - **Attention : l'état des entrées ne sera pas lu en dehors de cette étape**
3. Traitement des programmes
   - Exécution des programmes utilisateur.
   - Exécution des programmes système.
4. Ecriture des sorties
   - Le système met à jour l'état des sorties TOR et analogiques.
   - **Attention : c'est donc uniquement l'état de la sortie après exécution de tous les programmes qui sera écrit sur les sorties.**

automate_cycle.png

Système temps réel

Un système d'exploitation temps réel garantit un temps de réponse maximum à toute requête ou événement matériel.

Ce type de système d'exploitation est utilisé dans les systèmes embarqués, les automates programmables, les systèmes de contrôle-commande, ...En résumé, dans tous les domaines où la réactivité à un événement externe est primordiale.

1.2.2 Propriétés du système d'exploitation d'un API

Un système d'exploitation d'un API possède les propriétés suivantes :

Temps réel : par la mise en place d'un chien de garde (watchdog).

Interruptible : les tâches peuvent être interrompues par une tâche de **priorité supérieure** parce que l'événement qu'elle traite est **fugitif** ou d'**importance** pour l'application.

Mono/Multi-tâche : Plusieurs tâches peuvent être exécutées de façon apparemment simultanée.

Accès facilité aux entrées/sorties.

Accès facilité à la mémoire : Permet de partager des données entre les tâches, de gérer les accès concurrents à la mémoire, ...

Interaction avec le programme utilisateur .

Les événements sont capturés par des fonctions appelées **event handlers**. Ces fonctions sont automatiquement appelées par le système d'exploitation lorsqu'un événement défini se produit.

Système Multitâches

Un système multitâche permet d'exécuter plusieurs tâches "en même temps". En réalité, une stratégie de priorité des tâches est mis en place. Selon le nombre de processeurs présents et le nombre de tâches, elles seront exécutées en parallèle ou en alternance.

2 Environnement de programmation : CoDeSys

Dans ce module, nous programmerons sous l'environnement de développement CoDeSys.

2.1 Présentation

CoDeSys est un environnement de développement intégré (IDE) permettant de programmer des automates programmables industriels (API). Il est basé sur la norme IEC 61131-3. Il permet de programmer des automates de différentes marques (Beckhoff, Wago, Schneider, ...). Il est disponible sous Windows et sous Linux. Il est gratuit pour une utilisation non commerciale.

2.2 Langages

CoDeSys permet de programmer des automates en utilisant les langages suivants :
- Instruction List (IL).
- Ladder Diagram (LD).
- Function Block Diagram (FBD).
- Structured Text (ST).
- Sequential Function Chart (SFC).

2.3 Architecture d'un programme

CoDeSys permet de programmer en Monotâche et en Multitâche :
- 15 tâches périodiques au maximum.
- Priorité des taches de 1 à 15, de la plus prioritaire à la moins prioritaire.
- Temps de cycle compris entre 100 µs et 10 s.

2.4 Unité de Programme – Program Organisation Unit (POU)

Une unité de programme est un ensemble d'instructions qui peuvent être appelées depuis un autre programme. Il existe 3 types de POU :

Programme (Program – PRG) : Un programme consiste en une série d'instruction pouvant produire une ou plusieurs valeurs en sortie. Il peut être appelé depuis un autre programme ou depuis un bloc fonction. Toutes les valeurs des variables du programme restent inchangées entre deux appels, quelque soit sa provenance.

Bloc fonction (Function Block – FB) : Un bloc fonction est un ensemble d'instructions pouvant produire une ou plusieurs valeurs en sortie. Il peut être appelé depuis un autre programme ou depuis un bloc fonction. Un bloc fonction sera toujours appelé au travers d'une instance de bloc fonction, copie du bloc fonction. Par conséquent, les valeurs des variables du bloc fonction seront inchangées entre deux appels d'une même instance uniquement. Ce type de POU sera particulièrement adapté à la programmation orientée objet que nous développerons dans ce module.

Fonction (Function – FUN) : Une fonction ne peut produire qu'une seule valeur en sortie. Elle peut être appelée par n'importe quel POU et les valeurs des variables ne persistent pas d'un appel à l'autre.

Structure d'un POU

Un POU est divisé en 2 parties :
1. Déclaration des variables.
   - Variables d'entrées/sorties : **VAR_INPUT**, **VAR_OUTPUT**, **VAR_IN_OUT**.
   - Variables Locales : **VAR**, **CONSTANT**.

2. Implémentation du POU

METHOD et ACTION

Au sein d'une POU, il est possible de définir des méthodes et des actions.

ACTION : Portion de code toujours accessible qui ne peut utiliser que le **contexte** de l'entité à laquelle elle est associée.

Exemple : Une fonction RESET qui remet à zéro toutes les variables d'entrées/sorties d'un bloc fonction.

METHOD : Portion de code accessible qui utilise le contexte de l'entité à laquelle elle est associée, mais qui peut également créer son propre contexte. On peut définir **l'accessibilité** d'une méthode (PRIVATE, PROTECTED, PUBLIC, INTERNAL, ...).

2.5 Les types de données

2.5.1 Les types de données natifs

CodeSys propose un ensemble de types de données natifs. Ces types de données sont définis par la norme IEC 61131-3. Pour chaque type de données, le tableau suivant précise son empreinte mémoire ainsi que les règles de nommage (préfixe) que nous utiliserons dans ce module.

Type de données	Empreinte mémoire	Règles de nommage
BOOL	1 bit	x
BYTE	8 bits	by
SINT	8 bits	si
USINT	8 bits	usi
WORD	16 bits	w
INT	16 bits	i
UINT	16 bits	ui
DWORD	32 bits	dw
DINT	32 bits	d
UDINT	32 bits	udi
REAL	32 bits	r
LWORD	64 bits	lw
LINT	64 bits	li
ULINT	64 bits	uli
LREAL	64 bits	lr
STRING	Variable	s

TABLE 1 – Empreinte mémoire et règles de nommage des types de données natifs

2.5.2 Les types de données de datation

2.5.3 Les tableaux

Sous l'environnement CoDeSys, il est possible de déclarer des tableaux de variables. Ils peuvent être de dimension 1, 2 ou 3. Aussi, et contrairement au langage C, il est possible de définir le premier et le dernier indice. **Ainsi, le premier indice d'un tableau n'est pas forcément 0.**

Type de données	Empreinte mémoire	Règles de nommage
TIME (T#)	32 bits	tim
TIME_OF_DAY (TOD#)	32 bits	tod
DATE (D#)	32 bits	date
DATE_AND_TIME (DT#)	32 bits	dt

TABLE 2 – Empreinte mémoire et règles de nommage des types de données de datation

Déclaration d'un tableau

La déclaration d'un tableau se fait à l'aide de la syntaxe suivante :
`<nom> : ARRAY [<ll1>..<ul2>|,<ll2> <ul2>|,<ll3>..<ul3>] OF <type>;.`
Avec `<lli>` la borne inférieure et `<uli>` la borne supérieure de la dimension i.

Notre convention de nommage veut que l'on ajoute le prefixe `a<dim><type>` à la déclaration d'un tableau. Avec `<dim>` le nombre de dimensions et `<type>` le préfixe correspondant à celui des éléments du tableau (Cf Table **??**).

Par exemple, pour définir un tableau `tab` de taille 10 de type `WORD` et un tableau `tab2` de taille 20x10 de type `INT`, on écrira :

```
VAR
    a1wtab : ARRAY [0..9] OF WORD;
    a2itab2 : ARRAY [1..20, 1..10] OF INT;
END_VAR
```

Il est possible de récupérer le premier et le dernier indice d'un tableau à l'aide des fonctions `LOWER_BOUND(<array_name>,<dim>)` et `UPPER_BOUND(<array_name>,<dim>)`, respectivement.

Tableau à longueur variable

Lorsqu'un tableau est donné en `VAR_IN_OUT` d'une fonction, d'un bloc fonction ou d'une méthode, il est possible de ne pas préciser la taille du tableau par la syntaxe `<nom> : ARRAY [*|, *|, *] OF <type>`. Dans ce cas, le tableau est dit à longueur variable. Il est possible de récupérer la taille du tableau à l'aide de la fonction `SIZEOF(<array_name>)`.

Par exemple, pour définir un tableau `tab` de dimension 1 et de taille variable de type `WORD`, on écrira :

```
a1wtab : ARRAY [*] OF WORD;
```

2.5.4 Les structures

Définition d'une structure

La définition d'une structure se fait à l'aide de la syntaxe suivante :

```
TYPE <struct_name> :
    STRUCT
        <nom1> : <type1>;
        <nom2> : <type2>;
        ...
        <nomi> : <typei>;
    END_STRUCT
END_TYPE
```

Avec `<nomi>` le nom de la variable et `<typei>` le type de la variable.

Sous l'environnement CoDeSys, il est possible de déclarer des structures de variables. Notre convention de nommage veut que l'on ajoute le prefixe `T` au nom d'une variable de type structure et `m_<data_type><member_name>` au nom d'un membre de la structure.

Par exemple, pour définir une structure `TVerin` contenant deux membres `m_xSorti` et `m_xRENTRE` de type `BOOL`, on écrira :

```
TYPE TVerin :
    STRUCT
        m_xSorti : BOOL;
        m_xRENTRE : BOOL;
    END_STRUCT
END_TYPE
```

Dans une structure uniquement, il est possible de déclarer des membres de type BIT afin de ne réserver qu'un seul bit mémoire (contrairement au type BOOL qui réserve 1 octet mémoire). Cela peut permettre d'assigner un nom à différents bits d'un registre, par exemple.

Accès aux membres d'une structure : Il est possible d'accéder aux membres d'une structure à l'aide de la syntaxe `<struct_name>.<member_name>`. Par exemple, pour accéder au membre `m_xSorti` de la structure `enrVerin1` de type `TVerin`, on écrira `enrVerin1.m_xSorti`.

Héritage : Le mot clé EXTENDS permet de définir une structure héritant d'une autre structure. La structure fille hérite de tous les membres de la structure mère auxquels elle peut ajouter ses propres membres. La syntaxe de cet héritage simple est alors :

```
TYPE <struct_fille> EXTENDS <struct_mere> :
    STRUCT
        <nom_n+1> : <type_n+1>;
        <nom_n+2> : <type_n+2>;
        ...
    END_STRUCT
END_TYPE
```

2.5.5 Les énumérations

Une énumération est un type permettant de nommer des constantes entières. Elle augmente la lisibilité du code et permet de le rendre plus robuste. On peut par exemple définir une énumération ETAT contenant les constantes ETAT_OUVERT, ETAT_FERME, ETAT_EN_COURS et ETAT_ERREUR. Chaque ETAT correspond à une valeur entière que le développeur ne connait pas forcément.

Par défaut, le premier élément de l'énumération vaut 0 et chaque élément suivant vaut l'élément précédent + 1. Il est possible de modifier cette valeur en utilisant l'opérateur `|:=`. Par défaut, le type des valeurs de chaque élément est INT.

Définition d'une énumération

La définition d'une énumération se fait à l'aide de la syntaxe suivante :

```
TYPE <enum_name> :
    (
        <nom1> |:= <valeur1>,
        <nom2> |:= <valeur2>,
        ...
        <nomi> |:= <valeuri>
    )|<base_data_type>|:= <valeur_par_defaut>;
END_TYPE
```

Par exemple, pour définir une énumération ETAT contenant les constantes ETAT_OUVERT, ETAT_FERME, ETAT_EN_COURS et ETAT_ERREUR, on écrira :

```
TYPE ETAT :
    (
        ETAT_OUVERT,
        ETAT_FERME,
        ETAT_EN_COURS,
        ETAT_ERREUR
    ):= ETAT_OUVERT;
END_TYPE
```

Voici alors un exemple de déclaration, d'initialisation et d'utilisation d'une variable de type ETAT :

```
VAR
    etat : ETAT;
END_VAR
etat := ETAT_FERME;
IF etat = ETAT_FERME THEN
    (* Faire quelque chose *)
END_IF
```

Attention *Noms de constante identiques*

Si deux types énumération contiennent des constantes de même nom, il est alors impératif de préciser le type d'énumération. Par exemple, si on définit une énumération `ETAT` contenant les constantes `ETAT_OUVERT`, `ETAT_FERME`, `ETAT_EN_COURS` et `ETAT_ERREUR` et une énumération `ETAT_VERIN` contenant les constantes `ETAT_OUVERT`, `ETAT_FERME`, `ETAT_EN_COURS` et `ETAT_ERREUR`, il faudra alors écrire `ETAT.ETAT_OUVERT` et `ETAT_VERIN.ETAT_OUVERT` pour différencier les deux constantes.

2.5.6 Les unions

Une union est un type permettant d'utiliser une même zone mémoire pour stocker des données de types différents. Cela signifie que la modification de la valeur d'une variable de l'union modifiera la valeur des autres variables.

Puisqu'il est partagé entre tous les membres de l'union, l'espace mémoire occupé par une union est égal à la taille de la variable occupant le plus d'espace mémoire.

Définition d'une union

La définition d'une union se fait à l'aide de la syntaxe suivante :

```
TYPE <union_name> :
    UNION
        <nom1> : <type1>;
        <nom2> : <type2>;
        ...
        <nomi> : <typei>;
    END_UNION
END_TYPE
```

Nos conventions de nommage veulent que l'on ajoute le prefixe `T` au nom d'une variable de type union et `m_<data_type><member_name>` au nom d'un membre de l'union.

Enfin, des types de données spécifiques sont définis pour les unions :

ANY_TYPE : Type de données permettant de définir une union de type quelconque. Uniquement utilisable pour les interfaces

WSTRING : Type de données permettant de définir une union de type chaine de caractères. Ce type de données est similaire au type STRING, mais l'encodage utilisé ici est UniCode au lieu de ASCII.

Remarque *Le type WSTRING*

Contrairement au STRING, le nombre de caractères affichable par une variable de type WSTRING d'une taille donnée dépend des caractères utilisés. En effet, certains caractères sont codés sur 1 octet, d'autres sur 2 octets. Par exemple, le caractère `A` est codé sur 1 octet, tandis que le caractère `é` est codé sur 2 octets.

2.5.7 Les pointeurs

Rappel *Pointeur*

Un pointeur est une variable contenant l'adresse mémoire d'une autre variable. Elle permet, de partager l'accès à une zone mémoire (variable) entre différents contextes. En langage C, par exemple, cela permet de créer des fonctions capables de modifier des variables locales à la fonction appelante.

Lorsque vous avez utilisé la fonction `scanf`, vous avez passé en paramètre l'adresse de la variable à modifier à l'aide de l'opérateur `&`. Cela permet à la fonction `scanf` de modifier la valeur de la variable passée en paramètre.

Définition d'un pointeur

La définition d'un pointeur se fait à l'aide de la syntaxe suivante :

```
<pointer_name> : POINTER TO <type>;
```

Ici, `<type>` peut désigner le type d'une variable, une fonction, un bloc fonction, une méthode ou même un POU.

Les opérateurs disponibles pour l'utilisation de pointeurs sont les suivants :

- `^` : Opérateur de déréférencement. Il permet d'accéder à la valeur pointée par le pointeur.
- `ADR` : Opérateur d'adressage. Il permet d'accéder à l'adresse mémoire pointée par le pointeur.
- `[ ]` : Opérateur d'indexation : Il permet d'accéder à un élément d'un tableau pointé par un pointeur.

Nos règles de nommage veulent que l'on ajoute le prefix `p` au nom d'une variable de type pointeur.

Exemple:

```
VAR
    piA : POINTER TO INT := 0;
    iAlpha AT %MW1: INT := 7;
    iBeta AT %MW3: INT := 4;
    piB : POINTER TO INT := ADR(iBeta);
    dwGapInBytes: DWORD; (* en octet *)
END_VAR
```

```
(* Traitement *)
piA := ADR (iAlpha);
piA^ := 0 ; (* <=> iAlpha := 0; *)
piB[0] := 0 ; (* <=> iBeta := 0; *)
dwGapInBytes := piB - piA; (* = 4 *)
```

2.5.8 Les références

Sous CodeSys, une référence possède des propriétés similaires à celles d'un pointeur. Elle présente des avantages et inconvénients suivants par rapport à un pointeur :

- Plus simple à utiliser : pas besoin d'utiliser l'opérateur `^` pour accéder à la valeur pointée.
- Syntaxe plus simple
- Sécurisée sur le type : le compilateur vérifie que la référence pointe bien sur une variable du type attendu.
- Il est impossible de modifier la variable pointée par la référence,
- On ne peut référencer que des variables (pas de fonctions, blocs fonction, méthodes ou POU).

La fonction `__ISVALIDREF(ref_int` permet de vérifier si une référence est valide ou non. Elle renverra `TRUE` si la référence est valide (pointe vers une variable et non 0), `FALSE` sinon.

Exemple:

```
VAR
    ref_int: REFERENCE TO INT := 0;
    iAlpha: INT := 7;
    iBeta: INT := 4;
    xRef: BOOL;
END_VAR
```

```
(* Traitement *)
xRef := __ISVALIDREF (ref_int); (* FALSE *)
ref_int REF := iAlpha ; (*<=> ADR (iAlpha); *)
xRef := __ISVALIDREF (ref_int) ; (* TRUE *)
ref_int := 0 ; (*<=> ADR(iAlpha)^:= 0 ; *);
ref_int REF := iBeta (*<=> ADR (iBeta)*);
ref_int := 0 ; (* <=> iBeta := 0 ; *)
```

2.6 Le langage SFC

Le langage SFC (*Sequential Function Chart*) est un langage graphique permettant de décrire le comportement d'un système automatisé. Il est basé sur le formalisme des GRAFCET (*GRAphe Fonctionnel de Commande Etape Transition*).

Remarque

Bien qu'ils soient très similaires à première vue, le GRAFCET est un langage destiné à la spécification alors que le SFC est un langage destiné à la programmation.

Rappels sur le SFC

Un graph SFC consiste en une succession d'**étapes** et de **transitions**. A l'origine, une seule étape initiale est active. A chaque cycle, les transitions sont évaluées. Une transition est dite **franchissable** si sa **réceptivité** est `TRUE` et que l'étape qui la précède est active. Si une transition est franchissable, elle est franchie et l'étape suivante devient active. Lorsqu'une étape est active, elle exécute l'ensemble des **actions** qui lui sont associées.

Rappel de vocabulaire :

Étape : Une étape est un ensemble d'actions qui s'exécutent en parallèle. Une étape est représentée par un rectangle.

Transition : Une transition est un ensemble de conditions qui doivent être vérifiées pour passer d'une étape à une autre. Une transition est représentée par un trait horizontal.

Action : Une action est une opération élémentaire qui peut être réalisée par le système. Une action est représentée par un rectangle vertical.

Réceptivité : Une réceptivité est une condition booléenne qui doit être vérifiée pour qu'une transition soit franchissable.

2.6.1 Les actions en SFC

Sous CodeSys, l'activation d'une étape entraîne l'exécution de l'ensemble des actions qui lui sont associées. Elles peuvent être exécutées :

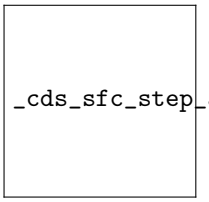

FIGURE 1 – Action en SFC

1. Une seule fois à l'activation de l'étape (Les **Actions d'entrées** – **Entry actions**).
   - La présence d'une action d'entrée est indiquée par un E dans le coin inférieur gauche de l'étape.
2. Exécutée à chaque cycle tant que l'étape est active (Les **Actions actives** – **actions**).
   - La présence d'une action active est indiquée par un triangle dans le coin supérieur droit de l'étape.
3. Une seule fois à la désactivation de l'étape (Les **Actions de sortie** – **Exit actions**).
   - La présence d'une action de sortie est indiquée par un X dans le coin inférieur droit de l'étape.

A chaque action est associée un qualificatif dont la signification est donnée dans le tableau suivant :

Qualificatif	**Nom**	**Description**
N	Non-stored	L'action est exécutée tant que l'étape est active
S0	Stored	L'action est exécutée dès que l'étape est active. Elle restera active jusqu'à ce qu'elle soit désactivée (même après que l'étape n'est plus active)
R0	Reset	L'action est désactivée et cesse donc d'être exécutée
L	time Limited	L'action commence son exécution à l'activation de l'étape et s'arrête après un temps défini ou lorsque l'étape est désactivée
D	time Delayed	L'action commence son exécution après un temps défini suivi l'activation de l'étape et s'arrête lorsque l'étape est désactivée
P	Pulse	L'action est exécutée une seule fois à l'activation de l'étape
SD	Stored and time Delayed	L'action commence son exécution après un temps défini suivi l'activation de l'étape, même si cette dernière n'est plus active. L'action s'exécute tant qu'elle n'est pas désactivée (même après que l'étape n'est plus active)
DS	Delayed and Stored	L'action commence son exécution après un temps défini suivi l'activation de l'étape, si celle-ci est toujours active. L'action s'exécute tant qu'elle n'est pas désactivée (même après que l'étape n'est plus active)
SL	Stored and time Limited	L'action commence son exécution à l'activation de l'étape et s'arrête après un temps défini ou lorsque l'étape est désactivée.

2.6.2 Les transitions en SFC

La réceptivité associée à une transition sous CodeSys peut être vue comme l'appel d'une fonction. En effet, il est possible d'exécuter un algorithme pour évaluer la réceptivité d'une transition.

2.6.3 Les drapeaux d'un graph SFC

Les drapeaux sont des variables qui (souvent booléennes) qui permettent le contrôle et le suivi de l'évolution d'un graph SFC. Le tableau suivant décrit les principaux drapeaux que nous rencontrerons :

`SFCInit : BOOL`	Son passage à `TRUE` entraine l'initialisation normalisée du graph SFC et de tous les drapeaux. Tant qu'elle est à `TRUE`, le graph SFC reste dans son état initial, sans qu'aucune action ne soit exécutée.
`SFCReset : BOOL`	Son passage à `TRUE` entraine la réinitialisation du graph SFC et de tous les drapeaux. Tant qu'elle est à `TRUE`, le graph SFC reste dans son état initial et **l'action associée à l'étape initiale est exécutée**.
`SFCPause : BOOL`	Son passage à `TRUE` entraine la mise en pause du graph SFC.
`SFCTrans : BOOL`	Passe à `TRUE` à chaque franchissement d'une transition.
`SFCCurrentStep : STRING`	Contient le nom de l'étape active. Si plusieurs étapes sont active, celui de l'étape la plus à droite.

3 Programmation classique hiérarchisée à l'aide du GMMA

Cette section propose un paradigme de programmation d'une application automatisée à l'aide du GMMA sur l'application e !Cockpit.

L'application consiste en une tâche principale nommée *MainTask* qui est exécutée périodiquement. Cette tâche appelle une seule unité de programme (POU) : *MainProgram* qui mettra en oeuvre un GMMA en langage SFC. Chaque étape du GMMA sera associée à une POU correspondant au mode à jouer. Enfin, la POU de chaque mode sera implémentée dans un langage adapté au type de tâche à réaliser. Enfin, on ajoute deux gestionnaires d'événements :

Ce paradigme permet de respecter une hiérarchisation des tâches et des POUs : Le GMMA est préemptif et autorise ou non les différents modes en fonction de l'état du système.

La tâche principale fonctionne avec une priorité comprise entre 4 et 15. Elle est exécutée cycliquement avec une période qui doit être compatible avec la dynamique du procédé. Une surveillance doit être mise en place par l'intermédiaire d'un watchdog (chien de garde) réglé avec une sensibilité de 1. Cette tâche appelle alors une seule POU : le programme principal *MainProgram*.

Dynamique du procédé

La dynamique du procédé correspond à la vitesse à laquelle le procédé peut évoluer. Afin de fonctionner correctement, le système automatisé doit être capable de réagir à tous les événements du procédé. Pour cela, la période de la tâche principale doit être inférieure à cette dynamique.

Pour déterminer ce temps, on peut effectuer une étude s'intéressant à l'entrée la plus rapide qui doit être prise en compte par le système automatisé. Le cycle de l'automate doit alors être plus rapide que le temps de cette entrée.

Par exemple, si le système automatisé doit réagir à une entrée TOR, il faut déterminer le temps de changement d'état de cette entrée et s'assurer que le cycle de l'automate est plus rapide que ce temps en configurant le watchdog.

La POU MainProgram , appelée par la tâche principale, gère la POU associée au GMMA. Cela signifie qu'elle appellera la POU du GMMA après avoir effectué les vérifications nécessaires. Par exemple, elle pourra faire un reset du GMMA au premier cycle de l'automate.

La POU du GMMA est une POU de type SFC. Elle est composée de plusieurs étapes. Chaque étape est associée à une POU correspondant à un mode de fonctionnement. A chaque étape, on utilise les actions d'entrée pour paramétrer le mode puis l'action active pour que la POU correspondante soit appelée à chaque cycle automate.

Les POU des modes sont des POUs implémentées en ST ou en SFC. Elles sont appelées par la POU du GMMA et sont exécutées à chaque cycle automate si le mode correspondant est actif.

Les gestionnaires d'événements sont des POUs implémentés en ST ou en SFC. Ils permettent de gérer les événements qui ne sont pas liés à un mode de fonctionnement. Typiquement, on définira un événement pour le premier cycle (*OnFirstRun*) pour effectuer diverses initialisations (par exemple un booléen global *xFirstCycle-Run* qui fera un reset du GMMA) et un événement en fin de cycle (*OnPrepareStop*) qui pourra fixer les modes de repli si nécessaire. On peut aussi utiliser, par exemple, un gestionnaire d'événement pour gérer les alarmes.

Exemple Cette section présente un code d'exemple d'application du GMMA dans une structure hiérarchisée.

Variable globales (Global Variable List – GVL) : On définit une variable globale indiquant que l'on se trouve dans le premier cycle :

```
VAR_GLOBAL
    xFirstCycleRun : BOOL;
END_VAR
```

2 Events Handlers : On écrit une fonction pour le premier cycle :

```
FUNCTION OnFirstRun : DWORD
    VAR_IN_OUT
        EventPrm : CmpApp.EVTPARAM_CmpApp;
    END_VAR

    GVL.xFirstCycleRun := TRUE;
    OnFirstRun := 0;
```

Tâche principale La tâche principale est configurée avec une période de 40 ms et une priorité de 4. Elle appelle la POU *MainProgram*. Le watchdog est configuré avec une sensibilité de 1 et un temps de 60 ms.

Le programme principal (MainProgram) : Le programme principal, en langage ST, s'occupe de lancer correctement le GMMA :

```
PROGRAM MainProgram
    VAR
        xSFRResetGmma : BOOL := FALSE;
    END_VAR

    IF GVL.xFirstCycleRun THEN
        xSFRResetGmma := TRUE;
        GVL.xFirstCycleRun := FALSE;
    END_IF

    GM_Gmma(SFRRESET := xSFRResetGmma);
```

Le GMMA : Le GMMA est implémenté en langage SFC. La déclaration des variables est la suivante :

```
PROGRAM GM_Gmma
    VAR_IN_OUT
        SFRReset : BOOL;
    END_VAR

    VAR
        xD1Reset : BOOL := FALSE;
        xF1A2Reset : BOOL := FALSE;
    END_VAR
```

Activité 3: Application : Etape D1 du GMMA

Question 3 Dessiner la première étape du GMMA, appelée GM_S_D1 associée à sa transition.

Question 4 Écrire l'action d'entrée `GM_Gmma.GM_AP1_ResetChartD1` qui met à `TRUE` le booléen `xD1Reset`.

Question 5 Écrire l'action active `GM_Gmma.GM_AN_D1_ArretUrgence` qui :

- Passe à `TRUE` le booléen `xD1Reset` si nécessaire.
- Appelle la POU `D1_ArretUrgence` avec le paramètre `SFCReset`
- Appelle la POU `D1_ArretUrgencePost` pour le traitement post arrêt d'urgence.

Les POU spécifiques à chaque mode sont implémentées en langage ST, LADDER ou SFC selon si le comportement associé est combinatoire ou séquentiel.

Par exemple, la POU `D1_ArretUrgence` implémenté en SFC aura les déclarations suivantes :

```
1 PROGRAM D1_ArretUrgence
2     VAR_IN_OUT
3         SFCReset : BOOL;
4     END_VAR
```