Université de Bretagne Occidentale

Ellidiss Technologies

Stage de fin d'étude

Modélisation et analyse de systèmes distribués temps réel

Auteur :
Cyril Muller

Tuteurs :
M. Jérome Legrand
M. Frank Singhoff

20 août 2014

Table des matières

Remerciements **4**

Introduction **5**

1 Protocoles d'accès **6**

1.1 Polling 6
1.2 Token Ring 6
1.3 TDMA 7
1.4 CSMA-CA 7

2 Bus Réseau **8**

2.1 Controller Area Network 8
2.2 FlexRay 10
2.3 Local Interconnect Network 10
2.4 ARINC 429 11
2.5 ARINC 629 12
2.5.1 Basic Protocol 13
2.5.2 Combined Protocol 15
2.6 ARINC 636 16
2.7 ARINC 825 16
2.8 Avionics Full DupleX switched Ethernet 17
2.9 MIL-STD-1553 19

3 Modélisation et techniques d'analyse **21**

3.1 Émetteurs/Récepteurs non synchronisés 21
3.2 Émetteurs/Récepteurs synchronisés 21
3.2.1 Dépendance de précédence 21
3.2.2 Dépendance avec offset/gigue 22
3.3 Modèle ADL Cheddar 23
3.3.1 CAN 24
3.3.2 ARINC 429 25
3.3.3 ARINC 629 25

3.3.4 ARINC 636 26
3.3.5 AFDX 27
3.3.6 MIL-STD-1553 28
3.4 Modèles AADL 28
3.5 Annexes Comportementales 30
3.5.1 CAN 30
3.5.2 ARINC429 31
3.5.3 ARINC629 31
3.5.4 MIL-STD-1553 31

Conclusion **32**

A Modèles AADL ARINC429 avec offset **33**
A.1 Bus library 33
A.2 Hardware 33
A.3 Software 34
A.4 Système 40

B Modèles Cheddar ARINC 429 avec offset **40**

Références **49**

Remerciements

Je tiens tout d'abord à remercier Pierre DISSAUX, Directeur de la société Ellidiss Technologies, de m'avoir accueilli comme stagiaire au sein de son établissement, ainsi que de son aide et conseils durant la période de mon stage.

Je remercie également Jérôme LEGRAND, mon maître de stage, pour son encadrement, ses conseils et méthodes de recherche.

Je tiens aussi à remercier Arnaud SCHACH qui a aussi répondu aux questions que j'ai pu avoir.

Toutes ces personnes ont contribué durant ces 5 mois à rendre mon stage enrichissant et motivant.

Introduction

Un système temps réel est un système n'étant pas seulement soumis à des contraintes de justesse des résultats, mais aussi et surtout à une contrainte de temps. En effet si le système échoue à respecter ces contraintes de temps, il provoquera une erreur système car l'information n'a pas était transmise dans le temps imparti et il ne pourra donc pas être considéré comme système temps réel.

Dans le monde de la modélisation des systèmes temps réel, on peut trouver nombre d'analyses sur chaque composants indépendamment les uns des autres. Il existe plusieurs analyses possibles pour modéliser une station reliée à un réseau temps réel de même que pour la modélisation du bus. En revanche, les analyses concernant les systèmes distribués dans leur globalité, c'est à dire de bout en bout, et non pas point par point ne sont que peu nombreuses et encore incomplètes. En effet, afin de modéliser le système entier, il faudra prendre en compte les temps de réponse des stations émettrices et réceptrices, ainsi que celui du bus reliant ce stations.

L'objectif de ce stage sera donc de trouver des pistes afin de modéliser des systèmes distribués temps réel à l'aide du langage AADL [ADL] (*Architecture Analysis and Design Langage*) au sein du logiciel AADLInspector [AIC] pour ensuite utiliser Cheddar [CDR] pour la vérification temporelle du modèle.

Ainsi dans un premier temps nous verrons les différents protocoles utilisés dans les réseaux temps réel, puis dans un second temps différents bus existant étant utilisés dans l'industrie ainsi que leurs spécificités et enfin les modélisations choisies pour chaque bus et l'analyse de ce modèle.

1 Protocoles d’accès

Dans cette section nous verrons les différents modes et protocoles d’accès à un réseau temps réel.

1.1 Polling

Dans ce type de protocole il y a une station maîtresse qui envoie séquentiellement un message aux différentes stations esclave afin de leur donner la possibilité d’envoyer leur message si elles en ont, sinon elles "écoutent" le bus à la recherche d’un message, celui peut provenir de la station maîtresse ou alors d’un autre esclave. Ce type de protocole est aussi qualifié d’architecture Maître/Esclave.

Le système est donc déterministe, il n’y aura jamais 2 messages sur le bus en même temps car seul le maître est en mesure d’initier toutes communications. De plus on peut connaître d’avance le délai maximal d’émission quand chaque station souhaite émettre.

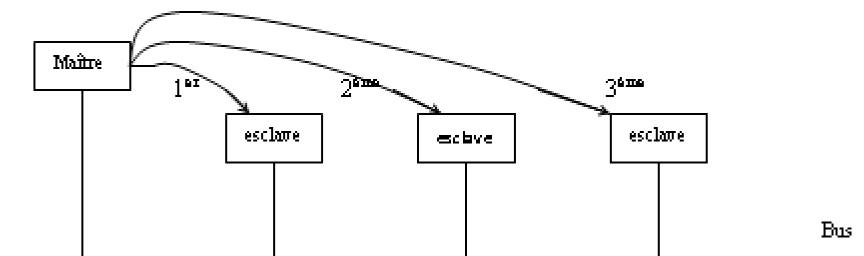


FIGURE 1 – Exemple d’architecture fonctionnant en polling.

Comme on peut le voir sur la figure 1, la station maître envoie un message à tour de rôle à chaque station esclave afin de leur donner la possibilité d’émettre.

1.2 Token Ring

Ce protocole définit une structure en anneau où chaque station est connectée à seulement 2 autres stations. Un "jeton" est passé de station en station afin de donner la possibilité d’émettre à une station, en effet seul la station en possession de ce jeton est autorisée à émettre et si une station désire émettre mais qu’elle ne possède pas ce jeton, elle doit attendre.

Lorsqu’une station a émit un message, celui est passé aux autres stations avec le jeton et ce jusqu’à atteindre son destinataire. Il faudra donc attendre que le message atteigne sa destination pour qu’un nouveau message puisse être envoyé.

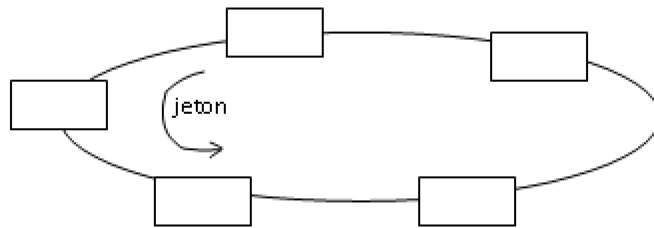


FIGURE 2 – Exemple d'architecture fonctionnant en Token Ring.

1.3 TDMA

Le protocole TDMA (Time Division Multiple Access) est un protocole à plusieurs émetteurs et chacun d'entre eux se voit attribuer un intervalle pendant lequel il est autorisé à émettre ses données par la station maître qui synchronise toutes les transmissions. Après une première phase de synchronisation, les différentes stations connectées au bus vont émettre à tour de rôle, il n'y a donc aucun risque de collision et de se retrouver avec plusieurs stations émettant au même moment. De plus on peut connaître d'avance le temps d'émission de chaque station

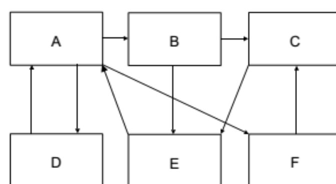
1.4 CSMA-CA

Dans protocole CSMA-CA (Carrier Sense Multiple Access with Collision Avoidance) chaque station connectées au bus se voit attribuer une priorité fixe sous la forme de bit récessif ou dominant. Lorsqu'une station souhaite émettre, elle écoute pour savoir si le bus est libre et si oui elle utilise un système d'accusés de réception réciproque entre l'émetteur et le destinataire comme montré tel que suit.

1. Le nœud envoie un bit et regarde si la réponse est différente
   – si oui et que le nœud est récessif alors il attend car un nœud dominant transmet.
   – si non, le nœud peut transmettre.
2. Le nœud émetteur en envoie le signal Ready To Send (RTS) avec notamment des informations sur la vitesse de transmission ainsi que la quantité de données transmises.
3. Le(s) nœud(s) récepteur(s) envoie(nt) le signal Clear To Send (CTS)
4. L'émetteur envoie les données
5. Le(s) nœud(s) récepteur(s) envoie(nt) l'accusé de réception ACK quand tout a été reçu.

2 Bus Réseau

Un bus réseau, ou bus de données est un câble reliant plusieurs stations ou nœuds afin de leur permettre d'échanger des données. En effet l'avantage d'un bus et qu'il permet de réduire le nombre de câbles nécessaire à l'établissement d'un réseau, car en utilisant un bus, il n'est plus nécessaire de tirer un câble allant d'une station vers chacune des stations avec qui elle souhaite communiquer (voire figure 3).



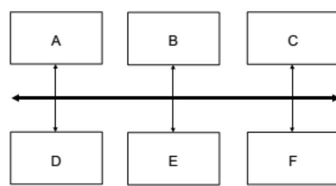


FIGURE 3 – Schéma avec et sans l'utilisation de bus.

2.1 Controller Area Network

Le bus CAN (Controller Area Network) est un bus système très répandu dans l'industrie et plus particulièrement dans l'automobile. Ce type de bus est normalisé avec la norme ISO 11898. L'accès au bus de données est basé sur le protocole **CSMA-CA** (voir section 1.4 CSMA-CA) en **half duplex**, c'est à dire que des données peuvent être envoyées dans les 2 sens du bus mais pas simultanément.

Les bus CAN sont des bus **multi émetteur** à priorités. Les collisions sont évitées en attribuant des priorités à chacun des nœuds et en utilisant le système de bits récessifs/dominants. Lorsqu'une station souhaite émettre elle commence par envoyer bit par bit son identifiant puis écoute le bus, si le bit de retour est identique alors elle continue à émettre son identifiant puis émet ses données. Par contre si le bit écouté est différent de celui envoyé alors cela signifie qu'une station plus prioritaire souhaite émettre alors la station arrête d'émettre et attend la libération du bus pour réessayer d'émettre.

Il existe 2 normes pour la couche physique du bus CAN :

– ISO 11898-3 (2006) ex ISO 11519-2 (1994) : CAN "low-speed, fault tolerant" (de 10kbits/s à 125kbits/s inclus)
– ISO 11898-2 (2003) : CAN "high-speed" (de 125kbits/s à 1Mbits/s maximum)

Chaque équipement connecté au bus est appelé un nœud et un bus CAN low-speed peut accepter jusqu'à **20 nœuds** et le bus high-speed **30 nœuds**.

Il existe également 2 standards pour la couche de liaison :

– ISO 11898 part A : CAN 2.0A "standard frame format" (identification sur 11 bits)
– ISO 11898 part B : CAN 2.0B "extended frame format" (identification sur 29 bits)

La longueur maximale d'un bus CAN dépend la vitesse de transmission choisie, ci dessous un tableau présentant la longueur maximale de bus en fonction du débit.

Vitesse (Kbits/s)	Longueur (m)	Longueur d'un bit (μs)
1000	30	1
800	50	1.25
500	100	2
250	250	4
125	500	8
62.5	1000	16
20	2500	50
10	5000	100

Table 1 – Débit max en fonction de la longueur du réseau.

Le format d'une trame CAN est tel que suit :

HEADER	DATA	FOOTER

– HEADER :
  – CAN 2.0A : 19 bits
  – CAN 2.0B : 37 bits
– DATA : jusqu'à 8 octets, soit 64 bits max
– FOOTER : 28 bits

Soit un total de 47 bits d'encapsulation pour la norme CAN 2.0A et de 65 bits pour la norme CAN 2.0B. La durée de transmission peut aussi varier à cause de l'effet de *bit stuffing* ou bits de bourrage. En effet, tous les 5 bits identiques d'affilés, un bit supplémentaire de signe contraire est rajouté. Seuls les bits de données et 34 des bits d'encapsulation (pour les 2 normes

CAN) sont concernés. Les bits de bourrage seront supprimés au moment de la réception. Donc dans le pire des cas, le nombre maximal de bits de bourrage est de :

$$N_{bourrage} = \left\lfloor \frac{34 + 8 \times n}{5} \right\rfloor \quad (1)$$

Où n est le nombre d'octet de la partie donnée. Nous avons donc une longueur maximale de trame pour CAN 2.0A de :

$$L_{max} = N_{bourrage} + 47 + 8 \times n \quad (2)$$

Et pour CAN 2.0B de :

$$L_{max} = N_{bourrage} + 65 + 8 \times n \quad (3)$$

Valeurs possible de L_{max} pour CAN 2.0A :

n	L_{max}
1	63
2	73
3	82
4	92
5	101
6	111
7	121
8	130

Valeurs possible de L_{max} pour CAN 2.0B :

n	L_{max}
1	71
2	81
3	100
4	110
5	119
6	129
7	139
8	148

TABLE 2 – Valeurs possibles de L_{max} en fonction du nombre d'octet de données.

2.2 FlexRay

FlexRay est un protocole de bus de données utilisé dans l'automobile et développé pour être plus rapide et plus fiable que les réseaux CAN. Aujourd'hui, FlexRay est un ensemble de standards ISO, de 17458-1 à 17458-5. Un réseau FlexRay possède un débit de **10 Mbits/s** en plus d'être redondant, tolérant aux fautes et déterministe. Un réseau est dit redondant lorsque les mêmes données sont envoyées simultanément sur deux bus distincts avec la même destination. Cela permet de limiter les erreurs en combinant les deux messages reçus.

2.3 Local Interconnect Network

Le bus de données série LIN (Local Interconnect Network) est principalement utilisé dans l'industrie automobile, c'est un bus plus lent que

CAN ou FlexRay mais est plus économique tout en offrant un certain niveau de fiabilité, il est parfois utilisé comme sous réseau de CAN pour les équipements de confort. Son protocole de communication s'appuie sur l'architecture **polling** (section 1.1 Polling) ou maître/esclave pouvant contenir jusqu'à 16 esclaves et il est basé sur la norme ISO 9141. Ce type d'architecture réseau fonctionne de la façon suivante : le maître initie toute communication et un seul esclave répond à ce message. Ce réseau est donc déterministe car les esclaves ne peuvent transmettre que si le maître leur en a donné l'opportunité.

Le débit maximal d'un bus LIN est de **20 Kbits/s** pour une longueur de bus de **40 mètres** avec des trames d'une taille variant entre **2, 4 et 8 octets**.

2.4 ARINC 429

La norme ARINC 429 décrit un bus de communication connu sous le nom de Digital Information Transfer System (DITS) développé par Aeronautical Radio Incorporated (ARINC) souvent appelé bus ARINC 429 par abus de langage. ARINC 429 définit à la fois une architecture, une interface électrique et un protocole pour transmettre des données numériques.

Le bus de données ARINC 429 est un bus de données série simplex (unidirectionnel) n'ayant qu'un **seul émetteur** et pouvant avoir jusqu'à **20 récepteurs** (voir figure 4). Il n'y a donc aucun besoin de synchronisation car n'ayant qu'un seul émetteur, le bus est donc déterministe et il n'y pas de risque de collision. Il existe 2 vitesses de transmission sur le bus : **12.5 Kbits/s** et **100 Kbits/s**. Toutes les trames envoyées sont d'une longueur fixe de **32 bits** dont 19 bits de données.

La norme ARINC 429 ne fonctionne pas avec un système d'adressage mais avec un système de label. En effet lorsque l'émetteur souhaite envoyer des données, il envoie *Request to Send* avec un label correspondant au type de données qu'il va envoyer, ainsi les stations réceptrices peuvent déterminer si le message leur est destiné en vérifiant le label.

L'accès au bus se fait de la façon suivante :

1. L'émetteur envoie une trame *Request to Send* avec notamment le label.
2. Le destinataire intéressé envoie *Clear to Send*.
3. L'émetteur envoie ses données après le signal *Data Follows* et ce jusqu'à un maximum de 126 trames. Puis termine la transmission avec *Final Word*.

4. Le destinataire répond alors *Data Received OK* ou *Data Received not OK*. Il peut aussi renvoyer *NACK* pour la réémission de la trame erronée.

Note : Si l'émetteur ne reçoit pas le *Clear to Send* du récepteur, il fera un maximum de 4 tentatives d'émission.

Ainsi le nombre de bits transmis lors d'une transmission ARINC 429 est :

$$L_{max} = n \times 32 \quad (4)$$

Avec n le nombre de trames envoyées

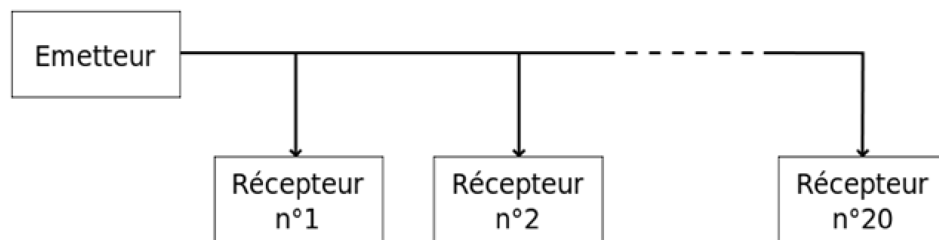


FIGURE 4 – Exemple de topographie d'ARINC 429.

2.5 ARINC 629

La norme ARINC 629 a été développée comme successeur à la norme ARINC 429, offrant plus de capacité de traitement de données ainsi que des taux de transfert plus élevés. Le standard 629 peut être utilisé à la place, ou conjointement au standard 429. Les bus de données ARINC 629 sont des bus **full duplex** (pouvant envoyer et recevoir des informations dans les 2 sens simultanément) et qui peuvent accepter jusqu'à **120 terminaux**. Ces bus sont dit **redondant**, c'est à dire qu'ils sont toujours par 2 et que les terminaux envoient et reçoivent les messages en double afin de limiter les erreurs de transmission (voir figure 5).

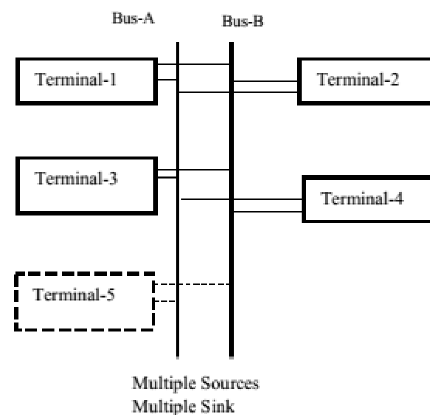


Figure 5 – Exemple de topographie d'ARINC 629.

La vitesse de transmission est de **2 Mbits/s** (l'utilisation de fibre optique peut augmenter ce débit) pour un câble pouvant atteindre **100 mètres** de longueur. Le standard supporte aussi 2 types protocole Data Link, le *Basic Protocol* et le *Combined Protocol*. Ces 2 protocoles ne peuvent coexister sur le même bus dus aux différences fondamentales les séparant. En effet même s'ils gèrent les messages périodiques de la même façon : ils suivent des tables de messages et les paramètres de contrôle des transmissions sont identiques, ils diffèrent dans la façon de gérer les messages sporadiques.

Ce protocole est similaire au protocole **TDMA** dans le sens où les différentes stations se voient attribuer un intervalle de temps afin de transmettre leurs données et le respect de ces contraintes temporelles est assuré par plusieurs timers synchronisés par la station *leader*.

Par la suite on parlera de **cycle mineur** qui correspond au temps de transmission maximal alloué à une station pour envoyer ses messages apériodiques et périodiques et de **cycle majeur** l'hyper période des transmissions de toutes les stations

L'ordre dans lequel les stations sont autorisées à émettre est déterminé au moment de l'élection du *leader* suivant la séquence d'initialisation et la dérive d'horloge des stations. Cette ordre reste le même pour chaque cycle mineur.

Dans ce protocole, les terminaux sont toujours en train d'écouter le bus et notamment les événements, BA (Bus Active) pour le bus actif, BQ (Bus Quiet) quand le bus est silencieux et BC (Bus Clash) en cas de collision.

2.5.1 Basic Protocol

Un bus ARINC 629 BP (Basic Protocol) se compose de 2 sous-modes, le mode périodique où les temps de transmission sont fixés pour chaque cycle mineur et le mode apériodique où ces temps peuvent varier d'un cycle mineur à un autre. Afin d'éviter les collisions, le bus utilise plusieurs timers tels que suit.

- *Transmit Interval* (TI) : le même pour tous les terminaux (0.5 à 64 ms) et est aussi le plus long des timers, égale au cycle mineur minimum. Il démarre au moment où un terminal commence à émettre.
- *Synchronisation Gap* (SG) : le même pour tous les terminaux et permet de s'assurer que tous les terminaux obtiennent un accès au bus et est supérieur à TG. Il démarre à chaque fois que BG est détecté et peut

être réinitialisé avant son expiration si BA est détecté. Il est redémarré quand le terminal envoie de nouveau.
– *Terminal Gap* (TG) : timer qui permet de gérer la compétition d'accès au bus entre les terminaux.

La valeur fixe du cycle mineur est égal à la valeur du timer TI et peut être garantie à condition que la relation suivante soit vraie.

$$TI \geq SG + \sum_{i=1}^{n} TG_i + Tx_i \tag{5}$$

Tx_i est le temps de transmission de la station i. Celui ci peut varier d'un cycle mineur à un autre. On peut donc définir TX_i le temps de transmission pire cas et ainsi transformer l'inégalité précédente 5 en égalité et donc avoir une valeur fixe pour TI. La figure 6 montre un exemple de cycle mineur en mode périodique.

En revanche, si jamais la condition suivante est vrai, alors le *Basic Protocol* opérera en mode apériodique.

$$TI < SG + \sum_{i=1}^{n} TG_i + Tx_i \tag{6}$$

On peut noter que les deux équations 5 et 6 sont mutuellement exclusives.

De plus la synchronisation au niveau du cycle mineur est assurée par le timer SG et non plus par TI comme on peut le voir dans l'inégalité 6, TI sera terminé avant que SG ne soit actif. Cette modification résulte donc bien en une longueur variable de cycle mineur.

Il est donc plus difficile de garantir le comportement du mode apériodique, mais cela peut être possible en anticipant le comportement pire cas des transmissions sporadiques de calculé TI en fonction de celui ci.

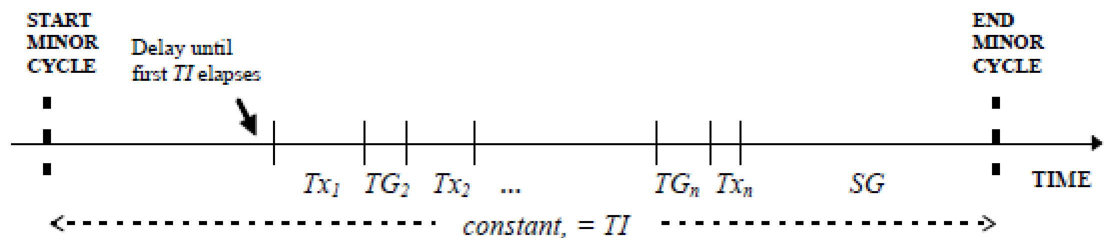


FIGURE 6 – Cycle mineur en mode périodique.

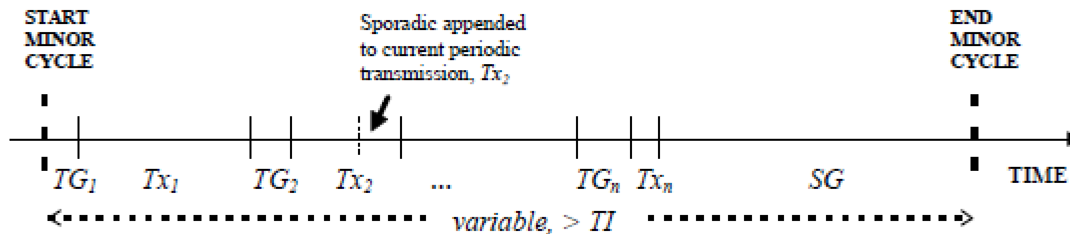


FIGURE 7 – Cycle mineur en mode apériodique.

2.5.2 Combined Protocol

Le protocole ARINC 629 CP a été développé pour compenser certaines faiblesses du BP, notamment pour avoir une approche plus efficace dans gestion combinée des tâches périodiques et sporadiques.

Les transmissions périodiques (Niveau 1) sont servies dans un ordre fixe sans préemption comme dans le BP, en revanche la durée du cycle mineur ne varie jamais.

Les transmissions sporadiques seront servies pendant la période de temps allouée. On distingue 2 niveaux de priorité de messages sporadiques :

- Niveau 2 : messages sporadiques courts et fréquents (plus prioritaires).
- Niveau 3 : messages sporadiques long et moins fréquents (moins prioritaires).

Comme pour le BP, on retrouve les timers TG et TI, en revanche ce dernier n'est appliqué qu'à la première transmission périodique de chaque cycle mineur, c'est à dire seulement pour la station leader. Il est ensuite remplacé par l'événement *Concatenation Event* (CE). De plus, le CP utilise d'autres timers dont voici la liste :

- *Aperiodic Synchronization Gap* (ASG) : utilisé au sein du cycle mineur pour la synchronisation du passage des messages de niveau 1 au niveau 2.
- *Periodic Synchronization Gap* (PSG) : utilisé pour synchronisation au niveau du cycle mineur, c'est à dire en fin de cycle.
- *Aperiodic access Time-out* (AT) : indique le temps de début du prochain cycle mineur afin qu'il n'y pas de transmission sporadique qui prendrait trop de temps et empiéterais sur le cycle mineur suivant.

Durant chaque cycle mineur, chaque station est limitée à une transmission obligatoire de niveau 1 et une transmission facultative de niveau 2. Les

transmissions de niveau 3 ne sont pas limitées si se n'est pas *AT* et sont aussi facultatives.

2.6 ARINC 636

La norme ARINC 636 est adaptée du protocole FDDI (Fiber Distributed Data Interface) ou **Token Ring** définie pour des réseaux à base de fibre optique. Le protocole du Token Ring fonctionne de la façon suivante : les stations se passent un jeton de voisin en voisin et seul le terminal en possession du jeton est autorisé à émettre sur le réseau. C'est donc une architecture en anneau pouvant accueillir jusqu'à **500 stations**, c'est à dire que chaque nœud du système n'est connecté qu'à 2 et seulement 2 autres nœuds en double. C'est à dire qu'il existe un anneau primaire par lequel les données sont envoyées par défaut, mais aussi un anneau secondaire dont les données voyagent dans le sens inverse de l'anneau primaire, afin de pallier à une éventuelle rupture de l'anneau primaire (voir figure 8).

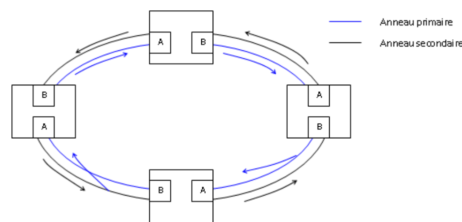


FIGURE 8 – Exemple d'architecture FDDI.

On peut voir que chaque nœud possède 2 ports de communication A et B, le port A correspond à l'entrée du réseau primaire et la sortie du réseau secondaire et le port B à la sortie du réseau primaire et l'entrée du réseau secondaire. Donc lorsqu'une station émet un message, celui ci est passé en même temps que le jeton de voisin en voisin jusqu'à atteindre son destinataire.

De plus du fait de l'utilisation de la fibre optique, le réseau est insensible aux perturbations électromagnétiques et offre un débit de **100 Mbits/s**.

2.7 ARINC 825

Ce protocole ARINC est l'adaptation du réseau pour l'avionique basé sur la version **CAN 2.0B**, c'est à dire des trames avec une identification sur 29 bits. Voir donc section 2.1 Controller Area Network, pour les spécifications.

2.8 Avionics Full DupleX switched Ethernet

Le bus de données AFDX (Avionics Full DupleX switched Ethernet) est un bus Ethernet redondant et fiabilisé qui a été développé pour équiper l'Airbus A380. Ce type de bus est donc un bus Ethernet auquel un certain nombre de modifications ont été appliquées afin de créer une nouvelle norme pour l'avionique, il est donc **full duplex** avec un débit de **100 Mbits/s**. En effet l'AFDX est normalisé par la partie de 7 de la norme ARINC 664, lui même adapté de la norme IEEE 802.3 décrivant des réseaux Ethernet commutés. Le réseau AFDX est dit **réseau commuté** car les équipements terminaux chargés de l'émission et de la réception des données s'organisent autour de commutateurs qui s'occupent d'acheminer le message d'un terminal à un autre.

La transmission de données sur un réseau AFDX passe par la réservation d'un **lien virtuel** ou virtual link en anglais. Ces liens virtuels se partagent les 100 Mbits/s de bande passante disponible comme illustré sur la figure 9.

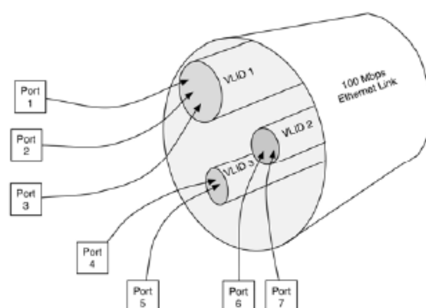


FIGURE 9 – Trois liens virtuels dans un même lien physique.

Chaque lien virtuel se voit attribuer 2 paramètres :

- Bandwidth Allocation Gap (BAG) intervalle minimal entre la transmission de 2 trames sur le même lien virtuel
- L_{max}, taille maximale d'une trame pouvant être transmise sur le lien virtuel

La formule pour calculer la bande passante disponible pour un lien virtuel est donc :

$$BP_{max} = L_{max} \times 8 \times BAG \quad (7)$$

Le tableau ci dessous représente les différentes valeurs possibles de BAG.

BAG (ms)	Fréquence (Hz)
1	1000
2	500
4	250
8	125
16	62.5
32	31.25
64	15.625
128	7.8125

Table 3 – Valeur possible pour le BAG.

Par exemple si VLID1 a un BAG de 32 millisecondes, alors les trames Ethernet sont envoyées une toutes les 32 millisecondes sur VLID1. Si VLID1 a un L_{max} égale à 200 octets, alors la bande passante maximale BP_{max} sur VLID1 est de 50 000 octets par seconde (200*8/(32/1000)).

L'utilisation de liens virtuels permet de garantir les temps de transmission ainsi que le déterministe du réseau même si en pratique cela résulte par un réseau Ethernet sous exploité.

Comme nous l'avons vu précédemment dans cette section, le réseau AFDX est redondant et cette redondance est gérée par des commutateurs. Donc il y a 2 réseaux distincts sur lesquels une même trame est envoyée en double comme on peut le voir sur la figure 10. La redondance des données est ensuite traitée par le commutateur afin d'éliminer les trames incomplètes ou corriger les erreurs ce qui permet d'assurer une transmission du message plus fiable.

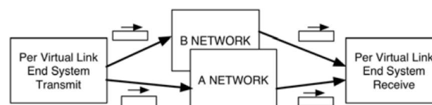


Figure 10 – Redondance du réseau AFDX.

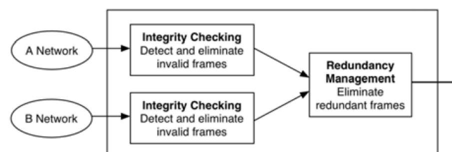


Figure 11 – Gestion de la redondance du réseau AFDX.

2.9 MIL-STD-1553

La norme MIL-STD-1553 décrit un bus de communication largement utilisé en avionique militaire, aussi adopté par l'OTAN sous la notation STANAG 3838. Le bus de données est un **bus half duplex redondant** avec une vitesse de transmission de **1 Mbits/s**. Le protocole d'accès au bus est le **polling**, c'est à dire qu'une station maître appelée *Contrôleur de Bus* donne la possibilité d'émettre aux stations esclaves dont le nombre est **limité à 31** (limitation liée à l'adressage). Une station esclave ne peut émettre que si la station maîtresse lui a donné la "parole".

Il existe en tout 3 type d'équipements connectés à un bus MIL-STD-1553 :

- *Contrôleur de Bus* ou Bus Controller : terminal qui orchestre toutes les transmissions sur le bus (1 seul).
- *Terminaux Abonnés* ou Remote Terminals : utilisent le bus pour communiquer (jusqu'à 31).
- *Moniteur de Bus* ou Bus Monitor : "écoute" les données envoyées sur le bus (≥ 0).

Le contrôleur et les terminaux communiquent avec 3 types de mots différents faisant tous une taille de **16 bits**.

- *Mots de Commande* ou Command Words : mots transmis par le contrôleur, composé des bits suivants :
  - bits 1 à 5 : adresse du terminal cible.
  - bit 6 : à 0 pour la réception et à 1 pour l'émission.
  - bit 7 à 11 : adresse des données (**non modélisé**).
  - bits 12 à 16 : nombre de mots de données.
- *Mots de Données* ou Data Words : données transmises soit par le contrôleur soit par un terminal.
- *Mots de Statut* ou Status Words : réponse d'un terminal au contrôleur, composé des bits suivants :
  - bits 1 à 5 : adresse du terminal cible.
  - bits 6 à 16 : bits de contrôle (**non modélisé**).

Entre 2 messages, il doit y avoir un temps d'attente (inter-message gap) ayant comme valeur minimale 4 μs et pouvant aller jusqu'à 1 ms pour les modèles plus anciens. Généralement la valeur du inter-message gap est supérieur à 4 μs.

De plus les terminaux doivent répondre au contrôleur entre 4 et 12 μs, sinon, si au bout de 14 μs, le terminal n'a pas répondu, le contrôleur considérera que le message ou la commande n'a pas été reçu.

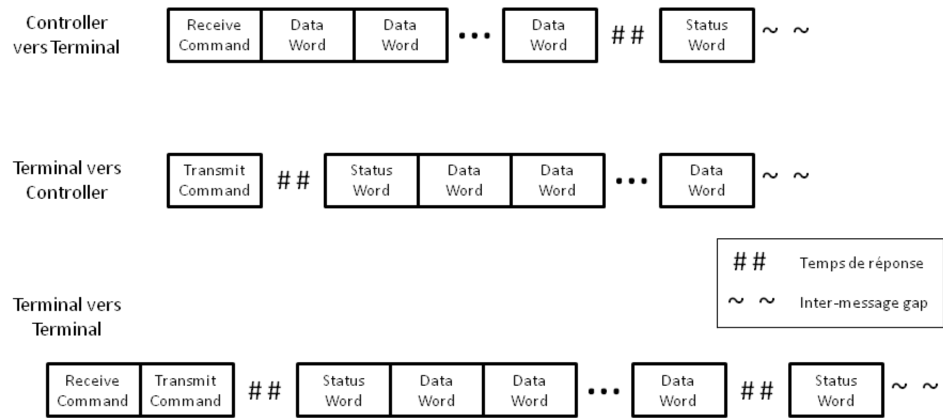


FIGURE 12 – Exemple de transfert de données entre le *Controller* et les *Terminals*.

3 Modélisation et techniques d'analyse

Ici nous verrons 3 types d'analyses utilisant la théorie de l'ordonnancement temps réel. Les bus seront modélisés par des processeurs qui représenteront l'occupation du bus et l'émission des messages. Ces messages seront envoyés sur le bus par des tâches s'exécutant sur des systèmes différents (aussi modélisés par des processeurs).

Dans un premier temps nous verrons un cas où les tâches s'exécutent indépendamment les unes des autres, c'est à dire que les émetteurs et les récepteurs ne sont pas synchronisés et les messages seront stockés dans des buffers (comme dans le cas de l'utilisation un driver).

Dans un deuxième temps, nous verrons 2 cas où les émetteurs et les récepteurs sont synchronisés d'abord avec des tâches ayant des dépendances de précédence, c'est à dire que les tâches réceptrice seront activées sur réception de message et enfin un deuxième cas en utilisant des offsets/jitter afin de modéliser la précédence en imposant un délai sur les tâches réceptrices.

Nous utiliserons les termes suivants :
- T_i : la période d'une tâche i.
- C_i : le temps d'exécution pire cas d'une tâche i.
- J_i : le temps de gigue (ou *jitter* en anglais) d'une tâche i.
- R_i : le temps de réponse pire cas d'un tâche i.

3.1 Émetteurs/Récepteurs non synchronisés

Ici les tâches s'exécutent indépendamment les unes des autres, les messages envoyés sont stockés dans des buffers (ou modélisé par une ressource partagée/sémaphore) pour être lus quand la tâche réceptrice est prête.

On peut aussi considérer que la réception est gérée par une tâche spécifique effectuant un polling sur le buffer de réception afin d'activer les tâches réceptrices concernées par le message.

3.2 Émetteurs/Récepteurs synchronisés

3.2.1 Dépendance de précédence

Nous utiliserons les dépendances de précédence et donc cela signifie que les tâches réceptrices seront activées sur réception de message. Nous

aurons donc des précédences telles qu'un message ne peut apparaître sur le bus tant que la tâche émettrice ne s'est pas terminée (envoyée le message sur le bus), de même que la tâche réceptrice ne peut s'exécuter tant qu'il n'y a pas de message sur le bus (activation sur arrivé de message).

L'utilisation de précédences permet à Cheddar de calculer les temps de jitter automatiquement à l'aide des outils mis à disposition (tools -> end-to-end response time), c'est à dire l'algorithme de Tindell. Les précédences sont donc utilisées pour la simulation et les jitters pour les tests de faisabilité.

En revanche les temps de réponse de bout en bout obtenu avec l'algorithme de Tindell sont des temps très pire cas.

3.2.2 Dépendance avec offset/gigue

Dans cette partie nous appliquerons directement la méthode de calcul de temps de réponse de bout en bout de Tindell afin de calculer les jitter que nous injecterons aux tâches comme un offset.

Le temps de réponse d'une tâche est donné avec l'équation de récurrence suivante :

$$r_i^{n+1} = J_i + C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i^n + J_j}{T_j} \right\rceil C_j \quad (8)$$

On peut choisir comme valeur de départ $r_i^0 = C_i$.

Cette équation 8 de récurrence convergera vers $r_i^{n+1} = r_i^n$ si la charge du processeur est $\leq 100\%$.

La valeur de J_i correspond au temps de réponse de la tâche précédente, comme celui de la tâche émettrice par exemple. Cette équation permet donc de calculer un temps de réponse global à la transmission d'un message de bout en bout.

Le délai de transmission d'un message dépend du temps d'accès à la couche MAC et peut être calculé avec l'équation suivante :

$$R_m = J_m + M_m \quad (9)$$

Où J_m est la gigue du message (comme le temps de réponse de la tâche émettrice par exemple) et M_m le délai de communication.

3.3 Modèle ADL Cheddar

Comme précisé précédemment, on représentera les bus par des processeurs et les messages se déplaçant sur ce bus par des tâches s'exécutant sur le processeur. Les tâches seront ordonnancées selon des priorités fixes POSIX. Les messages sur le bus et les tâches réceptrices hériteront aussi de cette même priorité.

Cette représentation nous permettra d'utiliser Cheddar afin de vérifier la faisabilité temporelle des modèles construits.

Représentation d'un message sous la forme d'une tâche :

Message	**Tâche**
Temps de transmission	Temps d'exécution (capacité C)
Délai critique	Deadline
Priorité du message	Priorité de la tâche (héritée de la tâche émettrice)

TABLE 4 – Équivalence tâche/message.

Le fait qu'un bus soit half duplex ou full duplex sera modélisé par un second processeur afin de pouvoir transmettre deux messages simultanément.

Modèle Cheddar	CAN	ARINC 429	ARINC 629	ARINC 636	AFDX
Bus	1 processeur mono core	1 processeur mono core	2 processeurs mono core	1 processeur mono core par lien entre 2 nœuds	2 processeurs mono core par lien virtuel

TABLE 5 – Choix de processeur pour modéliser les différents bus.

Nous utiliserons 2 processeurs P1 et P2 qui communiqueront par l'intermédiaire d'un bus qui sera lui aussi modélisé par un ou plusieurs processeurs (voir figure 13. Nous assignerons le jeu de tâches suivant à chacun des processeurs. Certaines tâches sur P1, t1_p1, t2_p1 et t3_p1 enverront des messages aux tâches t1_p2, t2_p2 et t3_p2 sur le processeur P2.

Les tâches t1_bus, t2_bus et t3_bus s'exécuteront sur le processeur BUS modélisant le bus de données et permettront de simuler l'occupation du bus et la transmission de messages.

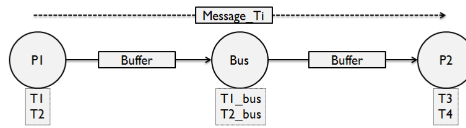


FIGURE 13 – Exemple de topologie de système distribué utilisé comme exemple.

Dans le tableau 6, on peut voir les paramètres des tâches qui seront utilisées afin de modéliser les communications entre les processeurs pour les bus CAN, ARINC429 et ARINC629. Pour les autres bus nous choisirons une architecture avec 3 processeurs dont nous détaillerons les jeux de tâches dans les sections suivantes.

Tâches	Capacité	Période	Priorité	Processeur
t1_p1	1	5	5	P1
t2_p1	2	10	4	P1
t3_p1	1	15	3	P1
t4_p1	4	30	2	P1
t5_p1	2	30	1	P1
t1_bus	1	5	5	BUS
t2_bus	2	10	4	BUS
t3_bus	1	15	3	BUS
t1_p2	1	5	5	P2
t2_p2	2	10	4	P2
t3_p2	1	15	3	P2
t4_p2	3	15	2	P2

TABLE 6 – Les différentes tâches assignés aux processeurs utilisées pour l'exemple des bus CAN, ARINC429, ARINC629.

Des précédences de dépendance devront être mises en place entre la tâche émettrice et la tâche modélisant le message sur le bus car le message ne peut être présent sur le bus tant qu'il n'a pas était envoyé.

Voir annexes B pour un exemple de modèle Cheddar ADL du bus arinc429 utilisant des offsets.

3.3.1 CAN

Les bus CAN utilise un système de synchronisation inhérent à son protocole, la modélisation d'une telle synchronisation ne peut être réalisée avec les choix que nous avons faits précédemment. Les temps

induits peuvent être calculés de façon statique puis ajoutés au temps de communication global, par exemple.

Les priorités des stations ne pourront pas être modélisées avec Cheddar, en revanche, il est possible de calculé une priorité aux tâches s'exécutant sur le processeur BUS afin de représenter les stations plus prioritaires. En effet pour calculer une telle priorité, on assignera manuellement une priorité pour chaque station à laquelle on pondérera la valeur de la priorité de la tâche émettrice. Ainsi, on se retrouvera avec un jeu de tâche représentant les messages émis avec des priorités différentes représentant l'ordre d'accès.

3.3.2 ARINC 429

Dans un deuxième exemple nous aurons 2 systèmes (processeurs) communiquant par un bus ARINC 429, le processeur P1 sera l'émetteur et P2 et P3 seront des récepteurs. Le processeur P1 enverra le signal *Request to Send* à toutes les stations. Ce signal contient aussi un label correspondant au type d'informations que P1 souhaite envoyer (déterminer par l'utilisateur et n'apparaissant pas dans Cheddar). Les processeurs P2 et P3 lisent chacun ce signal, on considérera donc que P2 reconnaît le label et envoi ensuite le signal *Clear to Send*. Les labels des messages étant unique pour chaque station, P3 ne sera donc pas concerné par les données qui seront envoyées par la suite.

Une fois que P1 a reçu le signal *Clear to Send* venant de P2, il envoie alors ses données en les précédant du signal *Data Follows* et les termine avec *Final Word*. Une fois que P2 a reçu toutes les données correctement, il envoie le signal *Data Received Ok*.

3.3.3 ARINC 629

Dans le cas d'un bus ARINC 629, nous ne nous intéresserons qu'à la norme ARINC 629 *Combined Protocol*, car le *Basic Protocol* possède un cycle mineur dont la période est variable en cas de mode apériodique et donc son comportement ne peut être prédit et modélisé à l'aide d'outils d'analyse temps réel.

ARINC 629 *CP* est basé sur la norme TDMA et donc une station maîtresse ordonnance l'ordre d'émission des autres stations. Cet ordre sera déterminé par avance lors de la construction du modèle (notamment à travers l'affectation des priorités des tâches s'exécutant sur le processeur *BUS*) afin de faire coïncider les tâches émettrices avec l'ordre d'émission.

Nous pouvons aussi considérer une autre modélisation en représentant non plus les stations émettrices par des processeurs mais par des tâches. Ces tâches s'exécuteront sur un même processeur et leur capacité correspondra aux valeurs des timers ARINC 629 (à déterminer lors de la construction du modèle). Lorsqu'une station n'est pas en train d'émettre, elle est en mode réception afin de traiter les messages entrant. La réception ne pourra donc pas être modélisée avec ce modèles.

Malgré le fait qu'un bus ARINC 629 soit full-duplex, de part son protocole, on peut affirmer qu'il y aura toujours au plus qu'un seul message sur le bus à un moment t. On ne choisira donc que de modéliser la redondance du bus.

3.3.4 ARINC 636

Pour ce protocole avec une structure en anneaux, nous aurons 3 nœuds : P1, P2 et P3, chacun étant relié à exactement 2 autres nœuds comme on peut le voir sur les schémas 14 et 15. L'anneau secondaire B n'étant présent que pour pallier à un échec du réseau primaire A, nous ne le modéliserons pas dans nos exemples afin de ne pas surcharger les résultats avec des liens inactifs.

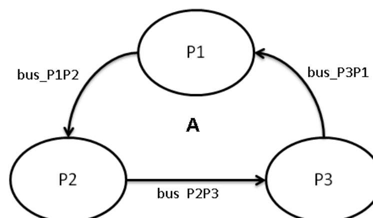


FIGURE 14 – Anneau primaire du réseau.

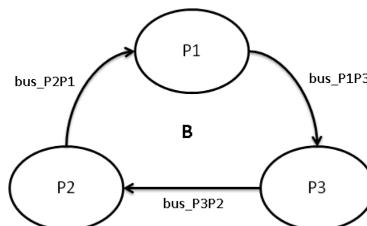


FIGURE 15 – Anneau secondaire du réseau.

Du fait de l'architecture différente de ce réseau, le jeu de tâches a dû être modifié afin de mieux représenter un réseau ARINC 636.

Tâche	C	T	P
T1_P1	3	10	3
T2_P1	2	15	2
T3_P1	7	30	1
T1_BUS_P1P2	1	10	3
T1_P2	1	10	3
T1_P2	5	20	1
T1_BUS_P2P3	1	10	3
T1_P3	2	10	3
T2_P3	8	30	1

Table 7 – Jeu de tâches pour ARINC 636.

Pour ce bus on ne fera pas de modèle avec des tâches émettrice/réceptrice non synchronisées et celui de la synchronisation par offset car la transmission du message se fait à travers le passage d'un "jeton" que sera représenter par les précédences de Cheddar. Chaque nœud du réseau est donc en attente du jeton et donc à fortiori du message. Seul la station possédant ce jeton est autorisée à émettre soit le message qu'elle vient de recevoir mais qui est destiné à une autre station, soit son propre message.

3.3.5 AFDX

Comme nous l'avons vu précédemment, la transmission de message dans un réseau AFDX passe par la réservation de *liens virtuel*. Ces liens virtuels étant limités par des contraintes connus, nous pouvons donc savoir combien de message seront envoyés par secondes sur chacun de ceci.

Nous choisirons donc de représenter le bus comme plusieurs liens virtuels que nous modéliserons chacun par un processeur. Si par exemple un bus AFDX possède 3 liens virtuels, nous le représenterons donc par 3 processeurs (il est aussi possible de le représenter avec un processeur multi-cœurs). Nous désignerons chacun des liens virtuels par VLIDx, pour Virtual Link Id et x le numéro de ce lien.

Tâche	C	T	P
T1_P1	3	10	3
T2_P1	2	15	2
T3_P1	4	30	1
T1_VLID1	1	10	3
T1_VLID2	1	15	2
T1_P2	2	10	3
T2_P2	1	5	1
T1_P3	2	15	2
T2_P3	3	30	1

TABLE 8 – Jeu de tâches pour AFDX.

3.3.6 MIL-STD-1553

Nous modéliserons 3 processeurs : *controller*, *slave1* et *slave2* afin de représenter 3 stations communicantes sur le réseau. Les tâches s'exécutant sur *controller* initierons les communications en envoyant les signaux correspondants (*Transmit Command* et *Receive Command*).

Dans un premier temps, *controller* initiera une communication dans le sens *controller* $\longrightarrow$ *slave1* avec la tâche *t1_controller*, puis dans un second temps *slave1* $\longrightarrow$ *slave2* avec *t2_controller* et enfin pour terminer *slave2* $\longrightarrow$ *controller* avec *t3_controller*.

Les tâches s'exécutant sur les processeurs *slave1* et *slave2*, envoie au *controller* la confirmation de la réception, c'est à dire le *status word* correspondant.

3.4 Modèles AADL

Bus	CAN	ARINC 429	ARINC 629	ARINC 636	AFDX
Transmission Time (par octet)	8 μs à 1 Mbits/s 64 μ s à 125 Kbits/s	80 μs à 100 Kbits/s 640 μ s à 12.5 Kbits	4 μs	80 ns	dépend du BAG et L_{max}

TABLE 9 – Temps de transmission pour chaque bus.

Les temps de transmission de chaque bus sera définit avec l'attribut **Transmission_Time** qui se compose d'un temps fixe t_{fixe} représentant le temps d'accès à la couche réseau et du temps de transmission d'un octet t_{octet}, soit du temps de propagation. Le temps de transmission d'une trame est donc calculé tel que suit :

$$t_{transmission} = t_{fixe} + n \times t_{octet} \tag{10}$$

Où n est le nombre d'octets composant la trame.

La détermination de la capacité des tâches s'exécutant sur le processeur modélisant le bus utilisera l'attribut *Transmission_Time* et *Source_Data_Size* qui représente la quantité d'informations sortantes et que l'on spécifie donc pour un port sortant.

Un exemple complet utilisant un bus ARINC 429 se trouve en annexe A.

Afin de modéliser le protocole réseau, c'est à dire si il y a une synchronisation entre l'émetteur et le récepteur, on utilisera la propriété du fichier *bus_properties.aadl Channel_Type*, c'est à dire, spécifier si le bus est half duplex ou full duplex. On peut considérer qu'un bus étant half duplex aura une synchronisation émetteur/récepteur et qu'un bus full duplex n'en aura pas.

Il y a aussi le problème de l'ordre d'accès au bus, car on ne peut simplement utiliser un ordonnanceur pour les messages car ils sont gérés en fonction de leur ordre d'arrivé et parfois d'autres paramètres spécifiques au protocole.

Par exemple pour les bus CAN, chaque station possède une priorité définie par l'utilisateur, ces priorités pourraient être ajoutées dans les propriétés d'un processeur afin de pondérer la priorité du message et ainsi l'ordonnancement par priorité peut être utilisé pour modéliser un bus CAN.

Pour d'autres bus, il faut donc considérer d'autres méthodes. Dans le cas d'un bus ARINC429 ou MIL-STD-1553, une station gère les accès au bus avec le protocole *Polling*, donc la gestion des messages est gérée au niveau du software et du point de vue du bus nous sommes sûr qu'il n'y aura au plus qu'un seul message en attente du bus à n'importe quel moment.

Avec le bus ARINC636, la topologie du réseau assure qu'un seul message transite sur le bus, de même que chaque station doit être en possession du "jeton" pour émettre dû au protocole Token Ring.

Enfin pour ARINC629, l'ordre d'émission des stations est ordonnancé au moment de l'initialisation et l'utilisation de timers permet d'assurer le déterminisme du bus.

Ainsi, seul le bus CAN nécessitera l'ajout d'une propriété pour la priorité des stations, en effet avec les outils déjà à disposition avec AADL, on peut modéliser les autres bus.

3.5 Annexes Comportementales

Avec les annexes comportementales, on peut spécifier plus précisément le comportement d'un bus de données, comme par exemple les différents protocoles de communication.

3.5.1 CAN

Les stations émettront bit par bit leur identifiant qui est aussi utilisé pour établir les priorités entre les stations. Ces identifiants peuvent être d'une taille de 11 ou 29 bis dépendant de la version CAN choisie.

Nous détaillerons aussi plus précisément les trames CAN afin de créer un modèle de données AADL (*data*) qui sera transmit de station en station.

- identifiant : 11 ou 29 bits suivi d'un bit dominant (bit RTR).
- taille de transfert : nombre d'octets de la partie données qui est codé sur 4 bits et ayant une valeur pouvant varier de 0 à 8.
- données : de 0 à 8 octets de données

Les stations étant toutes en écoute de bus par défaut, l'accès au bus commencera par la transmission des bits d'identification et la station ayant pu émettre son identifiant en entier sera celle qui obtiendra l'accès au bus et qui transmettra le reste de sa trame. Les autres repasseront en écoute de bus.

Il faudra aussi ajouter un Process pour le contrôle du bus : *controller*. Ce process servira de multiplexeur, il enverra les données reçues à toutes les stations auxquelles il est relié car la station qui vient d'émettre à besoin d'un retour d'information sur son émission et ceci n'est possible qu'en ajoutant un process multiplexeur.

La modélisation comportementale du bus CAN nécessitera aussi la prise en compte du bit stuffing et donc de devoir vérifier chaque bit émis. De même pour la station réceptrice devra être capable d'effacer ces bits,

car cela peut influencer les temps de transmission notamment en cas de présence de beaucoup de données.

Cette version du modèles CAN est plus complexe et pourrait poser problème pour des exemples plus complets avec plus de stations connectées au bus. Néanmoins, il permet de mieux cerner le protocole CAN et peut être une piste pour un modèle plus précis de ce bus.

3.5.2 ARINC429

Le protocole de communication est basé sur le système de maître/esclave. Il y aura donc une station maîtresse, appelée *master* initiant toutes les communications et une ou plusieurs stations esclave en attente de signal depuis la station maître (voir annexe A.

Les trames de données utilisées par les bus ARINC429 sont toutes d'une taille fixe de 32 bits avec 4 bits vides d'espace entre 2 trames.

- label : codé sur 8 bits, il correspond au type de données envoyées.
- données : codées sur 19 bits

Les autres bits servent pour la partie contrôle du protocole, partie que nous ne modéliserons pas.

3.5.3 ARINC629

Le protocole de communication est le TDMA, c'est à dire que l'ordre dans lequel les stations sont autorisées à transmettre est géré par des timers. Ces timers sont synchronisés par une station spécifique appelée *station leader*. Cette station est élue aléatoirement à la première initialisation ou en cas de changement sur le bus qui nécessite une nouvelle initialisation.

Les autres stations se verrons attribuer un numéro d'identifiant qui sera "activé" par la station leader afin d'autoriser leur accès au bus. Cet ordre sera déterminé par la station leader.

3.5.4 MIL-STD-1553

Pour ce modèle, nous utiliserons 3 stations représentées par des processeurs. L'une d'elle sera le contrôleur et les 2 autres seront de simples stations sur le réseau. Nous modéliserons les différentes communications possibles comme montrées sur la figure 12.

Conclusion

Ainsi des langages comme l'AADL offre la possibilité de construire des systèmes complexes en utilisant les outils mis à disposition par ce langage sans pour autant nécessiter des connaissances approfondies de tels systèmes.

La possibilité de pouvoir analyser ces modèles afin de pouvoir les vérifier est donc cruciale. Seulement de tels procédés sont récents et encore que peu documentés. Il faut donc effectuer des recherches en amont afin de déterminer quels modèles seraient les plus précis.

Donc l'objectif de ce stage était d'établir des premiers modèles pour des systèmes distribués ainsi que les techniques afin de créer ces modèles pour qu'ils puissent être intégrer au logiciel AADLInspector. Modèles qui permettront l'analyse temps réel des architectures à l'aide de Cheddar.

Des premiers modèles de systèmes distribués temps réel ont était établi, en AADL et l'équivalent en Cheddar ADL ainsi que les protocoles pour passer d'un modèles AADL à un modèle pouvant être traité avec Cheddar. Maintenant, il reste encore à étendre ces modèles aux autres bus temps réel existant ainsi que d'affiner ces modèles.

A Modèles AADL ARINC429 avec offset

Dans cette annexe, se trouve un exemple de système utilisant un bu ARINC 429 en AADL.

A.1 Bus library

```
BUS arinc429
PROPERTIES
  Transmission_Type => PUSH;
  Bus_Properties::Channel_Type => SIMPLEX;
END arinc429;

BUS IMPLEMENTATION arinc429.high_speed
PROPERTIES
  -- Bandwith = 100 Kbits/s
  -- Byte length = 80 us
  --Transmission_Time => [Fixed => 1 us .. 1 us; PerByte => 79 us .. 81 us;];

  -- Random values for tests
  Transmission_Time => [Fixed => 0 us .. 0 us; PerByte => 1 ms .. 1 ms;];
END arinc429.high_speed;
```

A.2 Hardware

```
SYSTEM hw
END hw;

SYSTEM IMPLEMENTATION hw.impl
SUBCOMPONENTS
  p1 : PROCESSOR proc.impl;
  p2 : PROCESSOR proc.impl;
  bus_arinc429 : BUS bus_library::arinc429.high_speed;
CONNECTIONS
  ba1 : BUS ACCESS bus_arinc429 -> p1.bus_arinc429;
  ba2 : BUS ACCESS bus_arinc429 -> p2.bus_arinc429;
END hw.impl;

PROCESSOR proc
FEATURES
  bus_arinc429 : REQUIRES BUS ACCESS bus_library::arinc429.high_speed;
PROPERTIES
  Scheduling_Protocol => POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL;
```

```
END proc;

PROCESSOR IMPLEMENTATION proc.impl
END proc.impl;
```

A.3 Software

```
SYSTEM sw
END sw;

SYSTEM IMPLEMENTATION sw.impl
SUBCOMPONENTS
  t1_p1 : PROCESS t1_p1_process.impl;
  t2_p1 : PROCESS t2_p1_process.impl;
  t3_p1 : PROCESS t3_p1_process.impl;
  t4_p1 : PROCESS t4_p1_process.impl;
  t5_p1 : PROCESS t5_p1_process.impl;
  t1_p2 : PROCESS t1_p2_process.impl;
  t2_p2 : PROCESS t2_p2_process.impl;
  t3_p2 : PROCESS t3_p2_process.impl;
  t4_p2 : PROCESS t4_p2_process.impl;
CONNECTIONS
  dc1 : PORT t1_p1.data_out -> t1_p2.data_in;
  dc2 : PORT t2_p1.data_out -> t2_p2.data_in;
  dc3 : PORT t3_p1.data_out -> t3_p2.data_in;
END sw.impl;

-----------------------------------------------------
-- T1_P1
-----------------------------------------------------
process t1_p1_process
features
  data_out : out event data port bus_library::arinc429_frame;
end t1_p1_process;

PROCESS IMPLEMENTATION t1_p1_process.impl
SUBCOMPONENTS
  emit : THREAD t1_p1_thread.impl;
CONNECTIONS
  dc1 : PORT emit.data_out -> data_out;
END t1_p1_process.impl;

thread t1_p1_thread
```

```
features
  data_out : out event data port bus_library::arinc429_frame {
    Source_Data_Size => 1 Kbyte;
  };
properties
  Period => 5ms;
  Compute_Execution_Time => 1ms..1ms;
  Priority => 5;
  Dispatch_Protocol => Periodic;
end t1_p1_thread;

thread implementation t1_p1_thread.impl
end t1_p1_thread.impl;

-----------------------------------------------------
-- T2_P1
-----------------------------------------------------
process t2_p1_process
features
  data_out : out event data port bus_library::arinc429_frame;
end t2_p1_process;

PROCESS IMPLEMENTATION t2_p1_process.impl
SUBCOMPONENTS
  emit : THREAD t2_p1_thread.impl;
CONNECTIONS
  dc1 : PORT emit.data_out -> data_out;
END t2_p1_process.impl;

thread t2_p1_thread
features
  data_out : out event data port bus_library::arinc429_frame {
    Source_Data_Size => 2 Kbyte;
  };
properties
  Period => 10ms;
  Compute_Execution_Time => 2ms..2ms;
  Priority => 4;
  Dispatch_Protocol => Periodic;
end t2_p1_thread;

thread implementation t2_p1_thread.impl
end t2_p1_thread.impl;
```

```
-----------------------------------------------------
-- T3_P1
-----------------------------------------------------
process t3_p1_process
features
  data_out : out event data port bus_library::arinc429_frame;
end t3_p1_process;

PROCESS IMPLEMENTATION t3_p1_process.impl
SUBCOMPONENTS
  emit : THREAD t3_p1_thread.impl;
CONNECTIONS
  dc1 : PORT emit.data_out -> data_out;
END t3_p1_process.impl;

thread t3_p1_thread
features
  data_out : out event data port bus_library::arinc429_frame {
    Source_Data_Size => 3 Kbyte;
  };
properties
  Period => 15ms;
  Compute_Execution_Time => 1ms..1ms;
  Priority => 3;
  Dispatch_Protocol => Periodic;
end t3_p1_thread;

thread implementation t3_p1_thread.impl
end t3_p1_thread.impl;

-----------------------------------------------------
-- T4_P1
-----------------------------------------------------
process t4_p1_process
end t4_p1_process;

PROCESS IMPLEMENTATION t4_p1_process.impl
SUBCOMPONENTS
  idle : THREAD t4_p1_thread.impl;
END t4_p1_process.impl;

thread t4_p1_thread
properties
  Period => 30ms;
```

```
  Compute_Execution_Time => 4ms..4ms;
  Priority => 2;
  Dispatch_Protocol => Periodic;
end t4_p1_thread;

thread implementation t4_p1_thread.impl
end t4_p1_thread.impl;

-----------------------------------------------------
-- T5_P1
-----------------------------------------------------
process t5_p1_process
end t5_p1_process;

PROCESS IMPLEMENTATION t5_p1_process.impl
SUBCOMPONENTS
  idle : THREAD t5_p1_thread.impl;
END t5_p1_process.impl;

thread t5_p1_thread
properties
  Period => 30ms;
  Compute_Execution_Time => 2ms..2ms;
  Priority => 1;
  Dispatch_Protocol => Periodic;
end t5_p1_thread;

thread implementation t5_p1_thread.impl
end t5_p1_thread.impl;

-----------------------------------------------------
-- T1_P2
-----------------------------------------------------
process t1_p2_process
features
  data_in : in event data port bus_library::arinc429_frame;
end t1_p2_process;

PROCESS IMPLEMENTATION t1_p2_process.impl
SUBCOMPONENTS
  receive : THREAD t1_p2_thread.impl;
CONNECTIONS
  dc1 : PORT data_in -> receive.data_in;
END t1_p2_process.impl;
```

```
thread t1_p2_thread
features
  data_in : in event data port bus_library::arinc429_frame;
properties
  Period => 5ms;
  Compute_Execution_Time => 1ms..1ms;
  Priority => 5;
  Dispatch_Offset => 2ms;
  Dispatch_Protocol => Periodic;
end t1_p2_thread;

thread implementation t1_p2_thread.impl
end t1_p2_thread.impl;

-----------------------------------------------------
-- T2_P2
-----------------------------------------------------
process t2_p2_process
features
  data_in : in event data port bus_library::arinc429_frame;
end t2_p2_process;

PROCESS IMPLEMENTATION t2_p2_process.impl
SUBCOMPONENTS
  receive : THREAD t2_p2_thread.impl;
CONNECTIONS
  dc1 : PORT data_in -> receive.data_in;
END t2_p2_process.impl;

thread t2_p2_thread
features
  data_in : in event data port bus_library::arinc429_frame;
properties
  Period => 10ms;
  Compute_Execution_Time => 2ms..2ms;
  Priority => 4;
  Dispatch_Offset => 5ms;
  Dispatch_Protocol => Periodic;
end t2_p2_thread;

thread implementation t2_p2_thread.impl
end t2_p2_thread.impl;
```

```
----------------------------------------------------
-- T3_P2
----------------------------------------------------
process t3_p2_process
features
  data_in : in event data port bus_library::arinc429_frame;
end t3_p2_process;

PROCESS IMPLEMENTATION t3_p2_process.impl
SUBCOMPONENTS
  receive : THREAD t3_p2_thread.impl;
CONNECTIONS
  dc1 : PORT data_in -> receive.data_in;
END t3_p2_process.impl;

thread t3_p2_thread
features
  data_in : in event data port bus_library::arinc429_frame;
properties
  Period => 15ms;
  Compute_Execution_Time => 1ms..1ms;
  Priority => 3;
  Dispatch_Offset => 6ms;
  Dispatch_Protocol => Periodic;
end t3_p2_thread;

thread implementation t3_p2_thread.impl
end t3_p2_thread.impl;

----------------------------------------------------
-- T4_P2
----------------------------------------------------
process t4_p2_process
end t4_p2_process;

PROCESS IMPLEMENTATION t4_p2_process.impl
SUBCOMPONENTS
  idle : THREAD t4_p2_thread.impl;
END t4_p2_process.impl;

thread t4_p2_thread
properties
  Period => 15ms;
  Compute_Execution_Time => 3ms..3ms;
```

```
  Priority => 2;
  Dispatch_Protocol => Periodic;
end t4_p2_thread;

thread implementation t4_p2_thread.impl
end t4_p2_thread.impl;
```

A.4 Système

```
SYSTEM arinc429_offset
END arinc429_offset;

SYSTEM IMPLEMENTATION arinc429_offset.impl
SUBCOMPONENTS
  hard : SYSTEM hw::hw.impl;
  soft : SYSTEM sw::sw.impl;
PROPERTIES
  Actual_Processor_Binding => (REFERENCE(hard.P1)) APPLIES TO soft.t1_p1;
  Actual_Processor_Binding => (REFERENCE(hard.P1)) APPLIES TO soft.t2_p1;
  Actual_Processor_Binding => (REFERENCE(hard.P1)) APPLIES TO soft.t3_p1;
  Actual_Processor_Binding => (REFERENCE(hard.P1)) APPLIES TO soft.t4_p1;
  Actual_Processor_Binding => (REFERENCE(hard.P1)) APPLIES TO soft.t5_p1;
  Actual_Processor_Binding => (REFERENCE(hard.P2)) APPLIES TO soft.t1_p2;
  Actual_Processor_Binding => (REFERENCE(hard.P2)) APPLIES TO soft.t2_p2;
  Actual_Processor_Binding => (REFERENCE(hard.P2)) APPLIES TO soft.t3_p2;
  Actual_Processor_Binding => (REFERENCE(hard.P2)) APPLIES TO soft.t4_p2;

  Actual_Connection_Binding => (REFERENCE(hard.bus_arinc429))
APPLIES TO soft.dc1;
  Actual_Connection_Binding => (REFERENCE(hard.bus_arinc429))
APPLIES TO soft.dc2;
  Actual_Connection_Binding => (REFERENCE(hard.bus_arinc429))
APPLIES TO soft.dc3;
END arinc429_offset.impl;
```

B Modèles Cheddar ARINC 429 avec offset

Dans cette section nous verrons l'équivalent en Cheddar ADL du modèles AADL du système distribué ARINC 429 dans l'annexe A.

La partie concernant les *address_spaces* et *buffers* ne sont pas présentes car elles ne présente qu'un intérêt mineurs pour cet exemple.

```
<?xml version="1.0" standalone="yes"?>
<cheddar>
  <core_units>
    <core_unit id="root.hard.p1.core">
      <name>root.hard.p1.core</name>
      <object_type>CORE_OBJECT_TYPE</object_type>
      <speed>1.0</speed>
      <scheduling>
        <scheduling_parameters>
          <scheduler_type>POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL</scheduler_type>
          <quantum>0</quantum>
          <preemptive_type>PREEMPTIVE</preemptive_type>
        </scheduling_parameters>
      </scheduling>
    </core_unit>
    <core_unit id="root.hard.p2.core">
      <name>root.hard.p2.core</name>
      <object_type>CORE_OBJECT_TYPE</object_type>
      <speed>1.0</speed>
      <scheduling>
        <scheduling_parameters>
          <scheduler_type>POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL</scheduler_type>
          <quantum>0</quantum>
          <preemptive_type>PREEMPTIVE</preemptive_type>
        </scheduling_parameters>
      </scheduling>
    </core_unit>
    <core_unit id="root.hard.bus_arinc429.core">
      <name>root.hard.bus_arinc429.core</name>
      <object_type>CORE_OBJECT_TYPE</object_type>
      <speed>1.0</speed>
      <scheduling>
        <scheduling_parameters>
          <scheduler_type>POSIX_1003_HIGHEST_PRIORITY_FIRST_PROTOCOL</scheduler_type>
          <quantum>0</quantum>
          <preemptive_type>NOT_PREEMPTIVE</preemptive_type>
        </scheduling_parameters>
      </scheduling>
    </core_unit>
  </core_units>
  <processors>
    <mono_core_processor id="root.hard.p1">
      <name>root.hard.p1</name>
      <object_type>PROCESSOR_OBJECT_TYPE</object_type>
```

```
    <processor_type>MONOCORE_TYPE</processor_type>
    <network>No_Network</network>
    <migration_type>NO_MIGRATION_TYPE</migration_type>
    <core ref="root.hard.p1.core"/>
  </mono_core_processor>
  <mono_core_processor id="root.hard.p2">
    <name>root.hard.p2</name>
    <object_type>PROCESSOR_OBJECT_TYPE</object_type>
    <processor_type>MONOCORE_TYPE</processor_type>
    <network>No_Network</network>
    <migration_type>NO_MIGRATION_TYPE</migration_type>
    <core ref="root.hard.p2.core"/>
  </mono_core_processor>
  <mono_core_processor id="root.hard.bus_arinc429">
    <name>root.hard.bus_arinc429</name>
    <object_type>PROCESSOR_OBJECT_TYPE</object_type>
    <processor_type>MONOCORE_TYPE</processor_type>
    <network>No_Network</network>
    <migration_type>NO_MIGRATION_TYPE</migration_type>
    <core ref="root.hard.bus_arinc429.core"/>
  </mono_core_processor>
</processors>
<tasks>
  <periodic_task id="root.hard.p1.soft.t1_p1.emit">
    <name>root.hard.p1.soft.t1_p1.emit</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p1.soft.t1_p1</address_space_name>
    <cpu_name>root.hard.p1</cpu_name>
    <policy>SCHED_FIFO</policy>
    <period>5</period>
    <capacity>1</capacity>
    <deadline>5</deadline>
    <priority>5</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
  </periodic_task>
  <periodic_task id="root.hard.p1.soft.t2_p1.emit">
```

```xml
    <name>root.hard.p1.soft.t2_p1.emit</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p1.soft.t2_p1</address_space_name>
    <cpu_name>root.hard.p1</cpu_name>
    <policy>SCHED_FIFO</policy>
    <period>10</period>
    <capacity>2</capacity>
    <deadline>10</deadline>
    <priority>4</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
  </periodic_task>
  <periodic_task id="root.hard.p1.soft.t3_p1.emit">
    <name>root.hard.p1.soft.t3_p1.emit</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p1.soft.t3_p1</address_space_name>
    <cpu_name>root.hard.p1</cpu_name>
    <policy>SCHED_FIFO</policy>
    <period>15</period>
    <capacity>1</capacity>
    <deadline>15</deadline>
    <priority>3</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
  </periodic_task>
  <periodic_task id="root.hard.p1.soft.t4_p1.idle">
    <name>root.hard.p1.soft.t4_p1.idle</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p1.soft.t4_p1</address_space_name>
```

```
  <cpu_name>root.hard.p1</cpu_name>
  <policy>SCHED_FIFO</policy>
  <period>30</period>
  <capacity>4</capacity>
  <deadline>30</deadline>
  <priority>2</priority>
  <start_time>0</start_time>
  <criticality>0</criticality>
  <jitter>0</jitter>
  <text_memory_size>0</text_memory_size>
  <stack_memory_size>0</stack_memory_size>
  <predictable_seed>TRUE</predictable_seed>
  <seed>0</seed>
  <blocking_time>0</blocking_time>
</periodic_task>
<periodic_task id="root.hard.p1.soft.t5_p1.idle">
  <name>root.hard.p1.soft.t5_p1.idle</name>
  <object_type>TASK_OBJECT_TYPE</object_type>
  <task_type>PERIODIC_TYPE</task_type>
  <address_space_name>root.hard.p1.soft.t5_p1</address_space_name>
  <cpu_name>root.hard.p1</cpu_name>
  <policy>SCHED_FIFO</policy>
  <period>30</period>
  <capacity>2</capacity>
  <deadline>30</deadline>
  <priority>1</priority>
  <start_time>0</start_time>
  <criticality>0</criticality>
  <jitter>0</jitter>
  <text_memory_size>0</text_memory_size>
  <stack_memory_size>0</stack_memory_size>
  <predictable_seed>TRUE</predictable_seed>
  <seed>0</seed>
  <blocking_time>0</blocking_time>
</periodic_task>
<periodic_task id="root.hard.p2.soft.t1_p2.receive">
  <name>root.hard.p2.soft.t1_p2.receive</name>
  <object_type>TASK_OBJECT_TYPE</object_type>
  <task_type>PERIODIC_TYPE</task_type>
  <address_space_name>root.hard.p2.soft.t1_p2</address_space_name>
  <cpu_name>root.hard.p2</cpu_name>
  <policy>SCHED_FIFO</policy>
  <period>5</period>
  <capacity>1</capacity>
```

```
    <deadline>5</deadline>
    <priority>5</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
    <offsets>
      <offset_type>
        <offset_value>2</offset_value>
        <activation>0</activation>
      </offset_type>
    </offsets>
  </periodic_task>
  <periodic_task id="root.hard.p2.soft.t2_p2.receive">
    <name>root.hard.p2.soft.t2_p2.receive</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p2.soft.t2_p2</address_space_name>
    <cpu_name>root.hard.p2</cpu_name>
    <policy>SCHED_FIFO</policy>
    <period>10</period>
    <capacity>2</capacity>
    <deadline>10</deadline>
    <priority>4</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
    <offsets>
      <offset_type>
        <offset_value>5</offset_value>
        <activation>0</activation>
      </offset_type>
    </offsets>
  </periodic_task>
  <periodic_task id="root.hard.p2.soft.t3_p2.receive">
```

```
    <name>root.hard.p2.soft.t3_p2.receive</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p2.soft.t3_p2</address_space_name>
    <cpu_name>root.hard.p2</cpu_name>
    <policy>SCHED_FIFO</policy>
    <period>15</period>
    <capacity>1</capacity>
    <deadline>15</deadline>
    <priority>3</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
    <offsets>
      <offset_type>
        <offset_value>6</offset_value>
        <activation>0</activation>
      </offset_type>
    </offsets>
  </periodic_task>
  <periodic_task id="root.hard.p2.soft.t4_p2.idle">
    <name>root.hard.p2.soft.t4_p2.idle</name>
    <object_type>TASK_OBJECT_TYPE</object_type>
    <task_type>PERIODIC_TYPE</task_type>
    <address_space_name>root.hard.p2.soft.t4_p2</address_space_name>
    <cpu_name>root.hard.p2</cpu_name>
    <policy>SCHED_FIFO</policy>
    <period>15</period>
    <capacity>3</capacity>
    <deadline>15</deadline>
    <priority>2</priority>
    <start_time>0</start_time>
    <criticality>0</criticality>
    <jitter>0</jitter>
    <text_memory_size>0</text_memory_size>
    <stack_memory_size>0</stack_memory_size>
    <predictable_seed>TRUE</predictable_seed>
    <seed>0</seed>
    <blocking_time>0</blocking_time>
```

```
</periodic_task>
<periodic_task id="root.hard.bus_arinc429.dc1.soft.t1_p1.emit">
  <name>root.hard.bus_arinc429.dc1.soft.t1_p1.emit</name>
  <object_type>TASK_OBJECT_TYPE</object_type>
  <task_type>PERIODIC_TYPE</task_type>
  <address_space_name>root.hard.bus_arinc429.dc1</address_space_name>
  <cpu_name>root.hard.bus_arinc429</cpu_name>
  <policy>SCHED_FIFO</policy>
  <period>5</period>
  <deadline>5</deadline>
  <priority>1</priority>
  <capacity>1</capacity>
  <start_time>0</start_time>
  <criticality>0</criticality>
  <jitter>0</jitter>
</periodic_task>
<periodic_task id="root.hard.bus_arinc429.dc2.soft.t2_p1.emit">
  <name>root.hard.bus_arinc429.dc2.soft.t2_p1.emit</name>
  <object_type>TASK_OBJECT_TYPE</object_type>
  <task_type>PERIODIC_TYPE</task_type>
  <address_space_name>root.hard.bus_arinc429.dc2</address_space_name>
  <cpu_name>root.hard.bus_arinc429</cpu_name>
  <policy>SCHED_FIFO</policy>
  <period>10</period>
  <deadline>10</deadline>
  <priority>1</priority>
  <capacity>2</capacity>
  <start_time>0</start_time>
  <criticality>0</criticality>
  <jitter>0</jitter>
</periodic_task>
<periodic_task id="root.hard.bus_arinc429.dc3.soft.t3_p1.emit">
  <name>root.hard.bus_arinc429.dc3.soft.t3_p1.emit</name>
  <object_type>TASK_OBJECT_TYPE</object_type>
  <task_type>PERIODIC_TYPE</task_type>
  <address_space_name>root.hard.bus_arinc429.dc3</address_space_name>
  <cpu_name>root.hard.bus_arinc429</cpu_name>
  <policy>SCHED_FIFO</policy>
  <period>15</period>
  <deadline>15</deadline>
  <priority>1</priority>
  <capacity>3</capacity>
  <start_time>0</start_time>
  <criticality>0</criticality>
```

```
      <jitter>0</jitter>
    </periodic_task>
  </tasks>
  <resources>
  </resources>
  <dependencies>
    <dependency>
      <type_of_dependency>PRECEDENCE_DEPENDENCY</type_of_dependency>
      <precedence_sink ref="root.hard.bus_arinc429.dc1.soft.t1_p1.emit"/>
      <precedence_source ref="root.hard.p1.soft.t1_p1.emit"/>
    </dependency>
    <dependency>
      <type_of_dependency>PRECEDENCE_DEPENDENCY</type_of_dependency>
      <precedence_sink ref="root.hard.bus_arinc429.dc2.soft.t2_p1.emit"/>
      <precedence_source ref="root.hard.p1.soft.t2_p1.emit"/>
    </dependency>
    <dependency>
      <type_of_dependency>PRECEDENCE_DEPENDENCY</type_of_dependency>
      <precedence_sink ref="root.hard.bus_arinc429.dc3.soft.t3_p1.emit"/>
      <precedence_source ref="root.hard.p1.soft.t3_p1.emit"/>
    </dependency>
  </dependencies>
</cheddar>
```

Références

[COC] S. Cochard.
Étude des bus avioniques.
Travail d'Étude et de Recherche UBO 2002.

[AIC] Logiciel AADLInspector développé par Ellidiss Technologies.
http ://www.ellidiss.com/products/aadl-inspector/

[ADL] Architecture Analysis and Desgin Langage.
http ://www.aadl.info/aadl/currentsite/

[CDR] F. Singhoff, J. Legrand, L. Nana, L. Marcé.
Cheddar : a Flexible Real Time Scheduling Framework.
ACM SIGAda Ada Letters, volume 24, number 4, pages 1-8. Edited by ACM Press, New York, USA. December 2004, ISSN :1094-3641.

[TDL] Ken Tindell et John Clark.
Holistic schedulability analysis for distributed hard real-time systems.

[FAB] Ricardo Bedin França, Mamoun Filali Amine and Jean-Paul Bodeveix.
AADL Modeling of a Generic Bus.

[LAU] Michaël Lauer.
Une méthode globale pour la vérification d'exigences temps réel. Application à l'Avionique Modulaire Intégrée.

[THW] K. W. Tindell, H. Hansson, A. J. Wellings.
Analysing Real-Time Communications : Controller Area Network (CAN).

[DUP] B. Dupouy.
Introduction aux bus et réseaux temps réel.

[SGF] F. Singhoff.
Real-time Scheduling Analysis.