

Design UML du projet 2017/2018

Dans ce rapport, nous allons détailler comment nous avons réalisé le diagramme de classe ainsi que le diagramme de séquence avec les codes graphiques correspondant. Ainsi, toute personne lisant ce rapport sera en mesure de comprendre, compléter et réaliser de nouveaux diagrammes.

Ces diagrammes ont été réalisés dans un but de clarification du projet et de préparation à l'auto-codage et l'implantation sur la maquette. Il est important d'avoir une structure bien défini avant d'attaquer le codage pour éviter de se perdre dans la complexité de celui-ci.

Dans tout le rapport, le modèle utilisé sera celui validé pendant le projet 2017/2018 correspondant au fichier Simulink AxelSplitHEV_v7.

1. Architecture

L'implantation choisie sera sur 3 processeurs connectés par deux bus CAN A et B (voir figure ci-dessous).

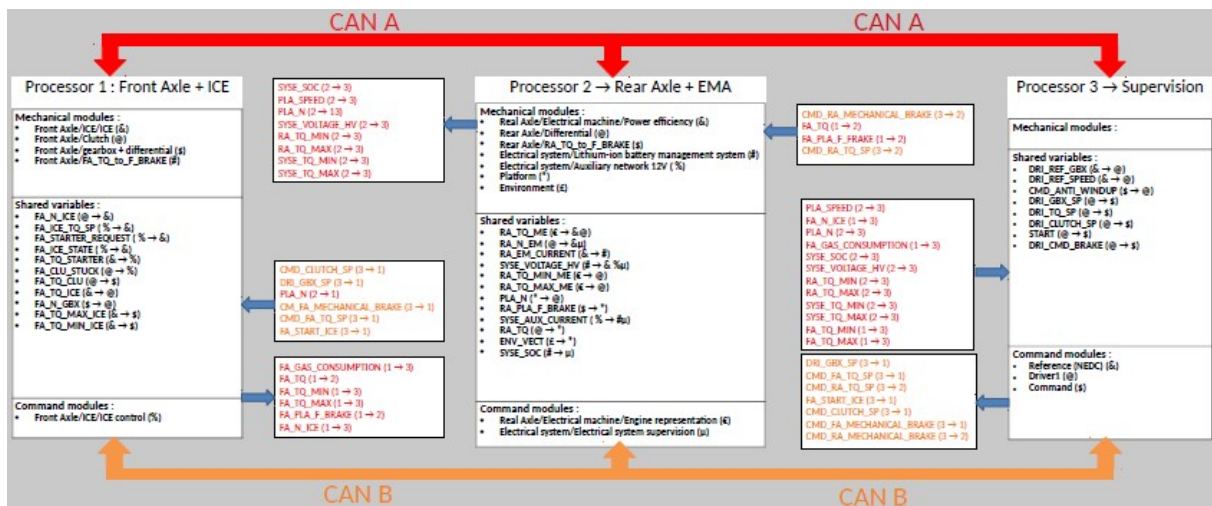


Figure 1: Structure d'implantation

Afin d'implanter le système entier sur trois processeurs, nous avons découpé le système en différents modules. Le choix de ces modules repose sur la cohérence physique qu'ils ont entre eux, leur complexité, et bien sûr la charge théorique sur chaque processeur. Nous avons donc décidé de placer l'axe avant et le moteur thermique sur un processeur, l'axe arrière, le moteur électrique et la plateforme sur un deuxième processeur, et enfin la supervision sur le troisième processeur.

Dans le futur, le module de commande de l'embrayage réalisée par l'équipe UT1 devrait être implanté sur le premier processeur.
L'architecture détaillée est présentée dans le PDF Architectures_processeurs.pdf.

Concernant le choix des variables transitant sur les bus CAN, nous avons décidé de raisonner par rapport au troisième processeur :

- Toutes les données envoyées par le premier et le second processeur transitent sur le bus CAN A, tandis que toutes celles envoyées par le troisième processeur transitent sur le bus CAN B.

Ainsi, 12 variables transitent sur le bus CAN A, et 7 variables transitent sur le bus CAN B.

Prenons donc le cas du bus CAN A pour vérifier sa charge théorique :

- Chaque donnée étant codées sur 16 bits, cela nous fait au maximum $12 \times 16 = 192$ bits à transmettre.

Le baudrate du bus CAN étant de 1 Mbits/s, et notre temps d'échantillonnage étant de 10 ms, il est capable d'envoyer 10 kbits toutes les 10 ms.
Nous avons donc largement le temps d'envoyer toutes les données dans le temps imparti, que ce soit sur le bus CAN A ou sur le bus CAN B.

2. Le logiciel

Le logiciel utilisé pour réaliser ces diagrammes est IBM Rational Software Architect Designer version 9.6. L'intégralité du projet existant est réalisée sur ce logiciel et par conséquent il vous sera nécessaire de l'utiliser pour pouvoir le modifier ou le compléter.

Pour cela, il vous suffit d'importer dans le logiciel le fichier .zip disponible dans ce dossier en suivant ces étapes :

- Commencer par créer un nouveau projet
- File -> New -> Project...
- Choisir General puis Project

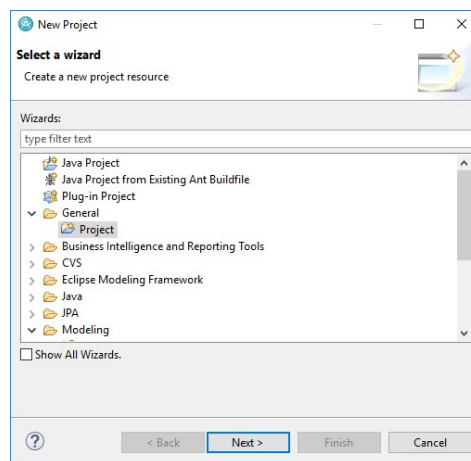


Figure 2: Création d'un projet

- Définir le nom de votre projet ainsi que votre espace de travail
- Puis File -> Import...
- Choisir General puis Archive File

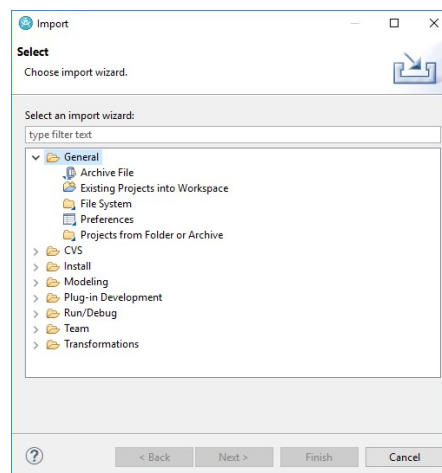


Figure 3: Sélection de l'archive 1

- Dans la barre de saisie du haut, récupérer le fichier .zip et définir votre nouveau projet comme espace de travail dans la barre de saisie du bas (attention à ce que celui-ci soit bien ouvert sinon il n'apparaîtra pas)

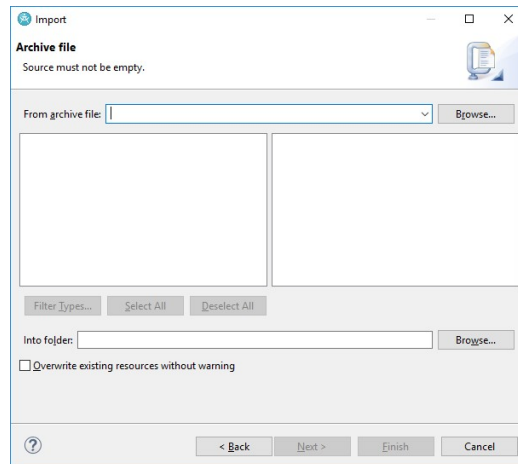


Figure 4: Sélection de l'archive 2

- Puis Finish

Vous pouvez maintenant modifier les diagrammes existant ou tout simplement vous en inspirer pour en réaliser de nouveaux. A noter que les classes et liens créés dans chaque diagramme sont réutilisables partout dans ce même projet. Par conséquent, il est utile de charger ce projet y compris si vous souhaitez en créer un nouveau pour avoir la majorité des classes prédéfinies.

- Nous avons par la suite répartie toutes ces classes sur trois processeurs comme indiqué sur la figure 1. Nous avons pour cela seulement créé des formes de couleurs qui n'ont aucune signification du point de vue du logiciel.
- Pour finir, les connections entre les différentes classes ont pu être modélisées. Pour cela, il suffit d'étudier séparément chaque attributs présent dans les classes et de vérifier où celui-ci est réutilisé. On peut finalement réaliser une interaction dirigée entre la classe en cours d'étude et la destination de la variable.
Nous avons fait le choix de modéliser seulement les interactions entre les classes de processeurs différents modélisant ainsi les échanges de variables sur les bus CAN. De plus, les variables au sein même d'un processeur seront directement accessibles par toutes les fonctions.

Cette démarche c'est avéré très efficace pour réaliser ce diagramme et nous a permis de rapidement ordonner les blocs à auto-coder.

3.2. Codes graphiques

Dans un but de clarté, de compréhension et de cohérence, nous avons imposé certaines règles graphiques :

- Les classes créées sont nommées de la même façon que les blocs Simulink qu'elles représentent.
- Les variables créées sont nommées de la même façon que les variables présentes dans Simulink.
- Les variables sont définies en *Public* (apparaissent avec un point vert sur le diagramme) si elles interagissent avec les autres processeurs et en *Protected* (apparaissent avec un point orange sur le diagramme) dans le cas contraire.
- Chaque classe possède une fonction représentant la fonction du bloc à auto-coder. Celle-ci se nomme de la manière suivante : *nom de la classe_main()*
- Un rectangle de couleur différente représente chaque processeur. Ce code couleur sera réutilisé sur tous les diagrammes
- Les interactions entre les classes sont également de la couleur du processeur dont la variable provient.
- Les interactions sont nommées de par les variables qu'elles transmettent (séparées par des points virgules dans le cas où il y en a plusieurs).

On peut retrouver ci-dessous une illustration de certaines de ces règles graphiques :

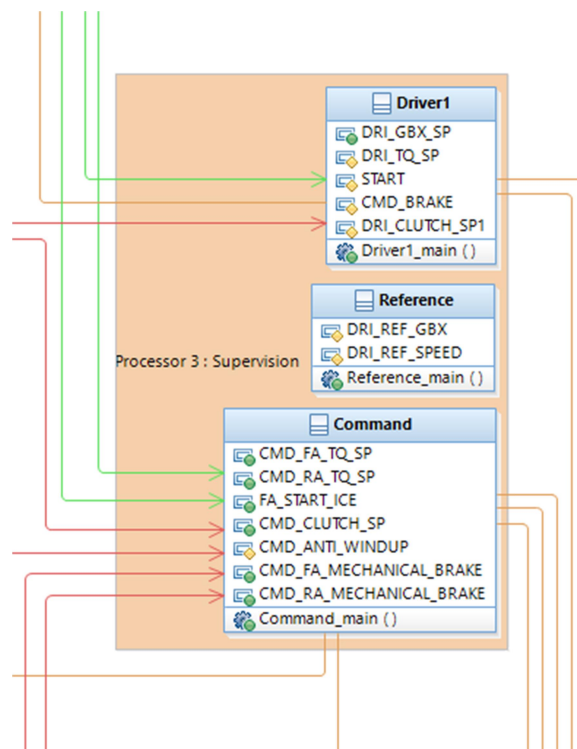


Figure 6: Illustrations des règles graphiques

Nous avons de plus rajouté un petit bloc permettant d'expliquer sur quel bus CAN les interactions s'effectueront (voir ci-après). Cela sera très utile pendant la phase de codage mais également pour la réalisation du diagramme de séquence.

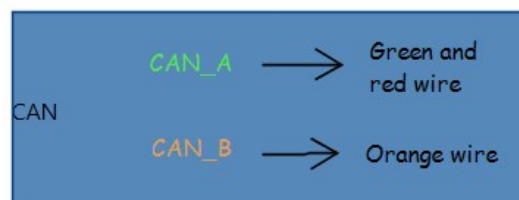


Figure 7: Répartition des CAN

4. Diagramme de séquence

Nous avons dans un second temps réalisé le diagramme de séquence préparant à l'auto-codage global de ce modèle. Ce diagramme va nous permettre de savoir dans quel ordre appeler nos fonctions mais également modéliser simplement les accès aux bus CAN. Celui-ci est présent sur la figure ci-dessous :

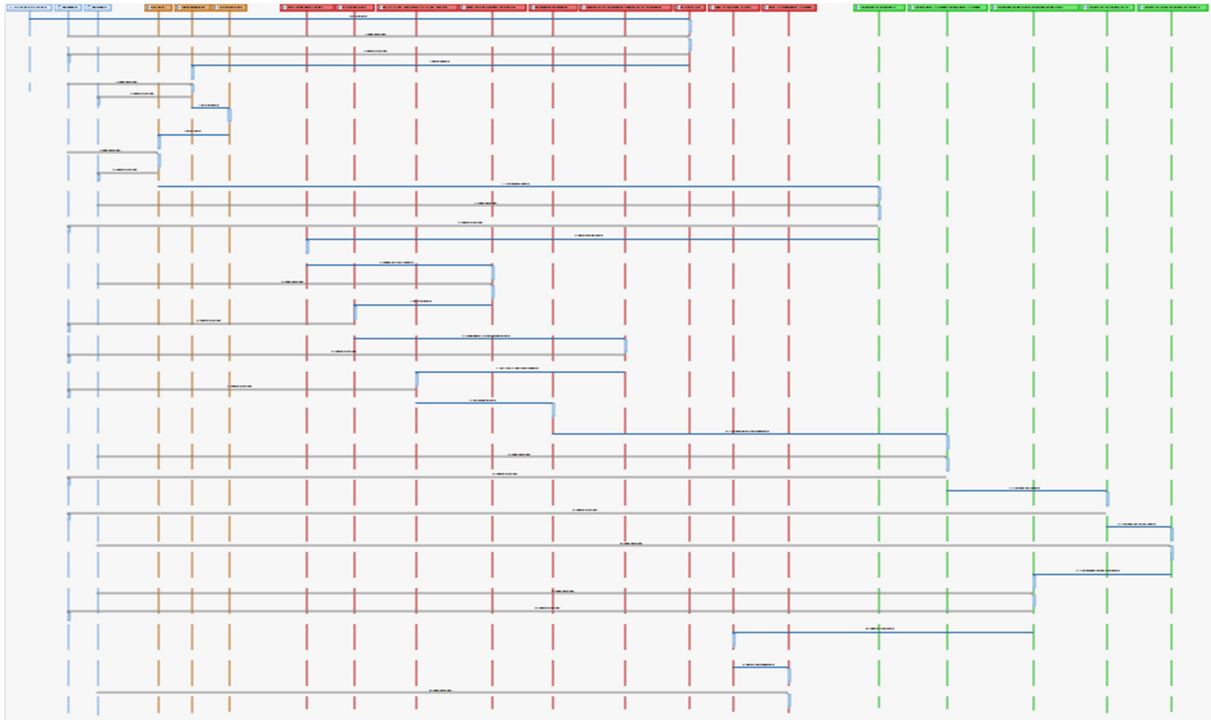






Figure 8: Diagramme de séquence global

4.1. Réalisation du diagramme de séquence

Pour la réalisation de ce diagramme de séquence, nous avons utilisé la démarche suivante :

- Il faut dans un premier temps ajouter toutes les lignes de vie (classes) sur notre diagramme. Sachant que celles-ci sont déjà créées, il suffit de les sélectionner et de les insérer dans notre diagramme.
- On rajoute un acteur modélisant le départ de notre application ainsi que des composants représentant les deux bus CAN avec lesquels nous allons interagir.
- Il nous faut par la suite déterminer l'ordre d'exécution des différentes fonctions (donc des différents blocs). Pour cela, nous nous sommes référés à l'ordre de simulation utilisé par Simulink. Cependant, cette simulation n'est au départ pas cohérente avec notre implantation car Matlab dispose tout le schéma à plat avant de l'exécuter (il ne tiens pas

compte des différents Subsystems et donc blocs que nous avons formés). Il faut donc conditionner cette simulation de la manière suivante :

-  Il faut dans un premier temps définir tous les blocs à auto-coder (soit toutes les classes) en tant que bloc atomique. Cela va permettre à Simulink de considérer ces Subsystems comme des blocs qu'il va simuler d'un seul coup. Pour cela, il faut réaliser un clic droit sur le bloc puis *Bloc Parameters (Subsystem)*. Il faut enfin cocher les cases *Treat as atomic unit* et *Minimize algebraic loop occurrences*.
 -  Cette configuration impose de faire très attention aux boucles algébriques. En effet, celles-ci vont créer des erreurs qui empêcheront la simulation de se dérouler. Dans le cas où ces boucles existent, il est nécessaire de les "casser" en ajoutant un retard sur les entrées de ces boucles.
 -  Une fois cette configuration faite, il faut sélectionner Display -> Blocks -> Sorted Execution Order.
 -  L'ordre d'apparition des classes sera indiqué en haut à droite de chaque bloc.
- Une fois que cet ordre d'exécution obtenu, il est possible de tracer des messages asynchrones entre les classes respectant cet ordre.
 - On rajoute finalement les interactions entre les bus CAN et les classes. Pour ce faire, on ajoute des interactions des bus vers les classes quand celles-ci ont besoin de valeurs présentes dans d'autre processeur (se référer au diagramme de classe). Et inversement on ajoute des interactions des classes vers le bus quand celles-ci ont des variables à partager sur les CAN (se référer également au diagramme de classe).

4.2. Codes graphiques

Dans un but de clarté, de compréhension et de cohérence, nous avons imposé certaines règles graphiques :

- Les classes créées sont nommées de la même façon que les blocs Simulink qu'elles représentent.
- Les classes sont de la couleur du processeur auquel elles appartiennent.

- Les interactions définissant l'ordre d'exécution des classes sont nommées par la fonction définie dans la classe (*nom de la classe_main()*).
- Les interactions définissant l'ordre d'exécution des classes sont de couleur bleues et plus épaisses.
- Les interactions des classes vers les bus CAN sont nommées de la manière suivante : *Public_Out_Variables*
- Les interactions des bus CAN vers les classes sont nommées de la manière suivante : *Public_In_Variables*
- Les interactions entre les bus CAN et les classes sont de couleur grises et plus fines.

On peut retrouver ci-dessous une illustration de certaines de ces règles graphiques :

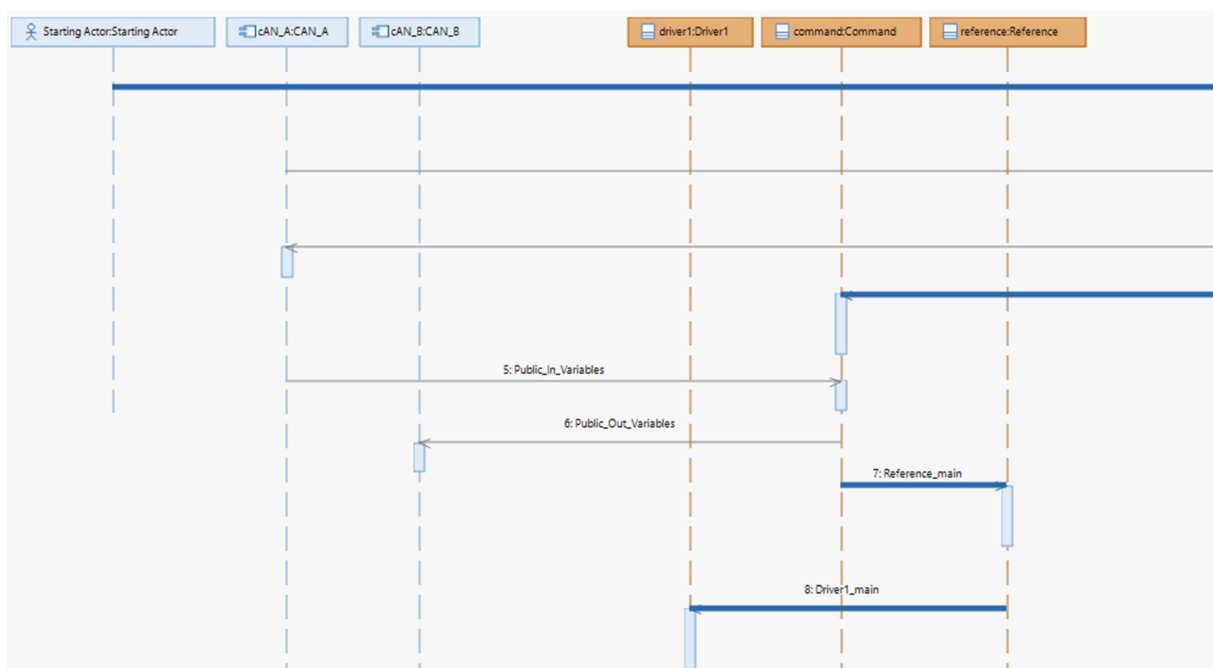


Figure 9: Illustration des codes graphiques