Analyse de diagnosticabilité d'architecture de fonctions embarquées - Application aux architectures automobiles

Manel Khlif

▶ To cite this version:

Manel Khlif. Analyse de diagnosticabilité d'architecture de fonctions embarquées - Application aux architectures automobiles. Systèmes embarqués. Université de Technologie de Compiègne, 2010. Français. NNT : . tel-00801608

HAL Id: tel-00801608
https://theses.hal.science/tel-00801608

Submitted on 17 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Technologie de Compiègne

Laboratoire Heudiasyc UMR 6599

Thèse

pour obtenir le grade de
DOCTEUR de L'Université de Technologie de Compiègne
Spécialité : Génie Informatique
préparée au laboratoire Heudiasyc,
dans le cadre de l'Ecole Doctorale Technologies de l'Information et de Systèmes (TIS)

Analyse de diagnosticabilité d'architecture de fonctions embarquées

Application aux architectures automobiles

Manel KHLIF

06/12/2010

Directeur de Thèse :

Mohamed SHAWKY

JURY :

M. Mohamed ABID	Professeur à l'Université de Sfax, Tunisie	Rapporteur
M. Guy GOGNIAT	Professeur à l'Université de Bretagne Sud	Rapporteur
M. Claude COVO	Coordinateur de la recherche à Renault Trucks, Lyon	Examinateur
M. Philippe DAGUE	Professeur à l'Université Paris-Sud	Examinateur
M. Walter SCHÖN	Professeur à l'Université de Technologie de Compiègne	Examinateur
M. Mohamed SHAWKY	Professeur à l'Université de Technologie de Compiègne	Directeur de thèse

Remerciements

Je tiens à exprimer tout d'abord mes remerciements à Mohamed SHAWKY, Professeur des universités de l'Université de Technologie de Compiègne, qui a dirigé cette thèse. Il a su orienter mes recherches aux bons moments en me faisant découvrir les systèmes électroniques embarqués. Malgré ses diverses responsabilités, il a toujours été disponible pour d'intenses et rationnelles discussions. Pour tout cela ainsi que pour sa confiance et son soutien tout au long de ma thèse, je le remercie vivement.

Merci à Walter SCHÖN, Professeur des universités de l'Université de Technologie de Compiègne, d'avoir accepté de présider le jury de cette thèse, à mes deux rapporteurs de thèse Mohamed ABID, Professeur des universités à l'Université de Sfax en Tunisie et Guy GOGNIAT, Professeur des universités à l'Université de Bretagne-Sud, pour la rapidité avec laquelle ils ont lu mon manuscrit et l'intérêt qu'ils ont porté à mon travail.

Merci également à Philippe DAGUE, Professeur des universités de l'Université de Paris Sud et Claude COVO, coordinateur de la recherche de Renault Trucks, d'avoir accepté d'examiner mon mémoire et de faire partie de mon jury de thèse.

Je n'oublierai pas les aides permanentes du personnel administratif du Département du Génie Informatique: Isabelle, Sabine, Nathalie, Céline, Magalie,… que je remercie beaucoup.

Finalement, j'adresse un grand Merci à toute ma famille qui a toujours été présente lorsque j'en ai eu besoin, en particulier mon mari, mes parents et mes deux sœurs pour leurs encouragements permanents.

À mes parents,
À mes deux sœurs,
À l'homme de ma vie,
À mes enfants,
À tous ceux qui m'aiment.

Résumé

Un système embarqué peut être défini comme un système électronique et informatique autonome, dédié à une tâche bien définie et soumis à des contraintes. Les défaillances des systèmes embarqués sont de plus en plus difficiles à prévoir, comprendre et réparer. Des travaux sur la sûreté de fonctionnement ont mis au point les techniques de vérification et des recommandations de conception pour maîtriser les risques. En même temps d'autres travaux ont entrepris d'améliorer la fiabilité de ces systèmes en rénovant les méthodologies de conception. Les méthodes de diagnostic, à leur tour, ont évolué afin d'améliorer la tolérance des systèmes embarqués aux pannes et leur capacité à s'auto-diagnostiquer. Ainsi, le domaine de l'analyse de la « diagnosticabilité » a vu le jour.

Aujourd'hui, le concepteur d'un système doit s'assurer que celui-ci est diagnosticable, c'est-à-dire que les fautes qui peuvent y apparaitre sont identifiables, avant de construire ou fabriquer le système. Les méthodes d'analyse de la diagnosticabilité se focalisent sur ce que nous appelons « la diagnosticabilité fonctionnelle » où l'architecture matérielle du système n'était pas directement considérée.

Cette thèse contribue à l'analyse de l'impact de l'interaction des fonctions-architecture sur la diagnosticabilité d'un système embarqué. L'approche que nous avons conçue est intégrable dans le cycle de conception des systèmes embarqués ; elle commence par l'analyse de la diagnosticabilité des systèmes à événements discrets (telle qu'elle est présentée dans la littérature). Notre méthode, exige ensuite la vérification d'un ensemble de propriétés que nous avons définies et appelées « propriétés de la diagnosticabilité fonctionnelle-architecturale ».

La vérification des propriétés s'effectue en deux étapes : la première étape est la vérification de la description de l'architecture (réalisée en AADL) et la deuxième étape est la vérification de l'interaction fonctions-architecture (réalisée en SystemC-Simulink). Pour l'analyse de l'interaction des fonctions avec l'architecture, réalisée en SystemC-Simulink, nous avons développé un prototype d'outil COSITA basé sur l'analyse des traces de la co-simulation du co-modèle. Nous avons comparé les résultats de l'analyse des traces de co-simulation avec des résultats que nous avons obtenus suite à une émulation sur une plateforme physique automobile dans le laboratoire Heudiasyc. Finalement, nous avons mis au point à travers cette thèse une méthodologie originale d'analyse de la diagnosticabilité qui prend en considération les contraintes de l'architecture matérielle du système.

Abstract

An embedded system can be defined as a constrained autonomous hardware and software system, dedicated to a specific task. The failures of embedded systems are increasingly difficult to predict, understand and repair. Research works on dependability have developed verification techniques and design recommendations to control risks. In the same time, other works were undertaken to improve the reliability of these systems by upgrading the design methodologies. Diagnostic methods, in turn, have evolved to improve the tolerance of embedded systems to faults and their ability for self-diagnosis, thus, the field of analysis of « diagnosability » has emerged.

Nowadays, system designers must ensure that a system is diagnosable, that the faults that may appear can be identified before building or deploying the system. Current methods of diagnosability analysis focus on what we call « functional diagnosability », where the hardware architecture of the system is not directly considered. This thesis contributes to the analysis of the impact of the function-architecture interaction on the diagnosability of an embedded system. Our approach can be integrated into the design cycle of embedded systems; it begins by analyzing the diagnosability of discrete event systems (as presented in the literature). Our method then requires the verification of a set of properties that we defined and called the functional-architectural diagnosability properties. Property verification is done in two stages: the first step is to check the description of the architecture (described in AADL) with respect to the diagnosability requirements, and the second step is to check the interaction « function-architecture » (described in SystemC-Simulink). We developed a prototype tool suite COSITA based on co-simulation trace analysis. We compared the results of trace analysis issued from co-simulation with those issued from the system emulation (Hardware In the Loop) on an automotive physical platform in Heudiasyc laboratory. Finally, we have developed through this thesis a new methodology for analyzing the diagnosability which takes into account the constraints of the hardware architecture of the system.

Table des matières

RESUME **3**

ABSTRACT **9**

TABLE DES MATIERES **11**

TABLE DES FIGURES **13**

TABLE DES TABLEAUX **15**

INTRODUCTION **16**

I. CONTEXTE ET PROBLEMATIQUE 16
II. DEMARCHE SCIENTIFIQUE ET DESCRIPTION DU MANUSCRIT 17

ARCHITECTURES EMBARQUEES ET DIAGNOSTICABILITE **19**

I. INTRODUCTION 19
II. LES DEFIS DE LA CONCEPTION DES SYSTEMES EMBARQUES 19
1. Conception matérielle et conception logicielle 20
2. Différentes exigences des systèmes 20
III. METHODOLOGIES DE CONCEPTION DES SYSTEMES EMBARQUES 20
1. L'ingénierie des systèmes 20
a. Ingénierie Best-effort 20
b. Conception des systèmes critiques 21
2. La conception basée modèle 21
a. Quelques méthodologies de la conception « basée modèles » 21
b. La conception itérative 22
IV. ANALYSE DE MODELES POUR LA CONCEPTION D'UNE ARCHITECTURE EMBARQUEE SURE 23
1. Exploration de l'espace de conception par l'analyse de modèles 24
2. Techniques d'analyse de modèles pour garantir la sûreté de fonctionnement 25
a. Arbres de défaillance (FTA) 25
b. L'Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité (AMDEC) 26
c. Analyse Préliminaire des risques 27
d. Vérification formelle 27
V. LE DIAGNOSTIC BASE MODELE 27
1. Les approches alternatives du diagnostic basé modèle 28
2. Les différentes communautés du diagnostic basé modèle 29
3. Les différentes approches de modélisation des systèmes à diagnostiquer 29
a. Les systèmes continus 30
b. Les systèmes discrets (ou systèmes à événements discrets) 30
c. Les systèmes hybrides 30
4. Les différents aspects et contextes des systèmes à diagnostiquer 30
a. Diagnostic centralisé et distribué 30
b. Les contextes en ligne et hors-ligne 31
VI. LES DIFFERENTES APPROCHES D'ANALYSE DE DIAGNOSTICABILITE 31
1. Les approches basées états pour l'analyse de la diagnosticabilité 32
2. Les approches basées événements pour l'analyse de la diagnosticabilité 32
3. Les approches hybrides pour l'analyse de la diagnosticabilité 32
VII. SYNTHESE DES DIFFERENTES APPROCHES D'ANALYSE DE DIAGNOSTICABILITE 33
VIII. CONCLUSION 33

ANALYSE DE LA DIAGNOSTICABILITE ENTRE ARCHITECTURE MATERIELLE ET FONCTION **35**

I. INTRODUCTION 35
II. ILLUSTRATION DE LA DIAGNOSTICABILITE DES SYSTEMES A EVENEMENTS DISCRETS 35
III. VERIFICATION DES PROPRIETES DE LA DIAGNOSTICABILITE FONCTIONNELLE-ARCHITECTURALE 40
1. Définition des propriétés de la diagnosticabilité fonctionnelle-architecturale 40
2. Vérification des propriétés 49

3. La co-modélisation matérielle-logicielle ... 50
4. La co-simulation ... 51
5. Couverture des conditions ... 52
6. Analyse des traces : l'outil COSITA ... 53
7. Interprétation de l'analyse des métriques ... 55
IV. CONCLUSION ... 56

ETUDE DE CAS : LA FONCTION SDK (SMART DISTANCE KEEPING) ... 57

I. INTRODUCTION ... 57
II. LES ARCHITECTURES ELECTRONIQUES EMBARQUEES DES VEHICULES ... 57
III. LA FONCTION SDK ... 59
1. Aperçu sur la fonction SDK ... 59
2. Le modèle mathématique de la fonction SDK ... 61
IV. ANALYSE DE LA DIAGNOSTICABILITE FONCTIONNELLE-ARCHITECTURALE DE SDK ... 62
1. Analyse de la diagnosticabilité fonctionnelle de SDK ... 62
2. Description de l'architecture embarquée ... 65
3. Vérification des propriétés de la diagnosticabilité fonctionnelle-architecturale ... 73
4. Co-Modélisation et co-simulation matérielle-logicielle ... 77
a. Modélisation de SDK en Simulink ... 77
b. Modélisation des ECUs et du bus CAN en SystemC ... 79
c. Co-simulation du co-modèle et analyse des traces ... 82
5. Résultat de l'analyse de la diagnosticabilité fonctionnelle-architecturale ... 84
a. Analyse de la disponibilité temporelle ... 84
b. Analyse de l'observabilité ... 85
V. CONCLUSION ... 86

DE L'ANALYSE DE LA DIAGNOSTICABILITE VERS LA CONCEPTION ... 87

I. INTRODUCTION ... 87
II. VALIDATION PAR L'EMULATION ... 87
1. La plateforme DIAFORE ... 87
2. Programmation de la plateforme ... 88
3. Emulation de SDK sur la plateforme Diafore ... 89
4. Analyse de la diagnosticabilité ... 90
a. Vérification des propriétés de l'architecture ... 90
b. Vérification des propriétés de l'interaction fonctions-architecture ... 90
c. Comparaison des résultats de l'émulation avec les résultats de la co-simulation (module SIMECO) ... 91
III. VALIDATION PAR RETOUR A LA CONCEPTION ... 91
1. Première proposition de modification de l'architecture ... 92
2. Deuxième proposition de modification de l'architecture ... 93
3. Comparaison des deux solutions ... 102
IV. CONCLUSION ... 103

CONCLUSION ET PERSPECTIVES ... 104

REFERENCES ... 106

LIVRABLES DE PROJETS ... 110

ANNEXES ... 111

Annexe A. Co modèle SystemC-Simulink des deux calculateurs ECU1 et ECU2 ... 111
Annexe B. OBSAN ... 116
Annexe C. La différence entre deux fichiers traces ... 117
Annexe D. COSITA ... 118
Annexe E. La Librairie MOTOHAWK ... 119
Annexe F. MOTOTUNE ... 120
Annexe G. Programme CAPL pour la génération du fichier .LOG ... 121
Annexe H. log2vcd ... 122
Annexe I. Installation de la librairie « engine.h » ... 127
Annexe J. Le protocole CAN ... 129

Table des figures

Figure 1. Processus de conception 19
Figure 2. Processus itératif de conception 22
Figure 3. Evolution dans le temps de quelques méthodologies de conception 23
Figure 4. Illustration de l'exploration de l'espace de conception 24
Figure 5. Arbre de défaillances 25
Figure 6. Criticité= Occurrence*Gravité*Détection 26
Figure 7. Les différentes approches du diagnostic 28
Figure 8. Le diagnostic basé modèles. 29
Figure 9. Les différentes structures de diagnostic 31
Figure 10. Synthèse sur les approches d'analyse de diagnosticabilité 33
Figure 11. L'exemple du système HVAC (*Heating, Ventilating, Air Conditioning*) : Système de Chauffage, Ventilation et Air Conditionné 35
Figure 12. Sous-système de HVAC : Pompe, Valve et Contrôleur (PVC) 36
Figure 13. Modèle de comportement des composants 36
Figure 14. Modèle fonctionnel global 37
Figure 15. La séquence d'événements observables du sous-système PVC 37
Figure 16. Observateur des événements du système HVAC 38
Figure 17. Diagnostiqueur du système PVC 39
Figure 18. Résultat de l'analyse (non diagnosticable) 40
Figure 19. Processus d'observation 45
Figure 20. Exemple d'événement 46
Figure 21. Cycles d'exécution de processus 48
Figure 22. Classification des propriétés selon différents critères 49
Figure 23. Répartition des fonctions sur le matériel 50
Figure 24. Analyse de la diagnosticabilité fonctionnelle-architecturale 50
Figure 25. Format du fichier VCD 53
Figure 26. Les différents modules de l'outil COSITA 54
Figure 27. Processus itératif de vérification de la diagnosticabilité basé co-modélisation matérielle-logicielle 56
Figure 28. Evolution du nombre de fonctions électriques dans le temps 57
Figure 29. Architecture interne d'un ECU 58
Figure 30. Liaisons entre capteurs, ECUs et actionneurs 58
Figure 31. Valeurs issues du radar 59
Figure 32. Détection d'un véhicule dans le chemin actuel 60
Figure 33. Détection d'un véhicule qui provient de la voie de coté 60
Figure 34. Trafic dense 60
Figure 35. Composants de l'architecture électronique participants à la SDK 61
Figure 36. Modèle de comportement des composants de la fonction SDK 63
Figure 37. Automate fini du comportement des calculateurs 63
Figure 38. Automate fini complet de la fonction SDK 64
Figure 39. Observateur des événements de la fonction SDK 64
Figure 40. Diagnostiqueur de la fonction SDK 65
Figure 41. Modèle Simulink de la fonction SDK 78
Figure 42. Block « Signal Builder » pour la génération des entrées de la fonction SDK 79
Figure 43. Exemple de scénario prédéfini pour les valeurs d'entrée de SDK 79
Figure 44. Aperçu du co-modèle SystemC-Simulink 80

Figure 45. Format de la trame CAN 81
Figure 46. Vitesse du véhicule avant et vitesse du véhicule arrière 82
Figure 47. Distance de sécurité et distance relative 82
Figure 48. Chronogrammes des signaux des deux calculateurs 84
Figure 49. Résultat de l'analyse de la disponibilité temporelle 85
Figure 50. Les deux différents types de calculateurs utilisés pour la plateforme DIAFORE .. 87
Figure 51. La plateforme DIAFORE 88
Figure 52. Variation des signaux du SDK 89
Figure 53. Signal Oscillant 90
Figure 54. La conversion du .log au .vcd 90
Figure 55. Co-simulation avec ECU1 à une fréquence de 80MHz 93
Figure 56. Configuration de l'architecture matérielle-logicielle avec un calculateur supplémentaire 94
Figure 57. Interface graphique de la saisie des entrées du module OBSAN 116
Figure 58. Interface graphique du résultat (OBSAN) 116
Figure 59. Interface graphique d'accueil de COSITA 118
Figure 60.Modèle Simulink avec des blocs Motohawk pour la programmation d'un ECU .. 119
Figure 61. Interface graphique de Mototune 120
Figure 62. Fenêtre "Library files" 127
Figure 63. Fenêtre "Include Files" 127
Figure 64. Fenêtre Additional Dependencies 128
Figure 65. La trame CAN 130

Table des tableaux

Tableau 1. Diagnostic en ligne et diagnostic hors ligne 31
Tableau 2. Aperçu général sur les propriétés de la diagnosticabilité fonctionnelle-architecturale 41
Tableau 3. Quelques routines de la librairie "engine.h" 51
Tableau 4. Récapitulation des avantages et inconvénients des deux solutions proposées 102
Tableau 5. Différences entre deux fichiers trace 117
Tableau 6. Bus CAN : Émission-détection 129

Introduction

I. Contexte et problématique

De nos jours, les systèmes embarqués trouvent leur application dans de plus en plus de domaines ; le transport, les télécommunications, les équipements médicaux, etc. Un système embarqué peut être défini comme un système électronique et informatique autonome, comprenant matériel et logiciel, dédié à une tâche bien définie et soumis à des contraintes. Les contraintes sont dues à deux types d'interactions des processus de calcul avec le monde physique : la réaction à un environnement physique (impliquant des contraintes de réactivité tels que les délais, le débit, etc.), et l'exécution sur une plate-forme physique (imposant des contraintes d'exécution telles que la puissance de calcul, les taux de défaillance matérielle, etc.).

Les défaillances des systèmes automatisés complexes sont de plus en plus difficiles à prévoir, comprendre et réparer. La nécessité de méthodes et d'outils d'aide à la supervision de ces systèmes a initié de nombreux travaux de recherche. Ainsi, l'implémentation de ces systèmes a été naturellement modulaire pour maîtriser la complexité et les risques, en espérant que les défaillances ne touchent qu'une partie des modules. Or cette modularité, ou la distribution des fonctions, a contribué à la vulnérabilité de ce type de système.

Les travaux sur la sûreté de fonctionnement ont permis de mettre au point les techniques de vérification et des recommandations de conception pour maîtriser les risques. En même temps d'autres travaux ont entrepris d'améliorer la fiabilité de ces systèmes en revoyant les méthodologies de conception. Les méthodes de diagnostic, à leur tour, ont évolué des approches associatives vers les approches à base de modèles et migrent de plus en plus du diagnostic hors-ligne vers le diagnostic en ligne temps réel. Afin d'améliorer la tolérance des systèmes embarqués aux pannes et leur capacité à s'auto-diagnostiquer, le domaine de l'analyse de la « diagnosticabilité » a vu le jour.

Le concepteur d'un système doit s'assurer que celui-ci est diagnosticable, c'est-à-dire que les fautes qui peuvent y apparaitre sont identifiables. La diagnosticabilité devient une exigence à vérifier lors de la conception, de la même importance que les propriétés liées à la sûreté de fonctionnement afin de fournir un système plus fiable, avec des coûts de maintenance prévisibles.

Les méthodes d'analyse de la diagnosticabilité se focalisent sur ce qu'on peut appeler « la diagnosticabilité fonctionnelle » où l'architecture matérielle n'était pas directement considérée. Nous nous sommes donc intéressés à ce point bien précis : l'architecture informatique matérielle, voire le couple architecture matérielle-logicielle influence-t-il la diagnosticabilité d'un système ?

Le domaine de recherche de la diagnosticabilité est récent. Plusieurs communautés ont développé des approches différentes pour le diagnostic à base de modèles. En effet, pour l'analyse de la diagnosticabilité des architectures électroniques embarquées, les approches les plus utilisées sont les approches utilisant des modèles « basés événements discrets », ne considérant pas directement la partie matérielle de l'architecture embarquée au moment de la description du système.

II. Démarche scientifique et description du manuscrit

La démarche scientifique de cette thèse s'inspire à la fois des méthodologies de conception des systèmes embarqués qui portent intérêt à l'architecture matérielle ainsi que des méthodologies d'analyse de diagnosticabilité. Ainsi, le point de départ de ce manuscrit est une étude bibliographique sur les méthodologies existantes de conception des systèmes embarqués et les techniques existantes d'analyse de modèles pour aboutir à une architecture embarquée sûre. Suivie par un aperçu étendu sur le diagnostic à base de modèles et d'un état de l'art sur les différentes approches d'analyse de diagnosticabilité. Nous constatons donc, qu'avec les méthodes existantes, il existe une absence d'information sur l'architecture matérielle dans la description d'une architecture électronique embarquée.

La seconde partie présente le cœur de la contribution de cette thèse ; Pour ceci, nous partons d'un exemple d'analyse de diagnosticabilité faisant office de référence dans la littérature, celui du système HVAC (*Heating Ventilating Air Conditionning*) utilisant une approche basée événements discrets, à savoir l'approche du « diagnostiqueur » de M Sampath. Cet exemple sert à prouver que les approches d'analyse de diagnosticabilité basées sur une modélisation de niveau fonctionnel ne sont pas suffisantes pour des systèmes faisant interagir des fonctions sur une architecture électronique (plus précisément à base de processeurs). Notre approche définit alors, des propriétés servant à vérifier la description d'un système pour qu'il soit diagnosticable de point de vue fonctionnel-architectural. L'outil COSITA (*CO-Simulation Trace Analysis*), a été développé pour vérifier des propriétés liées à l'interaction des fonctions avec l'architecture à partir des traces de co-simulation du système, co-modélisé à priori en SystemC et Simulink [1] [2].

La troisième partie, présente d'abord les architectures électroniques embarquées en automobile. Ensuite, la fonction SDK (*Smart Distance Keeping*) est présentée comme cas d'étude pour illustrer les particularités de notre approche d'analyse de la diagnosticabilité fonctionnelle-architecturale.

Dans la dernière partie, nous validons notre approche en deux étapes ; Premièrement, nous étudions et comparons deux systèmes différents :
- le modèle SDK en Simulink sur le modèle de l'architecture matérielle en SystemC [3]
- et le même modèle de la fonction SDK sur une plateforme électronique automobile réelle.

Ceci permet de juger la crédibilité des résultats d'analyse de diagnosticabilité à la Co-simulation (obtenus à l'étape de la conception), avec ceux obtenus à l'émulation, c'est à dire à la phase de l'implémentation. Ensuite, la deuxième étape de la validation est le retour à la conception, qui montre que l'analyse de la diagnosticabilité d'une architecture électronique embarquée est à la fois une exigence de la conception et une métrique d'analyse du modèle.

La méthodologie mise au point dans cette thèse présente des techniques originales d'analyse de diagnosticabilité à base de multi-modèle. Nous avons pour ceci définis des propriétés de la diagnosticabilité fonctionnelle-architecturale extensibles pour d'éventuelles analyses à base de multi-modèle.

Ce travail de doctorat est réalisé dans le cadre d'un projet de recherche national intitulé DIAFORE[1] (Méthodes et outils pour le diagnostic de fonctions réparties). Ce projet financé en partie par l'ANR (Agence Nationale de la Recherche), rassemble trois équipes de trois laboratoires français, une équipe d'un industriel français et un techno provider:

- Heudiasyc (Heuristique et Diagnostic des Systèmes Complexes) : une unité mixte (UMR 6599) entre l'UTC (Université de Technologie de Compiègne et le CNRS (Centre National de la Recherche Scientifique),
- LRI (Laboratoire de Recherche en Informatique) : une unité mixte de recherche (UMR8623) de l'Université Paris-Sud et du CNRS,
- LIST (Laboratoire d'Intégration des Systèmes et des Technologies) du CEA (*Commissariat* à l'*énergie* atomique et aux énergies alternatives),
- Renault Trucks de Lyon
- Serma ingénierie filiale du GROUPE SERMA TECHNOLOGIES.

L'objectif de ce projet est de faciliter la détection de défaillances pour des fonctions distribuées sur plusieurs ECU (*Electronic Computing Unit*) [4].

[1] http://www.systematic-paris-region.org/fr/projets/diafore

Architectures embarquées et diagnosticabilité

I. Introduction

Dans ce premier chapitre de la thèse, nous parlerons en premier lieu des défis et des méthodologies de la conception des systèmes embarqués telle que la dépendance mutuelle du matériel et du logiciel embarqué. En second lieu, nous évoquerons quelques techniques d'analyse de modèles pour garantir une architecture embarquée sûre. Ensuite, nous présenterons un aperçu étendu sur le diagnostic à base de modèles. Finalement, nous rappellerons les différentes approches d'analyse de diagnosticabilité.

II. Les défis de la conception des systèmes embarqués

La conception des systèmes embarqués est un domaine de l'informatique qui est resté dépendant de plusieurs autres domaines. En effet, les règles actuelles de la conception en informatique ne s'appliquent pas à la conception de systèmes embarqués telles qu'elles sont : elles doivent être enrichies par les modèles et les méthodes classiques du génie électrique. Le calcul et les logiciels (et donc l'informatique) font partie intégrante de systèmes embarqués. Ainsi, une connaissance conjointe de l'informatique et du génie électrique est souhaitable pour la conception des systèmes embarqués afin de réduire les erreurs de conception.

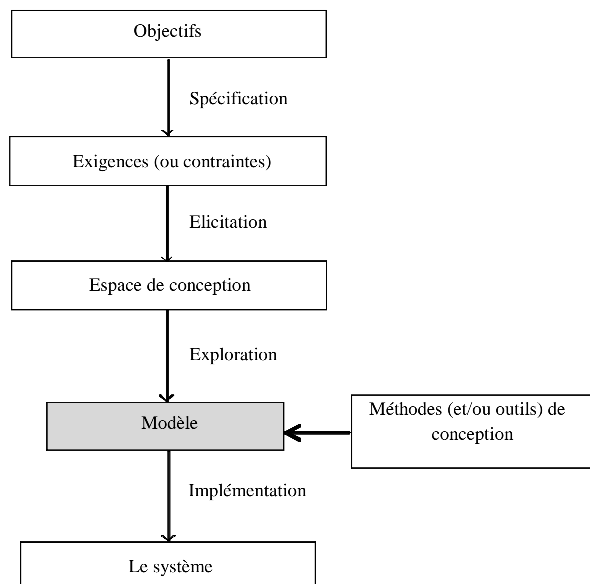

Figure 1. Processus de conception

La conception de systèmes, est le processus qui part des objectifs vers une spécification d'un ensemble d'exigences amenant à un espace de conception à explorer afin d'en déduire un modèle à partir duquel un système peut être implémenté plus ou moins automatiquement (Figure 1).

1. Conception matérielle et conception logicielle

La conception logicielle consiste à générer un programme et la conception matérielle consiste à générer une description à partir de laquelle une architecture matérielle peut être développée. Cependant, la conception matérielle est très dépendante des connaissances à priori sur l'architecture, donc assez dépendante du domaine d'application tels que : multimédia, contrôle/commande, calcul intensif, etc. [5]. Dans ces deux domaines, le processus de conception combine généralement le « *bottom-up* » (approche ascendante) et le « *top-down* » (approche descendante). Il existe une différence importante entre la conception des systèmes embarqués et les autres systèmes de calcul ; les systèmes embarqués comportent du calcul faisant l'objet de contraintes physiques. Ainsi, la conception des systèmes embarqués nécessite une approche qui intègre les méthodes de conception matérielle, les méthodes de conception logicielles et la théorie de contrôle d'une manière cohérente.

2. Différentes exigences des systèmes

Les concepteurs des systèmes traitent deux types d'exigences: Les exigences fonctionnelles qui sont les services, les fonctionnalités et les caractéristiques attendus, indépendamment de l'implémentation. Et les exigences extra-fonctionnelles sont notamment la performance, l'utilisation des ressources et la robustesse, ce qui caractérise la capacité d'offrir un minimum de fonctionnalités dans des circonstances loin des situations nominales [6].

III. Méthodologies de conception des systèmes embarqués

Historiquement, de nombreuses méthodologies de conception pour les systèmes embarqués étaient intimement liées aux langages permettant leur implémentation, qu'ils soient de programmation logicielle ou de modélisation matérielle. Ainsi, la conception des systèmes embarqués nécessite une approche qui subsume les deux méthodes.
Nous suivrons la taxonomie utilisée dans les travaux [7] [8], afin de présenter quelques méthodologies de conception des systèmes embarqués. Elle partage les méthodologies de conception des systèmes embarquée en deux classes différentes : l'ingénierie des systèmes et la conception basée modèle.

1. L'ingénierie des systèmes

L'ingénierie des systèmes est une approche qui concerne la conception et le contrôle de conception des systèmes complexes. L'ingénierie des systèmes se focalise sur la définition des exigences fonctionnelles, en les documentant, puis en poursuivant avec la synthèse de la conception et la validation du système**.** Aujourd'hui l'ingénierie des systèmes est soit « Best-effort » soit « systèmes critiques ».

a. Ingénierie Best-effort

L'ingénierie du Best-effort considère la conception comme un problème d'optimisation. Elle consiste à optimiser les performances du système et son coût. L'ingénierie des systèmes Best-effort vise l'utilisation efficace des ressources, par exemple l'optimisation du débit, de la gigue ou de la puissance. Elle est généralement utilisée pour des applications où une certaine dégradation, ou un déni de service temporaire est tolérable, comme dans les

télécommunications. Les exigences « strictes ou dures (*hard*)» du pire cas (*Worst-Case*) des systèmes critiques sont remplacées par les exigences «souples ou molles (*soft*)» de qualité de service (QoS : *Quality of Service*) [9].

b. Conception des systèmes critiques

La conception des systèmes critiques cherche à garantir la sécurité de ceux-ci à tout prix, même lorsque le système fonctionne dans des conditions extrêmes. Elle considère la conception comme un problème de satisfaction de contraintes. L'exemple typique pour cette approche est la sécurité dans l'avionique. La satisfaction des contraintes temps réel est garantie sur la base de l'ordonnancement statique et du WCET (*Worst-Case Execution Time*) [10], [11], [12]. La puissance de calcul nécessaire maximale est mise à disposition à tout moment. La fiabilité est atteinte notamment par l'utilisation de la redondance et par le déploiement de tout équipement de détection et traitement des défaillances.

2. La conception basée modèle

Les tendances récentes ont mis l'accent sur la combinaison des approches basées langages et celles basées synthèse (conception conjointe matérielle-logicielle ou *Hardware-Software co-design*) et sur l'acquisition, tôt dans le processus de conception, d'une indépendance maximale d'une plateforme d'implémentation spécifique. Ces nouvelles approches sont des approches de conception basée modèle (*Model-Based Design* ou MBD), qui soulignent la séparation du niveau de la conception du niveau de l'implémentation, et elles sont centrées sur la sémantique des descriptions des systèmes abstraits plutôt que sur des sémantiques de l'implémentation.

a. Quelques méthodologies de la conception « basée modèles »

Les langages synchrones, tels que Lustre et Esterel [13], incarnent une sémantique abstraite du matériel (synchronisme) au sein de différents types de structures logicielles (fonctionnelles). Les technologies d'implémentation sont disponibles pour plusieurs plateformes, y compris les architectures à déclenchement temporel (*Time Triggered Architectures* ou TTA) [14].
Une deuxième méthodologie, mise au point dans la communauté de l'automatisation de la conception repose sur SystemC [15] ; c'est un langage qui utilise aussi des sémantiques synchrones pour le matériel, mais permet l'introduction d'exécutions asynchrones et des mécanismes d'interaction à partir du logiciel écrit en C++. L'implémentation nécessite une séparation entre les composants qui doivent être réalisés sur le matériel et ceux qui doivent être programmés en logiciel. Différentes techniques d'exploration de l'espace de conception fournissent des orientations à fin de prendre ces décisions de partitionnement.
Un troisième type d'approches « basées modèles » (*Model Based Design ou MBD*) s'est développé autour d'une classe des langages populaires comme Matlab/Simulink, dont la sémantique est définie sur le plan opérationnel à travers son moteur de simulation. Cette catégorie de langages de conception « basés modèles » est dotée de fonctions mathématiques et de routines optimisées pour la conception et l'analyse des stratégies de contrôle par la simulation hors ligne. Ces outils peuvent être facilement couplés avec le matériel en temps réel pour une simulation de type « *hardware-in-the-loop* [2] » afin d'effectuer des tests des effets dynamiques sur le système plus rapidement et plus efficacement qu'avec les

[2] Hardware in the loop ou HIL : est une technique de simulation utilisée dans le développement et le test de systèmes embarqués temps réel complexes.

méthodologies de conception classiques. Cela peut aider à éliminer des erreurs en début de conception, produisant un système de contrôle plus robuste avec moins d'itérations dans le cycle de développement [16]. Ainsi, le MBD permet de réduire le coût de conception et le temps de développement.
Plus récemment, un autre type d'approche est fondé sur les langages de modélisation nouveaux tels qu'UML [17] et AADL [18], tentent d'être plus générique dans le choix de la sémantique et d'apporter ainsi des extensions dans deux directions : l'indépendance d'un langage de programmation particulier, et la focalisation sur l'architecture du système comme un moyen d'organiser le calcul, la communication, et les contraintes.
Parmi les avantages remarquables du MBD nous citons l'environnement de conception commun, qui facilite la communication générale, l'analyse des données et la vérification du système entre les groupes de développement, dans le processus de conception, tout en respectant le cycle de développement. Ceci facilite aux ingénieurs la tâche de localiser et de corriger les erreurs pendant la conception. Egalement, ceci facilite la réutilisation de la conception et la mise à niveau avec des fonctionnalités étendues.

b. La conception itérative

Les approches de conception itératives sont nombreuses et ont évolué au cours des années selon le niveau des connaissances sur le système que le concepteur possède à priori. En effet, souvent la conception se passe ni strictement « top-down » (des exigences à l'implémentation), ni strictement « bottom-up », mais d'une manière moins dirigée, en itérant la modélisation, l'analyse de modèle et la transformation de modèles (Figure 2).

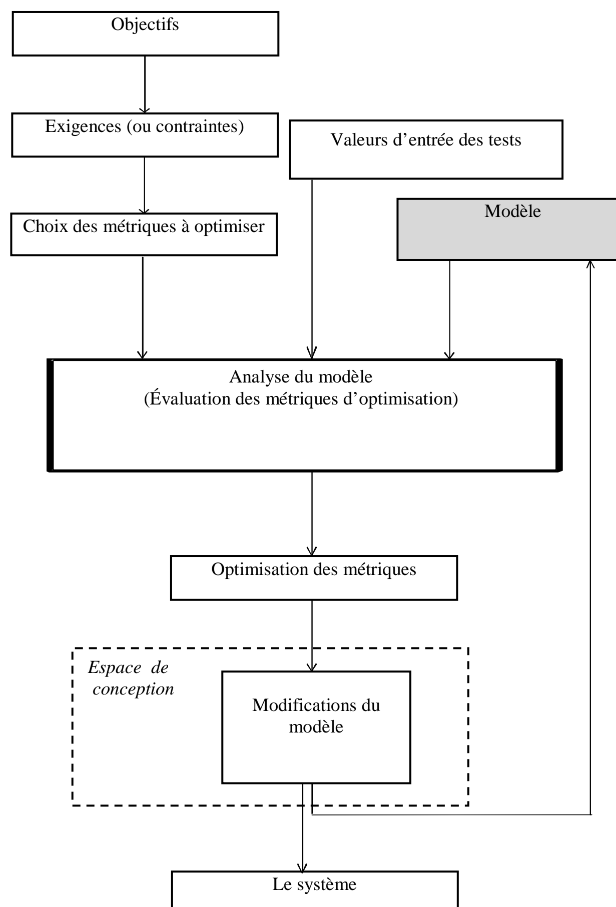

Figure 2. Processus itératif de conception

Dans les années 90, sont apparues les méthodes d'Adéquation Algorithme Architecture ou AAA [19] [20], orientées vers la conception d'architectures, ainsi que les méthodes de transformation de modèles du MBD. Depuis le début des années 2000, il y a également une tendance vers la conception à base de plateforme (Platform-Based Design ou PBD) [21] (Figure 3).

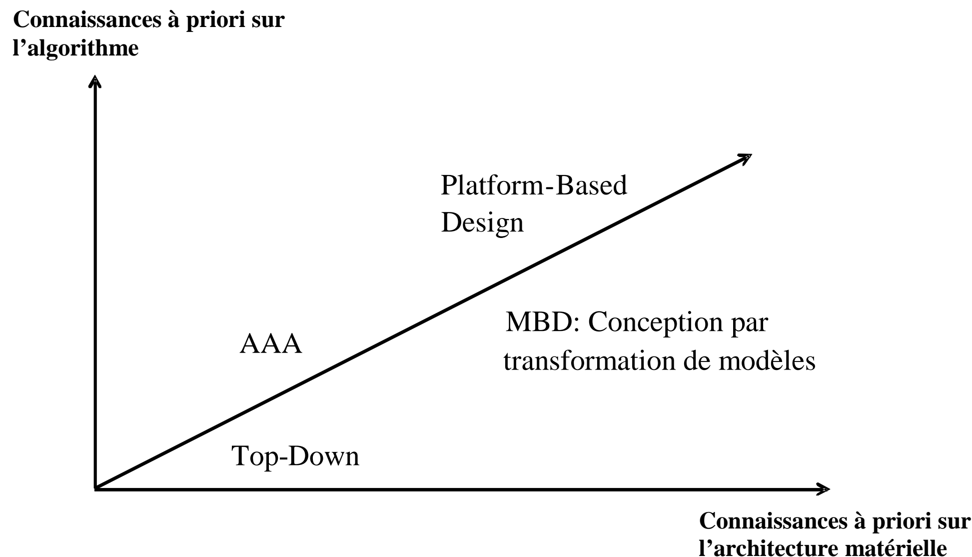

Figure 3. Evolution dans le temps de quelques méthodologies de conception

Dans cette thèse, nous focalisons nos travaux de recherche sur la conception des systèmes embarqués pour le domaine de l'automobile. Ces systèmes sont connus pour leurs applications embarquées temps-réel de complexité croissante et pour la nature distribuée des ressources (capteur / actionneur, mémoire) [22]. Nous nous intéressons ci-après à la conception itérative de cette catégorie d'architectures et plus précisément à l'étape « analyse des modèles ».

IV. Analyse de modèles pour la conception d'une architecture embarquée sûre

Le processus de conception des systèmes embarqués vise toujours la maximisation de la valeur du système et la minimisation de son coût. Ainsi, la valeur et le coût d'un système sont toujours corrélés et peuvent être mesurés à l'aide de quelques critères. Le processus de conception optimal est : « Créer des produits flexibles en un temps court avec des coûts de conception et de développement prédictibles ». Les objectifs finaux doivent converger au maximum avec la conception optimale et ce en ajustant et en variant les critères cités. Un processus de conception itératif prend en considération un « espace de conception » et génère des modèles qui participeront à l'optimisation des paramètres (ou métriques) fournis par l'équipe qui établit les spécifications. Afin d'explorer l'espace de conception par l'analyse de modèles, le concepteur fournit des fonctions d'évaluation, adaptées à l'estimation des critères d'optimisation déterminés à partir des objectifs finaux. L'exploration de l'espace de conception peut concerner le système entier, un sous-système, ou même une seule entité du système.

1. Exploration de l’espace de conception par l’analyse de modèles

L’exploration de l’espace de conception sert à :

- optimiser une architecture en respectant un ou plusieurs critères,
- à analyser en respectant les paramètres de conception,
- à analyser, évaluer et comparer les différentes architectures possibles entre elles,
- à découvrir des paramètres de conception qui n'ont pas été fixés. Par exemple le nombre de processeurs et leur vitesse respective
- et à la découverte des dépendances entre les attributs et estimer leur corrélation.

Les données nécessaires à l'exploration de l’espace de conception d'architecture comprennent :

- les exigences et les contraintes, à laquelle la conception de l'architecture doit se conformer, et à partir desquels sont définis des paramètres pour guider la conception de l'architecture
- Principes et méthodes heuristiques qui orientent la conception de l'architecture
- le modèle servant de base à l'analyse
- les techniques et les fonctions d'analyse grâce auxquelles les architectures alternatives peuvent être analysées et comparées

Une exploration exhaustive de l'espace de conception exige l'évaluation de chacune des solutions dans l'espace de conception (Figure 4) [23]. Parce que les ressources de calcul et le temps requis pour chaque évaluation et le grand nombre de possibilités dans un espace de conception typique, une telle exploration exhaustive est généralement infaisable. Les outils automatiques d'exploration de l'espace de conception doivent employer des méthodes heuristiques efficaces pour explorer l'espace de conception.

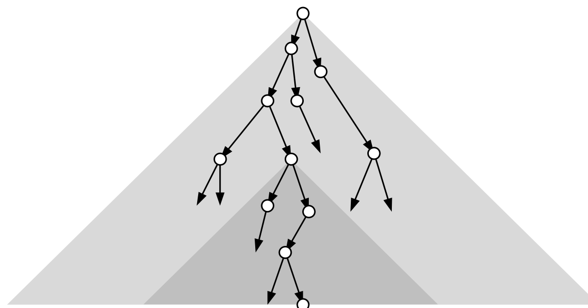

Figure 4. Illustration de l'exploration de l'espace de conception

Les flèches indiquent les choix de conception possible, les cercles désignent les choix de conception qui seront ensuite évaluées. Chaque choix de conception délimite le futur espace de conception, comme indiqué par les deux triangles gris différents dans le fond de la figure. Plusieurs options peuvent être évaluées en parallèle.

Les critères doivent être définis sur la base des exigences qui s'appliquent au système en cours de son développement, par exemple fondées sur des contraintes temps réel. Lors de la définition des critères et des métriques pour une évaluation spécifique à accomplir, il est nécessaire d'éviter une seule évaluation des options architecturales. Il est nécessaire d'utiliser un espace de critères approprié, en s'assurant qu'une qualité globale adéquate de l'architecture est maintenue. Un point de départ possible pour le choix des critères sont les normes. En effet,

il existe des normes génériques et des normes spécifiques au domaine. Par exemple la norme ISO / IEC 9126 est considérée comme générique et définit un cadre pour l'évaluation de la qualité des logiciels produits. Ceci prend en considération la fonctionnalité, la convivialité, l'efficacité, la maintenabilité et la portabilité.

Le critère convivialité n'existerait pas pour une norme spécifique telle que la norme ISO 26262 (basée sur la CEI 61508) adaptée à l'industrie automobile.

Notons que la sûreté de fonctionnement et la sécurité n'ont pas été citées dans la norme, mais sont deux métriques très utilisées pour l'analyse et l'évaluation des modèles. La sûreté de fonctionnement (SdF), est « la propriété qui permet aux utilisateurs du système de placer une confiance justifiée dans le service qu'il leur délivre » [24]. Cette confiance se justifie à travers une analyse qualitative et quantitative des différentes propriétés du système, à savoir les FMDS (fiabilité, maintenabilité, disponibilité, sécurité). Notons que ces propriétés sont liées et que les valeurs des indicateurs de l'une (s'ils existent), aident bien à estimer les autres. Notons aussi que la SdF doit être prise en compte tout le long du cycle de vie d'un produit.

2. Techniques d'analyse de modèles pour garantir la sûreté de fonctionnement

Les méthodes utilisées pour l'analyse et l'évaluation de la sûreté de fonctionnement sont nombreuses et dépendent de plusieurs facteurs tels que la nature du système à concevoir. Nous présenterons les méthodes les plus utilisées dans le domaine des systèmes embarqués. Ces méthodes ont été résumées, avec une vingtaine d'autres méthodes, dans la fiche de méthodes, que propose le groupe GTR M2OS (Management, Méthodes et Outils Standards) de l'IMdR (INSTITUT POUR LA MAÎTRISE DES RISQUES –Sûreté de fonctionnement-Management-Cindyniques) [25].

a. Arbres de défaillance (FTA)

L'arbre de défaillances est une méthode de type déductif, utilisée pour l'analyse de la fiabilité des systèmes. Elle est basée sur une représentation graphique partant de la cause et allant aux effets. Un arbre de défaillance est généralement présenté de haut vers le bas. La ligne la plus haute comporte un évènement, appelé Evénement Redouté (ER), dont il faut chercher à décrire comment il peut se produire. Ensuite, chaque ligne présente la ou les combinaison(s) susceptibles de provoquer l'évènement de la ligne supérieure auquel elles sont rattachées. Les relations sont représentées par des opérateurs logiques : porte « OU » et porte « ET » (Figure 5). Cette méthode fait l'objet de la norme internationale CEI 1025 [26].

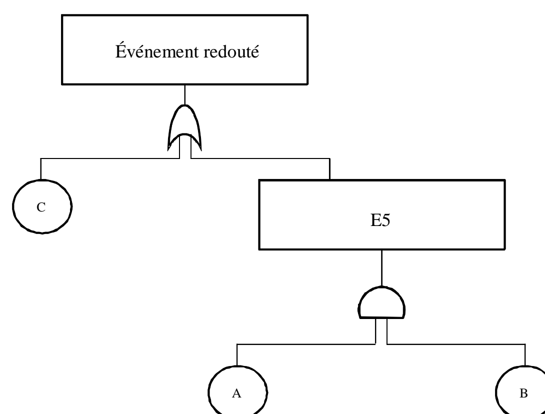

Figure 5. Arbre de défaillances

L’analyse par arbre des défaillances est utilisée dans de nombreux domaines tels que l’aéronautique, l’automobile, le nucléaire, etc. Le principal avantage de cette analyse est qu’elle permet de considérer des combinaisons d’évènements pouvant conduire à un événement redouté. Cependant, la qualité des résultats dépend de l’expérience et du savoir faire de celui qui effectue l’analyse. Lorsque le nombre de combinaisons d’événements dépasse quelques dizaines d’unités, la méthode requiert un programme informatique pour calculer l’occurrence de l’événement indésirable étudié et rechercher les coupes minimales.

b. L'Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité (AMDEC)

L’AMDEC est une méthode d’analyse inductive ayant pour objectifs l’identification des défaillances dont les conséquences peuvent affecter le fonctionnement d’un système. Elle consiste aussi à hiérarchiser les défaillances selon leur niveau de criticité afin de les maitriser. Cette méthode est introduite aux Etats Unis dans les années 1950 dans le domaine des armes nucléaires sous le nom de FMECA (*Failure Modes, Effects and Criticality Analysis*). L’AMDE (AMDE est la traduction de FMEA (*Failure Modes, Effects and Criticality*) [27]) est la version non quantifiée de l’AMDEC. C’est une procédure d'analyse des modes de défaillance potentiels au sein d'un système pour les classer par la sévérité et la probabilité des défaillances. L'AMDEC ajoute à l'AMDE une évaluation de la criticité des modes de défaillance permettant leur hiérarchisation. Le principe de l’AMDEC est d’identifier pour chaque mode de défaillance sa (ses) cause(s), son indice de fréquence (classe d'occurrence), ses effets, son indice de gravité (classe de sévérité), les mesures mises en place pour détecter la défaillance, et son indice de détection (classe de probabilité de détection). Ensuite, le produit : **C= (indice de fréquence) x (indice de gravité) x (indice de détection)** permet d’obtenir la criticité (Figure 6).

Indice	Critère: Occurrence
1	Moins d’une fois par an
2	Moins d’une fois par mois
3	Moins d’une fois par semaine
4	Plus d’une fois par semaine

Indice	Critère: Gravité
1	Temps d’arrêt inférieur à 12 h
2	Temps d’arrêt inférieur à 24 h
3	Temps d’arrêt inférieur à 1 semaine
4	Temps d’arrêt supérieur à 1 semaine

Indice	Critère: Détection
1	Détection efficace permettant une action préventive
2	Il y a un risque que la détection ne soit pas efficace
3	Le moyen de détection n’est pas fiable
4	Il n’y a aucun moyen de détection

Figure 6. Criticité= Occurrence*Gravité*Détection

L'AMDEC est très utilisée dans le secteur de l'automobile, de l'aéronautique, du ferroviaire et des équipements médicaux, tout au long du processus de conception, de développement et d'exploitation.

Cependant, cette méthode ne permet pas d'avoir une vision simultanée des pannes possibles et de leurs conséquences (deux pannes survenant en même temps sur deux sous-systèmes et pouvant avoir une conséquence sur le système global). Dans ce cas, des analyses complémentaires sont nécessaires, notamment par arbres de défaillance.

c. Analyse Préliminaire des risques

L'Analyse Préliminaire des Risques (APR) a été développée aux Etats-Unis au début des années 1960 dans les domaines aéronautique et militaire. L'APR est une méthode d'identification et d'évaluation des risques en début de conception d'un système. Elle se base sur l'analyse fonctionnelle et l'utilisation de listes de risques génériques. Elle permet de définir des critères de conception à appliquer pour permettre de réduire les risques identifiés. Le plus souvent, la méthode se présente sous forme d'un tableau d'APR. [28].
L'avantage de cette méthode est qu'elle permet d'estimer les dangers inévitables d'un système avec le niveau de détail requis. Cependant, ceci dépend de l'expérience acquise avec des produits similaires au système à analyser.

d. Vérification formelle

La vérification formelle est utilisée pour vérifier du code informatique ou aussi du code de modélisation de circuits électroniques, en utilisant des méthodes formelles (c'est-à-dire vérifiant des relations mathématiques), afin de démontrer leur validité par rapport à une certaine spécification (un ensemble de documents qui - par des textes et des diagrammes - décrit de manière formelle et exhaustive le produit informatique à réaliser). La vérification est basée sur plusieurs méthodes, souvent complémentaires [29]. Il existe plusieurs catégories de méthodes formelles: la vérification de modèle (ou *model checking*) [30], [31], l'analyse statique par interprétation abstraite, la preuve automatique de théorème et les assistants de preuve. Les méthodes de vérification formelle, sont généralement coûteuses en ressources et réservées aux logiciels assez critiques.

Nous remarquons que le concept de la diagnosticabilité (*Diagnosability* en anglais) est assez proche des notions défendues par les concepts FMDS (fiabilité, maintenabilité, disponibilité, sécurité) de la sûreté de fonctionnement. Nous estimons essentiel pour la conception des systèmes embarqués, notamment dans la phase de l'analyse des modèles d'inclure l'analyse de la diagnosticabilité. En effet, l'analyse de diagnosticabilité permet d'estimer les éléments FMDS et par conséquent contribue indirectement à l'estimation de la SdF : Un système sûr doit être aussi diagnosticable.

V. Le diagnostic basé modèle

Le diagnostic est l'identification des causes d'un trouble fonctionnel, par l'observation du système [31]. Les solutions au problème de diagnostic de pannes varient des approches associatives aux approches basées modèle. Le diagnostic basé modèle (*MBD : Model-Based Diagnosis*) s'appuie sur l'utilisation d'un modèle du système à diagnostiquer. Le diagnostic

basé-modèle est convenable, de point de vue méthodologie et pratique, à l'intégration de l'analyse de la diagnosticabilité dans la conception de systèmes électroniques [32]. Ainsi, nous limitons notre intérêt au diagnostic basé-modèle.

1. Les approches alternatives du diagnostic basé modèle

Hamscher, W. et al dans [33] affirment qu'il est intéressant de considérer les alternatives de l'approche basée modèle à la fois comme un moyen de la mettre en contexte et d'établir les conditions appropriées pour son utilisation ;

- Une approche qui remonte aux débuts du diagnostic consiste à utiliser les programmes de test utilisés sur les dispositifs électroniques à la fin du processus de fabrication, afin d'assurer qu'ils satisfont les exigences.
- Une deuxième technique consiste à construire un « dictionnaire de fautes» en utilisant la simulation et une liste des types de fautes prévues. L'idée ici est de simuler le comportement du dispositif dans chacune des situations dans laquelle chacun de ses composants risque de tomber en panne. Ainsi, chaque simulation aboutit à une description de la manière dont le dispositif entier se comportera quand un composant spécifique tombe en panne dans un contexte spécifique. Le résultat est une liste de paires faute/ symptôme. La liste est ensuite organisée par symptôme, en fournissant un dictionnaire qui indexe à partir des symptômes observés une ou plusieurs fautes sous-jacentes susceptibles de provoquer une panne.
- Troisièmement, nous pouvons développer des programmes de diagnostic en se servant de l'expérience des experts, de la même manière utilisée pour construire des systèmes experts à base de règles. Les arbres de décision constituent une approche de collecte de connaissances pour le diagnostic offrant un moyen d'organiser une série de questions qui conduit par processus d'élimination au composant défectueux.

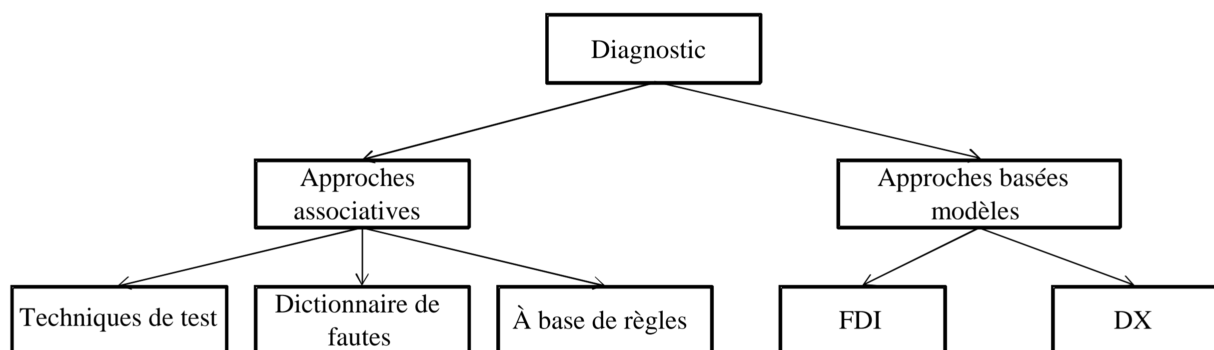

Figure 7. Les différentes approches du diagnostic

FDI : ***Fault Detection and Isolation*** **(Approches de contrôle automatique)**
DX : ***Diagnosis*** **(Approches d'Intelligence Artificielle)**

Les deux dernières et particulièrement les systèmes experts à base de règles, sont largement utilisés dans de nombreux domaines [34]. Ces approches aident au diagnostic de fautes éventuelles en réutilisant l'expérience passée. Toutefois, l'une des plus grandes limites de ces approches est que l'expérience peut être longue à obtenir, parfois plus longue que le cycle de vie du système. En outre, ces méthodes ne sont pas les plus adéquates à appliquer à un nouveau système, ou à un système mis à jour. En effet, toute modification du système rend la totalité ou l'essentiel des connaissances des experts obsolètes. En réponse aux limites des approches associatives de diagnostic, celles basées sur des modèles ont été développées (Figure 7). Elles reposent sur une description du comportement du système dans un langage

formel, appelé le « modèle » prévu par le concepteur et décrivant le comportement nominal du système. Le modèle doit être comparé au comportement réel du système afin de détecter les fautes éventuelles (Figure 8).

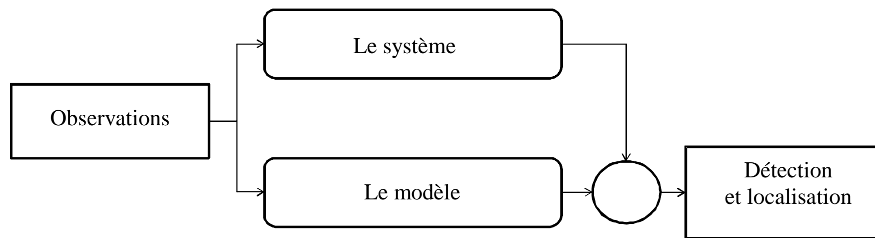

Figure 8. Le diagnostic basé modèles.

Nous pouvons obtenir un diagnostic plus précis, si le modèle décrit le comportement du système avec des fautes prédéfinies, ou lorsque le modèle décrit la structure du système (à savoir les composants qui prennent part dans le comportement du système).

2. Les différentes communautés du diagnostic basé modèle

Le diagnostic basé modèle a été étudié par deux communautés scientifiques différentes :

- l'intelligence artificielle, aussi connu comme l'approche DX *(Diagnosis)* [33] [35],
- et en automatique, également appelé l'approche FDI *(Fault Detection and Isolation)* [36] [37] [38].

Puig V et al expliquent dans [39] la différence entre les deux communautés : l'approche DX repose sur une théorie de diagnostic des systèmes statiques. D'un point de vue logique, la détection des fautes est réalisée à partir d'une vérification de consistance, organisée autour du concept de conflit. Dans cette approche, la localisation ou l'isolation de pannes est automatiquement dérivée de la phase de détection de conflits. D'autre part, l'approche FDI décompose le diagnostic en deux tâches distinctes : la détection et l'isolation des fautes. Ces deux tâches sont basées sur la génération et l'évaluation d'un ensemble de relations de redondance analytique obtenues hors-ligne à partir de modèles de composants élémentaires du système. Les deux approches sont davantage comparées et discutées dans [40].
Il n'existe aucun intérêt dans la modélisation d'un système et l'exécution d'un programme de diagnostic, si cela génère toujours des diagnostics qui sont si indéfinis qu'aucune décision ne peut en être prise. La vérification qu'un modèle avec ses observables connus à l'avance permette la détection et la discrimination des défaillances possibles est en d'autres termes la vérification de la diagnosticabilité [41]. Celle-ci est effectuée principalement pendant la phase de conception du système et indique si les observables attendues du système (ajouter des observables dans un système nécessite souvent l'ajout de capteurs) sont suffisants pour identifier dans un délai déterminé toute anomalie de fonctionnement.

3. Les différentes approches de modélisation des systèmes à diagnostiquer

Nous présentons dans cette section les critères majeurs permettant de catégoriser les systèmes à diagnostiquer par leur nature, ce qui aide à déterminer l'approche appropriée pour leur modélisation (définition des fautes et observabilité du système) et l'analyse de leur diagnosticabilité par la suite. Les communautés DX et FDI, classent les systèmes en deux catégories : les systèmes continus et les systèmes discrets. Cependant, il y a une tendance récente qui traite des systèmes hybrides. Dans cette section nous présentons brièvement ces

trois catégories, et nous nous focaliserons par la suite sur les systèmes discrets (domaine d'application de ma thèse).

a. Les systèmes continus

Pour les systèmes à comportement continu, l'approche de modélisation utilisée est celle des modèles à base d'états. Les fautes et les observations sont représentées par des variables et associées à un ensemble d'états.
L'observabilité dans un tel système est définie par la détermination d'un sous-ensemble de variables, appelées variables observables ; dont la valeur peut être observée. A un moment donné, le système est caractérisé par l'ensemble des valeurs de ses observables à cet instant.

b. Les systèmes discrets (ou systèmes à événements discrets)

Les systèmes discrets (ou systèmes à événements discrets) sont des systèmes qui mettent en jeu des informations qui ne sont prises en compte qu'à des moments précis. En général ces instants sont espacés d'une durée constante appelée période d'échantillonnage. Les systèmes discrets sont représentés par des modèles à base d'événements. Les fautes et les observations sont représentées par des événements discrets entre les états. L'observabilité dans ces systèmes est traitée par le partitionnement de tous les événements dans le modèle en deux sous-ensembles : Les événements observables (correspondent généralement à des capteurs installés dans le système) et événements non-observables qui comportent les événements « fautes ». Ainsi, ce que nous observons dans un système à événements discrets sont des séquences d'événements observables.

c. Les systèmes hybrides

Dans les systèmes hybrides, les aspects continu et discret coexistent dans la modélisation. Cette approche consiste à modéliser le système par un ensemble d'états et de transitions comme dans un système à événements discrets mais les états du système, sont modélisés par une approche « continu » [42].

4. Les différents aspects et contextes des systèmes à diagnostiquer

En plus de la nature continue, discrète ou hybride d'un système, il existe deux autres facteurs importants à prendre en considération, lors de la modélisation ; la structure du diagnostic : centralisé ou distribué et le contexte de son application : en ligne ou hors-ligne.

a. Diagnostic centralisé et distribué

La structuration des fonctions de diagnostic peut influencer la phase de modélisation. Il existe trois structures de diagnostic différentes : centralisée, décentralisée, et distribuée (Figure 9). Dans les structures centralisée et décentralisée, le diagnostiqueur a besoin uniquement d'un modèle global du système. Nous avons ainsi une vue globale des paramètres pertinents du système. L'inconvénient de cette approche est la difficulté de représenter dans un modèle unique le fonctionnement du système entier. Par contre, dans le diagnostic distribué, les diagnostiqueurs ont besoin d'un modèle local de chaque composant. Cette approche peut devenir très coûteuse dans le cas des grands systèmes complexes.

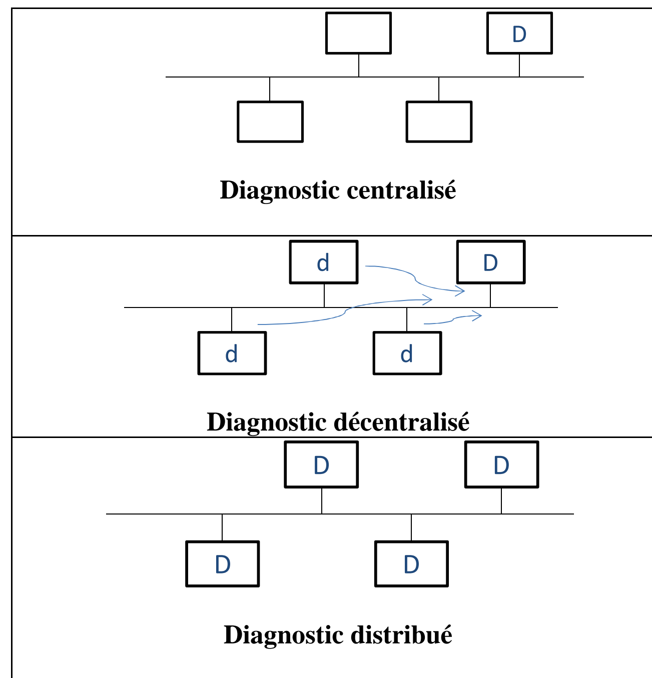

Figure 9. Les différentes structures de diagnostic

D : Diagnostiqueur
d : Sous-fonction de D

b. Les contextes en ligne et hors-ligne

Concernant le diagnostic en ligne, le système est à diagnostiquer dans son mode opérationnel. Par conséquent, l'état de fonctionnement du système est en évolution continue et certains changements ne peuvent être évités. Quant au diagnostic hors ligne, le système est à diagnostiquer en état d'arrêt. La défaillance des composants du système ne changera pas et les événements sont contrôlables. Les différences majeures entre le diagnostic en ligne et le diagnostic hors-ligne sont résumées dans le Tableau 1. Feng Lin dans [43] a étudié ces différences et a étudié également la diagnosticabilité à la fois en ligne et hors-ligne. Il a proposé de réduire le problème de la vérification de la diagnosticabilité en ligne à un problème d'atteignabilité.

Tableau 1. Diagnostic en ligne et diagnostic hors ligne

	Diagnostic en ligne	**diagnostic hors ligne**
Etat	Change	Ne change pas
Evénement	Ne sont pas tous contrôlables	tous contrôlables
Observation	Réservés aux sorties	Non réservés aux sorties
Séquence de test	Contrainte	Non contrainte

VI. Les différentes approches d'analyse de diagnosticabilité

La définition de la diagnosticabilité dépend de la définition de la nature du système étudié. Ainsi, il est difficile de définir la diagnosticabilité d'une manière unifiée. Cette section s'intéresse aux différentes approches d'analyse de diagnosticabilité qui existent dans la

littérature ; notamment les approches basées événements, les approches basées états et les approches hybrides. Dans [44], les différences entre les concepts utilisés dans les systèmes continus et les systèmes à événements discrets sont étudiées en détail.

1. Les approches basées états pour l'analyse de la diagnosticabilité

Les approches d'analyse de diagnosticabilité « basées états » sont applicables aux systèmes continus. Dans les deux communautés FDI et DX, seul le cas des systèmes centralisés a été traité avec les systèmes continus. En effet, pour les systèmes continus, la communauté FDI a une méthode bien connue appelée « l'Espace de parité », qui consiste à utiliser le modèle du système afin d'établir un ensemble de relations redondantes analytiques [45] [46]. Cependant, dans la communauté DX, il existe peu d'approches de diagnosticabilité applicables aux systèmes continus. Dans [47], l'analyse de la diagnosticabilité consiste à identifier les conditions de fonctionnement dans lesquelles les différentes fautes sont discriminables.

2. Les approches basées événements pour l'analyse de la diagnosticabilité

Les approches d'analyse de diagnosticabilité « basées événements » sont applicables aux systèmes à événements discrets. La plupart des travaux d'analyse de diagnosticabilité réalisés sur les systèmes à événements discrets utilisent les automates finis simples comme modèles de représentation. Dans ces travaux, nous distinguons :

- Les approches centralisées, telles que l'approche du «diagnostiqueur» de M Sampath et al [48], l'approche de l'algorithme polynomial de test de diagnosticabilité [49] et l'approche des algorithmes polynomiaux basés sur la construction d'automates non-déterministes [50].
- Les approches décentralisées. Parmi les travaux réalisés dans cette catégorie, l'approche de l'analyse de la diagnosticabilité des systèmes à événements discrets avec une structure modulaire [51] peut être considérée comme une généralisation de l'algorithme original d'analyse diagnosticabilité proposé par M. Sampath [48], appliquée dans le cas d'une architecture modulaire distribuée. D'autres ouvrages se sont inspiré aussi de l'approche de M. Sampath [48] et l'ont adaptée aux systèmes à événements discrets dans un contexte distribué [52] [53].

Il existe aussi des travaux dans ce domaine utilisant d'autres types de modèles à base d'événements discrets tels que les automates temporisés [54] et les réseaux de Petri [55]. Enfin, la diagnosticabilité des systèmes à événements discrets a été étudiée aussi à l'aide des sémantiques d'ordre partiel basées essentiellement sur la technique d'exécution des réseaux de Petri [56], [57],[58].

La propriété de la diagnosticabilité dans les systèmes à événements discrets peut être considérée comme n'importe quelle propriété formelle qu'on souhaite vérifier. Par conséquent, la vérification de la diagnosticabilité a également attiré les chercheurs intéressés par ce type de problèmes, y compris ceux qui travaillent sur le model-checking [59] et sur la modélisation des systèmes à l'aide des langages algébriques [60] ou des formules SAT (SATisfability) [61].

3. Les approches hybrides pour l'analyse de la diagnosticabilité

L'analyse de la diagnosticabilité des systèmes hybrides est une approche récente. Seul le cas des systèmes centralisés a été traité.

Dans la littérature, il existe peu de travaux sur l'étude de la diagnosticabilité des systèmes hybrides. Par exemple, dans l'approche [62], les systèmes hybrides impliquent à la fois les dynamiques continues et discrètes. Les états du modèle d'un système hybride reflètent l'état normal et l'état de panne des composants du système. Les fautes sont modélisées comme des changements d'états discrets ou continus.

VII. Synthèse des différentes approches d'analyse de diagnosticabilité

Selon l'état de l'art présenté, nous pouvons classer les travaux réalisés sur l'analyse de la diagnosticabilité comme suit : les deux communautés FDI et DX d'analyse de diagnosticabilité s'intéressent aux systèmes continus, aux systèmes à événements discrets et aux systèmes hybrides. Ces deux communautés s'intéressent aussi aux systèmes à structure centralisée et distribués quand ils sont « basés événements discrets », et aux systèmes centralisés quand ils sont continus ou hybrides (Figure 10). La plupart des travaux réalisés sur l'analyse de la diagnosticabilité, s'inscrivent dans le cadre de la structure centralisée du diagnostic.

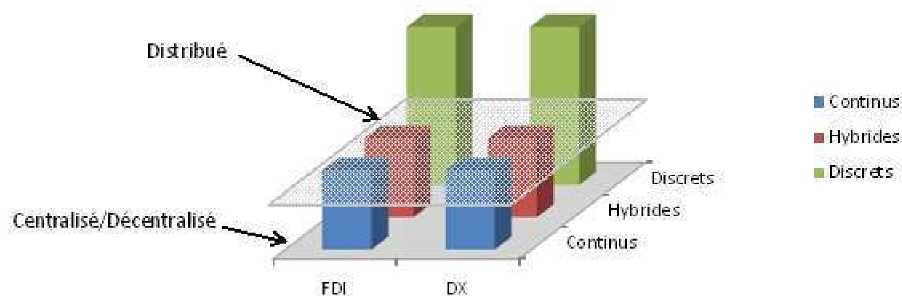

Figure 10. Synthèse sur les approches d'analyse de diagnosticabilité

Cependant, aucune approche d'analyse de diagnosticabilité ne s'est intéressée à la notion multi-modèle pour la description du système à analyser. Par exemple pour les architectures informatiques matérielles, les approches les plus utilisées pour l'analyse de la diagnosticabilité sont les approches « basées événements ».

VIII. Conclusion

Dans ce premier chapitre, nous avons commencé par présenter un état de l'art sur la conception d'un système embarqué informatisé, avec une focalisation sur la conception de l'architecture matérielle. Ensuite, nous avons présenté quelques approches d'analyse de diagnosticabilité « basées événement », qui considèrent que les systèmes basés événement (Event-Based Systems EBS) peuvent être modélisés par des événements discrets. Cependant, dans le cas d'un système embarqué, la modélisation des événements discrets ne fournit aucune description de l'interaction des fonctions avec l'architecture matérielle (dans le modèle formel représentant le système). Nous souhaitons introduire assez tôt dans ce document les aspects liés la diagnosticabilité d'une architecture électronique [63], afin d'amener le lecteur rapidement au cœur du sujet traité dans cette thèse.
Dans la partie suivante, nous essayons de combler cette absence d'information sur l'interaction fonctions-architecture. Nous ne proposons pas une méthode d'analyse de diagnosticabilité de l'architecture à partir d'un modèle formel décrivant la structure de

l’architecture car chaque instanciation d’un modèle d’architecture possède ses propres propriétés qui la distinguent d’autres instanciations possibles, nous définissons plutôt des indicateurs sur l’interaction fonction-architecture (comportement-structure) du système, qui accompagneront l’analyse de sa diagnosticabilité suivant les approches « basées événements discrets ».

Analyse de la diagnosticabilité entre architecture matérielle et fonction

I. Introduction

Dans ce chapitre, afin de présenter l'analyse de diagnosticabilité d'un système à événements discrets, nous allons illustrer les différentes étapes par un exemple. Cet exemple servira à prouver que les approches d'analyse de diagnosticabilité basées sur une modélisation uniquement au niveau fonctionnel ne sont pas suffisantes car elles omettent de prendre en compte les contraintes de l'architecture matérielle. Nous présenterons ainsi, notre approche qui prend en considération l'interaction architecture-fonction du système à concevoir et dont on souhaite vérifier sa diagnosticabilité au cours de sa conception.

II. Illustration de la diagnosticabilité des systèmes à événements discrets

Comme nous nous intéressons à des systèmes à événements discrets, nous nous limiterons à la seule approche d'analyse de diagnosticabilité du « diagnostiqueur » de Meera Sampath. Cette approche sert toujours comme référence à des travaux dans le domaine de la diagnosticabilité des systèmes à événements discrets. L'exemple présenté par cette approche est un système de chauffage, comprenant ventilation et air conditionné appelé HVAC (pour Heating, Ventilating, Air conditioning) (Figure 11).
L'étude se limite au fonctionnement des trois parties suivantes du système : la pompe, la valve et le contrôleur qu'on appellera PVC (pour Pompe, Valve, Contrôleur) (Figure 12).
L'idée est de modéliser le système dont on souhaite vérifier la diagnosticabilité à l'aide d'un automate fini ; les événements sont représentés par des arcs et les états sont représentés par des nœuds. Les événements sont partagés en deux catégories :

- événements observables
- et événements non-observables.

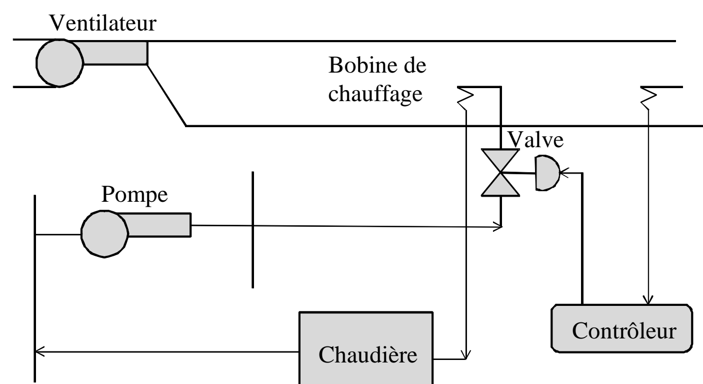

Figure 11. L'exemple du système HVAC (*Heating, Ventilating, Air Conditioning*) : Système de Chauffage, Ventilation et Air Conditionné

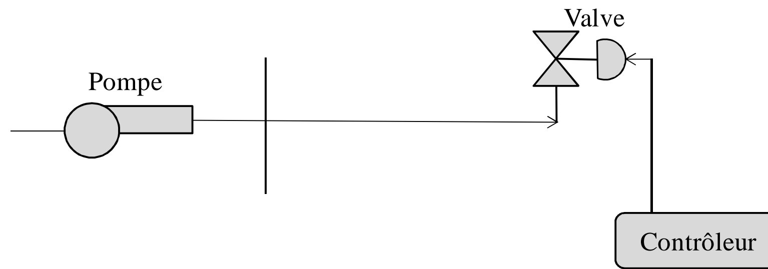

Figure 12. Sous-système de HVAC : Pompe, Valve et Contrôleur (PVC)

Si l'événement est observable, il faut mettre son nom sur l'arc qui le représente. Si l'événement est non-observable, il faut indiquer « no » sur l'arc qui le représente. Un événement est observable s'il existe au moins un composant du système (capteur ou autre) qui permet de l'observer sinon il est considéré comme non-observable. Tout événement du système considéré comme « faute », doit être représenté sur l'automate par un arc ayant comme étiquette la description de la faute et est considéré comme non observable.
Pour l'exemple du sous-système de HVAC comportant les composants suivants : la pompe, la valve et le contrôleur. Les événements « fautes » du comportement du composant valve sont :

- « stuck closed » (bloquée fermée) qui mène à l'état « SC »
- Et « stuck open » (bloquée ouverte) qui mène à l'état « SO » (Figure 13)

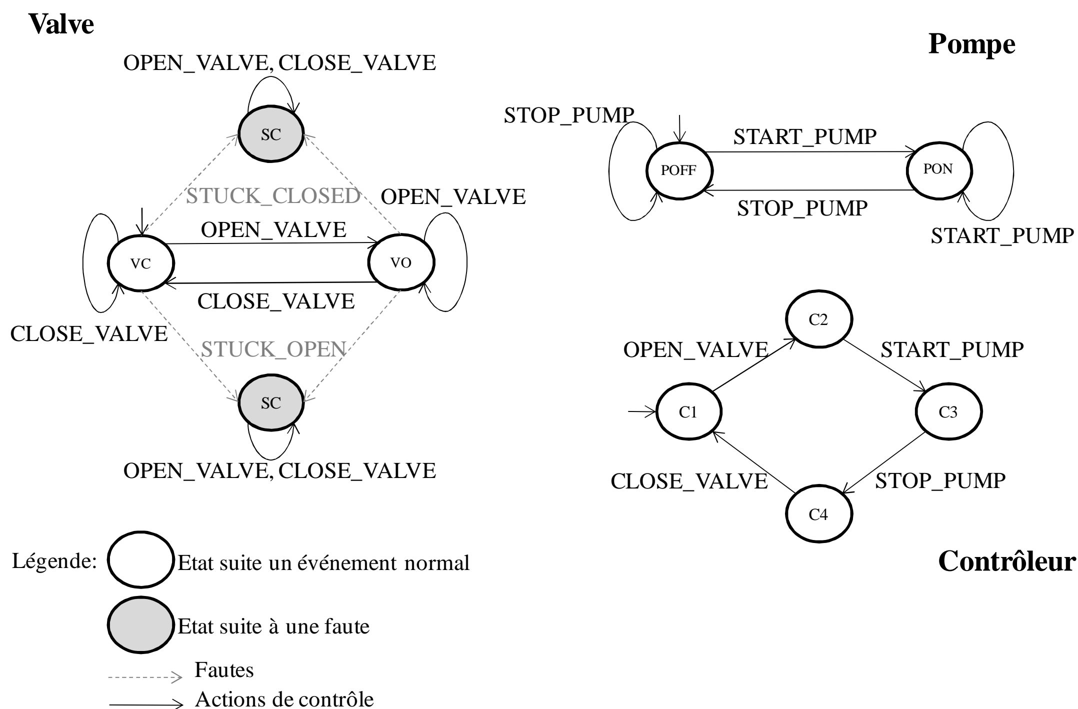

Figure 13. Modèle de comportement des composants

Ensuite, toute déviation de ce comportement nominal fera partie du mode dégradé, qui à son tour doit être modélisé aussi (Figure 14).

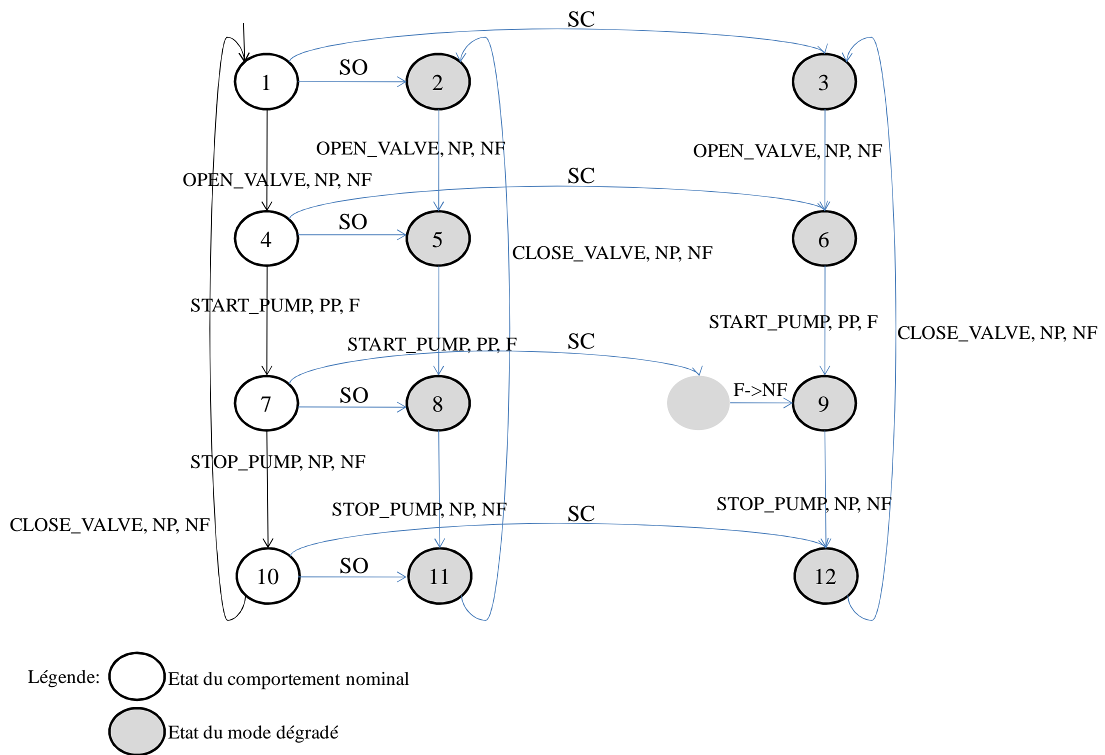

Figure 14. Modèle fonctionnel global

Capteur de pression
- **PP. Pump Pressure**
- **NP : No Pressure**

Capteur de flux :
- **F : Flow**
- **NF : No Flow**

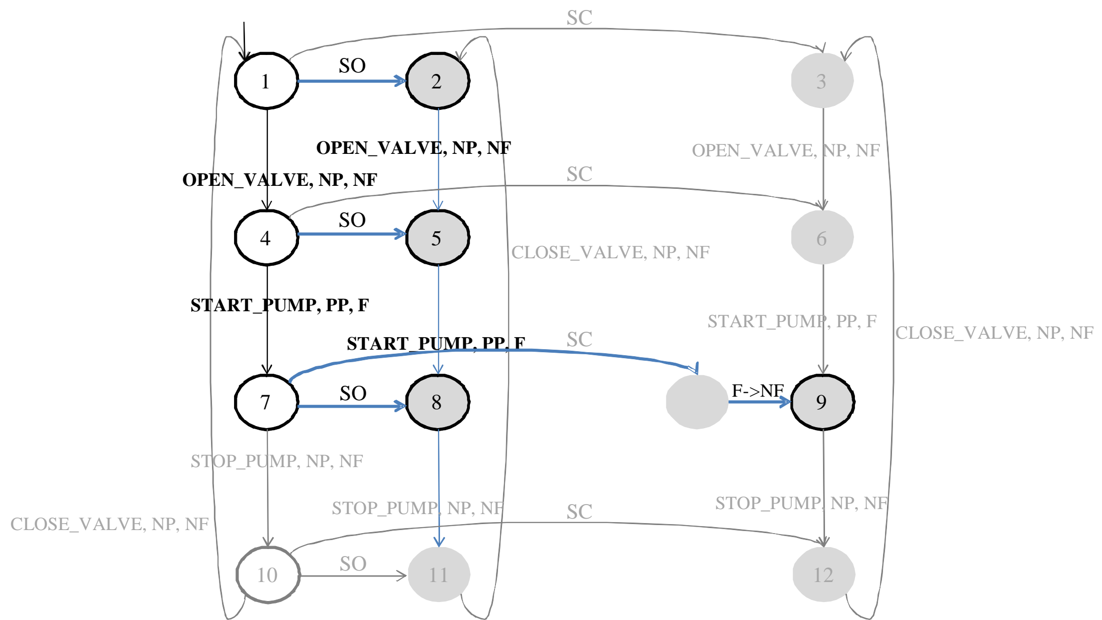

Figure 15. La séquence d'événements observables du sous-système PVC

<OPEN_VALVE,NP,NF>, <START_PUMP,PP,F>

À partir du modèle global du système, l'approche de Meera Sampath génère un « Diagnostiqueur » pour les séquences d'événements observables du système (ou sous-système) à analyser. Car, ces événements peuvent être observables à la fois en mode de fonctionnement nominal et dégradé du système ; il est donc nécessaire de distinguer les deux situations grâce au diagnostiqueur.
Dans le cas du sous-système PVC, O= <OPEN_VALVE,NP,NF>, <START_PUMP,PP,F> est une séquence d'événements observables. Elle est possible dans les deux cas de fonctionnement nominal et dégradé (Figure 15). L'expression de cette séquence d'événements observables en termes d'états, génère un ensemble Δ(O) de toutes les possibilités : Δ(O)= {(1,4,7) (1,4,7,8) (1,2,5,8) (1,4,5,8) (1,4,7,9)}.
De la même manière, il faut trouver tous les Δ(O) du système à analyser, jusqu'à en réaliser un observateur des événements, dont les arcs représentent les événements observables où chaque nœud regroupe les états qui peuvent provoquer le même événement (Figure 16).

Ainsi, dans l'observateur, un même nœud peut regrouper à la fois des états de fonctionnement nominal et des états de fautes. Pour ceci, vient la dernière étape qui est la réalisation du diagnostiqueur.
Le point de départ de la réalisation d'un diagnostiqueur est l'observateur. Ensuite, un état (ou un nœud) dans le diagnostiqueur peut être :

- N (Normal): si tous ses candidats sont étiquetés N
- F*i* certain : si tous ses candidats sont étiquetés F*i*
- F*i* incertain : pour les autres cas

Dans le cas du système PVC, il existe deux fautes possibles :

- F1 : SC, Stuck Close (bloqué fermé)
- F2 : SO, Stuck Open (bloqué ouvert)

D'après le diagnostiqueur du système PVC (Figure 17), F1 est certain et F2 est incertain.

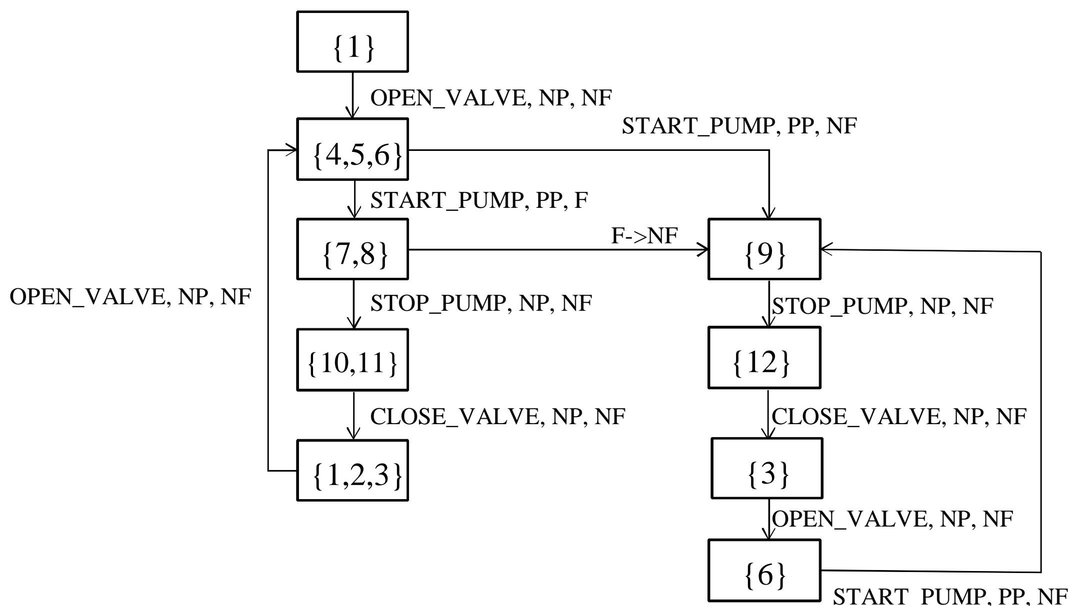

Figure 16. Observateur des événements du système HVAC

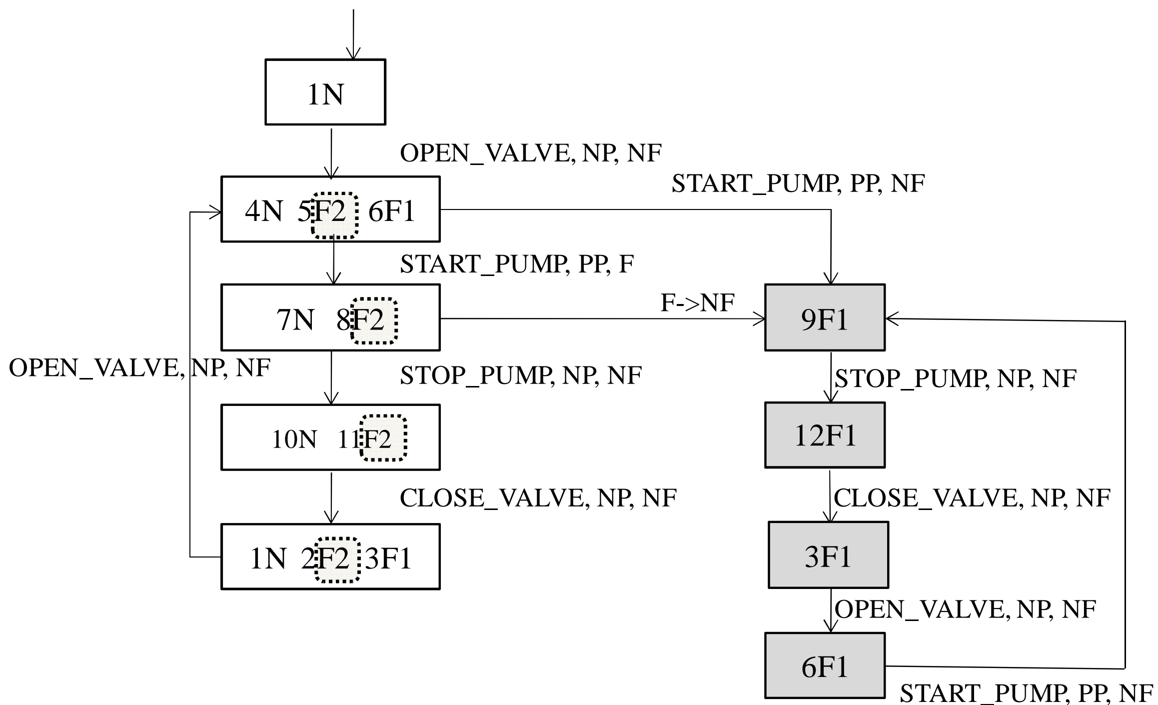

Figure 17. Diagnostiqueur du système PVC

Or d'après l'approche suivie, un système à événements discrets est diagnosticable si et seulement si son diagnostiqueur ne contient pas de cycles indéterminés. Un cycle est indéterminé si :

- Il existe un cycle d'états F*i* incertains dans le diagnostiqueur
- Et les états correspondant, du cycle des Fi incertains, forment aussi un cycle observable dans le modèle global du système.

Dans le cas du système PVC, il existe un cycle d'états F*i* dans le diagnostiqueur qui est le suivant : (4N 5F2 6F1), (7N 8F2), (10N 11F2), (1N 2F2 3F1), (4N 5F2 6F1). Les états correspondants de ce cycle de F*i* incertains forment deux cycles observables dans le modèle global du système : (4, 7, 10, 1, 4) et (5, 8, 11, 2, 5) (Figure 18). Ainsi, le sous-système PVC est non diagnosticable.
Cet exemple prouve que l'approche suivie est simple et efficace pour les systèmes à événements discrets. Cependant, pour pouvoir effectuer un diagnostic efficace en ligne d'un système embarqué, il y a certaines contraintes à imposer à l'architecture telles que : la connectivité des composants, le dimensionnement des calculateurs, la distribution des fonctions sur les calculateurs, etc. Ces contraintes peuvent être déduites de l'analyse de diagnosticabilité classique (telle qu'elle existe dans la littérature) que nous appelons « analyse de diagnosticabilité fonctionnelle », mais en prenant en compte les considérations d'architecture matérielle.

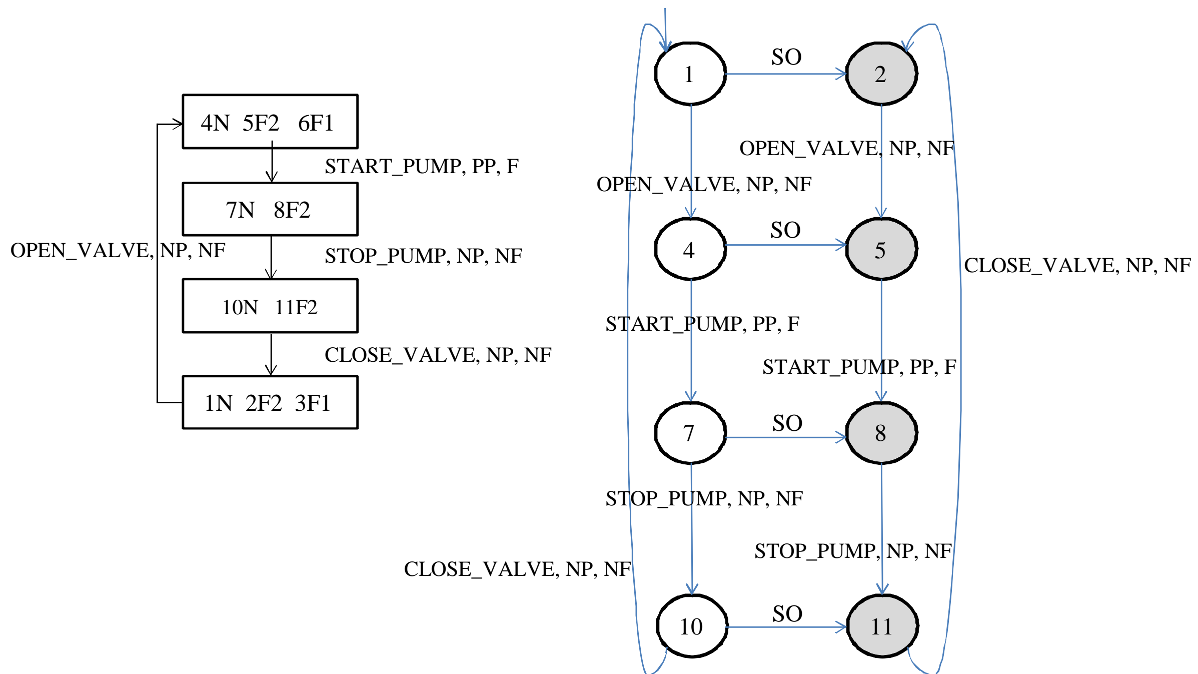

Figure 18. Résultat de l'analyse (non diagnosticable)

Dans cette thèse, nous définissons un certain nombre d'indicateurs pour l'interaction entre fonctions et architecture du système, qui permettent d'estimer l'adéquation de l'architecture vis-à-vis d'un critère donné. Ces métriques accompagneront l'analyse de sa diagnosticabilité suivant les approches basées événements discrets, que nous appelons « approches de diagnosticabilité fonctionnelle ». Ainsi, la vérification de ces métriques est nécessaire quand le résultat de l'analyse de la diagnosticabilité « fonctionnelle » est positif. Les résultats de l'analyse de la diagnosticabilité « fonctionnelle » sont binaires pour un ensemble de fautes donné. Ainsi, si nous incluons également l'analyse des métriques de l'architecture, pour le même ensemble de fautes, le résultat doit rester binaire : l'ensemble fonctions-architecture est diagnosticable ou pas.

III. Vérification des propriétés de la diagnosticabilité fonctionnelle-architecturale

Afin de vérifier la diagnosticabilité fonctionnelle-architecturale, nous proposons un ensemble de propriétés à vérifier. Nous nous focaliserons sur les architectures embarquées distribuées à base de calculateurs reliés à l'aide d'un ou plusieurs réseaux de communication.

1. Définition des propriétés de la diagnosticabilité fonctionnelle-architecturale

Nous définissons, un ensemble de propriétés à vérifier comme complément à la méthode de vérification de la diagnosticabilité des systèmes à événements discrets citée dans [48]. Ces propriétés sont nécessaires aux différentes structures de diagnostic présentées précédemment (à savoir diagnostic centralisé, décentralisé et distribué) et font intervenir la notion du temps pour certains scénarios. La première étape de la vérification des propriétés à effectuer, est la détermination de la structure du diagnostic (centralisé, décentralisé ou distribué) afin de

cerner l'ensemble des propriétés nécessaires. Ainsi, une architecture est dite diagnosticable si et seulement si l'ensemble des propriétés concernant sa structure est vérifié. Le tableau suivant représente un aperçu général sur les différentes propriétés (Tableau 2).

Tableau 2. Aperçu général sur les propriétés de la diagnosticabilité fonctionnelle-architecturale

Propriété	Descriptif	Dépendances des autres propriétés	Mode de vérification
Connectivité_diagnostiqueur	Tout composant (calculateur) contenant une fonction à diagnostiquer doit avoir au moins une connexion de type « données » avec le composant (calculateur) diagnostiqueur, ou qu'il soit lui-même diagnostiqueur dans le cas du diagnostic distribué	aucune	vérification de la description de l'architecture
Exécutabilité_diagnostiqueur	Pour toute fonction de diagnostic, il faut que la taille mémoire du calculateur sur lequel elle doit être implantée, soit supérieure ou égale à la taille mémoire de la fonction de diagnostic elle-même. Il faut aussi le temps d'exécution le plus long de la fonction du diagnostic, sur le même calculateur soit inférieur ou égal à la périodicité souhaitée pour l'exécution de la fonction de diagnostic.	aucune	vérification de la description de l'architecture
Atteignabilité	L'observateur doit atteindre (recevoir) les valeurs de toutes les variables d'états représentant les informations issues de capteurs ou autres composants du système	aucune	vérification de la description de l'architecture
Accessibilité	L'observateur doit recevoir des notifications sur les événements de tous les composants du système à analyser.	aucune	vérification de la description de l'architecture
Disponibilité temporelle	Cette métrique prend la valeur « vrai » lorsque le temps disponible sur le calculateur prévu pour le diagnostic est suffisant pour un diagnostiqueur (ou sous-fonction du diagnostiqueur dans le cas du diagnostic décentralisé)	aucune	Vérification de l'interaction de l'architecture avec les fonctions
Observabilité	Toute variable de chaque événement dans l'automate fini, représentant le système, doit être observable pour que la totalité de l'événement soit observable	• Atteignabilité • Accessibilité • Disponibilité temporelle	vérification de la description de l'architecture et vérification de l'interaction fonctions-architecture

Afin de définir les propriétés, nous définissons d'abord le système à l'aide d'un ensemble de variables et fonctions représentant ses différents composants et caractéristiques.

Soient les variables :
A : L'ensemble des calculateurs (ou composants) contenant une ou plusieurs fonctions à analyser
B : L'ensemble des composants sources de variables d'états ou d'événements
C : L'ensemble de tous les composants calculateurs

D_principal : Le calculateur (ou composant) diagnostiqueur (Cas du diagnostic centralisé et décentralisé)
D : L'ensemble des calculateurs (ou composants) diagnostiqueurs
E : L'ensemble des variables représentant les événements
F : L'ensemble de toutes les fonctions
F_Diag : L'ensemble des fonctions de diagnostic
M : L'ensemble des messages envoyés à travers le bus de communication
N : Le nombre total de A
O : Le calculateur ou composant contenant le processus observateur
P : Le nombre total de F
Q : Le nombre total de V
R : Le nombre total de E
S : Le nombre total de D
V : L'ensemble des variables d'états issues de capteurs ou autres composants du système

Soit le type :
void : Type de retour non défini différent de null

Soient les fonctions :
F_Data : $(x,y) \rightarrow \{0,1\} \backslash (x,y) \in C^2$ retourne 1 si x et y sont connectés par une connexion de type données, retourne 0 sinon
Size_F : $x \rightarrow y \backslash x \in$ F_Diag, $y \in \mathbb{R}$ retourne la taille de x
Size_Mem : $x \rightarrow y \backslash x \in C, y \in \mathbb{R}$ retourne la taille mémoire de x
ΔT : $x \rightarrow y \backslash x \in$ F_Diag, $y \in \mathbb{R}$ Fonction qui retourne la périodicité souhaitée pour l'exécution de la fonction x
WCET : $(x,y) \rightarrow z \backslash x \in$ F_Diag, $y \in C, z \in \mathbb{R}$ retourne le temps maximal d'exécution que x peut prendre sur la plateforme y, correspond au WCET (W*orst-Case Execution Time*) de x sur y [12]. Nous présumons l'existence de cette fonction.
F_I/O : $(x,y) \rightarrow z \backslash x \in$ F_Diag, $y \in B, z \in \{0,1\}$ retourne 1 si x utilise des inputs de y ou retourne des outputs à y
Conn_I/O : $(x,y) \rightarrow z \backslash x \in C, y \in B, z \in \{0,1\}$ retourne 1 si x et y sont connectés
Implemented : $(x,y) \rightarrow z \backslash x \in F, y \in C, z \in \{0,1\}$retourne 1 si x est implémentée sur y
Source : $x \rightarrow y \backslash x \in M, y \in C$ retourne le nom (ou adresse) du calculateur émetteur du message x.
Destination : $x \rightarrow y \backslash x \in M, y \in C$ retourne le nom (ou adresse) du calculateur récepteur du message x.
Contenu : $x \rightarrow y \backslash x \in M, y \in$ « void » retourne le contenu du message M.
Valeur : $x \rightarrow y \backslash x \in V, y \in$ « void » retourne la valeur de la variable V.
Origine : $x \rightarrow y \backslash x \in \{E, V\}, y \in B$ retourne le composant source de l'événement ou de la variable d'état.

Les propriétés doivent être vérifiées dans l'ordre suivant :

Règle 1 : Pour chaque structure de diagnostic, il existe un ensemble de propriétés à vérifier :

- Centralisé : propriétés 1.1, 2, 3, 4, 5 et 6
- Décentralisé : propriétés 1.2, 3, 4, 5 et 6
- Distribué : propriétés 1.3, 2, 3, 4, 5 et 6

Propriété 1 : Connectivité-Diagnostiqueur

- ***Propriété 1.1 :*** Pour tout composant (calculateur) contenant une fonction à diagnostiquer, il faut qu'il ait au moins une connexion de type « données » avec le composant (calculateur) diagnostiqueur.

$$\forall a \in A,$$
$$(F_Data\ (a, D_principal)) \lor (a == D_principal) \rightarrow Connectivité_Diagnostiqueur\ (a)=1$$

Représentation algorithmique de la propriété :
Soient :
Tab_A : Tableau représentant A
Var_D : Variable représentant D_principal
F_Data : Fonction se basant sur l'analyse de la description de l'architecture (parser le code)

```
Booléen Connectivité_Diagnostiqueur_Centralisé (Tab_A : Tableau de chaines de caractères, Var_D : chaines de caractères)
i : entier ;
Conn_Diag : Booléen ;
Pour i=1 à N Faire
    Si (F_Data (Tab_A[i], Var_D) ==1 OU Tab_A[i] ==Var_D)
    Alors Conn_Diag=1;
    Sinon Afficher (Tab_A[i]); Return 0; Sortir;
    Fin si;
Fin pour ;
Return 1;
Fin Connectivité_Diagnostiqueur_Centralisé;
```

- ***Propriété 1.2 :*** Pour tout composant (calculateur) de diagnostic auxiliaire, il faut qu'il ait au moins une connexion de type « données » avec le composant (calculateur) diagnostiqueur principal.

$$\forall d \in D,$$
$$((Conn_I/O\ (d, D_principal) ==1) \rightarrow Connectivité_Diagnostiqueur\ (d)=1$$

Représentation algorithmique de la propriété :
Soient :
Tab_D : Tableau représentant D
Var_D : Variable représentant D_principal
Conn_I/O : Fonction se basant sur l'analyse de la description de l'architecture (parser le code)

```
Booléen Connectivité_Diagnostiqueur_Décentralisé (Tab_D : Tableau de chaines de caractères, Var_D : chaines de caractères)
i : entier ;
Conn_Diag : Booléen ;
Pour i=1 à S Faire
    Si (Conn_I/O (Tab_D[i], Var_D))
    Alors Conn_Diag=1;
    Sinon Afficher (Tab_D[i]); Return 0; Sortir;
    Fin si;
```

```
    Fin pour ;
  Return 1;
  Fin Connectivité_Diagnostiqueur_Décentralisé ;
```

- ***Propriété 1.3 :*** Pour tout composant (calculateur) contenant une fonction à diagnostiquer, il faut qu'il soit lui-même un composant diagnostiqueur.

$\forall a \in A, \forall d \in D,$

$(a{=}{=}d) \rightarrow$ Connectivité_Diagnostiqueur (a)=1

Représentation algorithmique de la propriété :
Soient :
Tab_A : Tableau représentant A
Tab_D : Tableau représentant D

```
Booléen  Connectivité_Diagnostiqueur_Distribué  (Tab_A :  Tableau  de
chaines de caractères, Tab_D : Tableau de chaines de caractères)
i, j : entier ;
Conn_Diag : Booléen ;
Pour i=1 à N Faire
  Pour j=1 à S Faire
    Si (Tab_A[i]) == Tab_D[j])
    Alors Conn_Diag=1;
    Sinon Afficher (Tab_A[i]); Return 0; Sortir;
    Fin si;
  Fin pour
Fin pour ;
Return 1;
Fin Connectivité_Diagnostiqueur_Distribué ;
```

Propriété 2 : Exécutabilité diagnostiqueur

Pour toute fonction de diagnostic, il faut que la taille mémoire du calculateur sur lequel elle doit être implantée, soit supérieure ou égale à la taille mémoire de la fonction de diagnostic elle-même. Il faut aussi que le temps d'exécution le plus long de la fonction du diagnostic, sur le même calculateur soit inférieur ou égal à la périodicité souhaitée pour l'exécution de la fonction de diagnostic. Dans le cas d'une égalité de la taille mémoire ou du temps d'exécution, le calculateur ne peut exécuter que la fonction de diagnostic en question.

$\forall d \in D, \forall f \in$ F_Diag,

((Implemented (f,d)==1) ^ (Size_Mem (d)≥ Size_F (f)) ^ (ΔT (f) >= WCET (f,d))) → Exécutabilité (f, d) =1

Représentation algorithmique de la propriété :
Soient :
Tab_D : Tableau représentant D
Tab_F : Tableau représentant F
Size_mem : Fonction se basant sur l'analyse de la description de l'architecture (parser

le code)
Size_F : Fonction se basant sur l'analyse de la description de l'architecture (parser le code)
Implemented : Fonction se basant sur l'analyse de la description de l'architecture (parser le code)
Delta_Time : Fonction qui retourne la périodicité souhaitée pour l'exécution d'un processus (ou fonction) de diagnostic
WCET : Fonction qui retourne le temps d'exécution le plus long d'un processus donné sur une plateforme donnée, en se basant sur les approches de WCET [12].

```
Booléen Exécutabilité (Tab_F : Tableau de chaines de caractères,
Tab_D : Tableau de chaines de caractères)
i, j : entier ;
Exc : Booléen ;
Pour i=1 à S Faire
  Pour j=1 à P Faire
    Si ((Implemented (Tab_F[j], Tab_D[i]) ==1) ET
(Size_Mem(Tab_D[i])>= Size_F(Tab_F[j]) ET ((Delta_Time (Tab_F[j])>=
WCET (Tab_F[j], Tab_D[i])))
    Alors Exc=1;
    Sinon Afficher (Tab_F[j], Tab_D[i]); Return 0; Sortir;
    Fin si;
Fin pour ;
Fin pour ;
Return 1;
Fin Exécutabilité ;
```

Propriété 3: Atteignabilité

Dans notre approche, nous voyons que la décision de l'observabilité d'un événement, dans le modèle (l'automate fini) représentant le système (paragraphe II), ne de doit pas s'arrêter aux seules informations issues des capteurs, car l'architecture entière logicielle et matérielle, en interaction avec les capteurs, peut jouer un rôle important pour observer les événements du système. Ainsi un processus « observateur » (Figure 19), faisant partie du diagnostiqueur, doit être implémenté et doit atteindre (recevoir) les valeurs de toutes les variables d'états représentant les informations issues de capteurs ou autres composants du système (Figure 20).

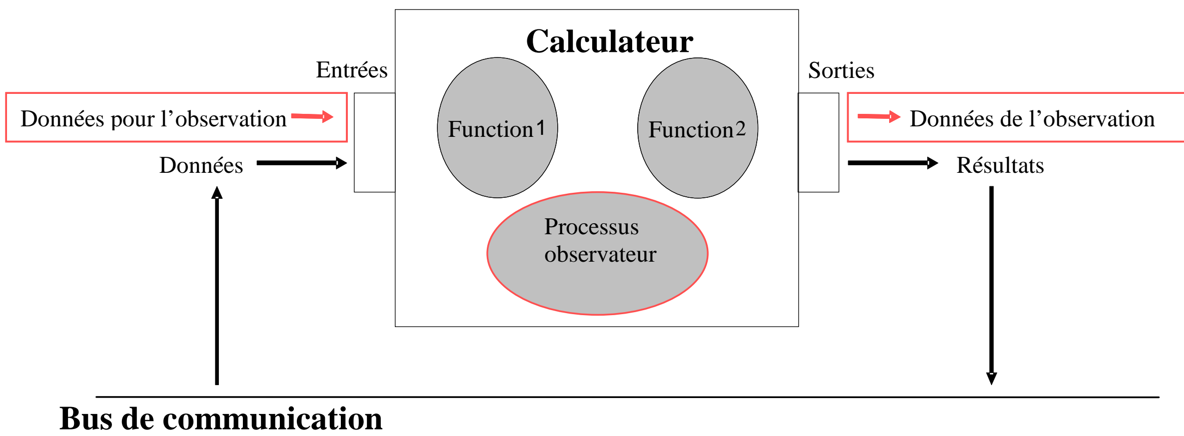

Figure 19. Processus d'observation

Le processus « observateur » est sensé vérifier l'observabilité des événements au cours du fonctionnement du système, sans interférer avec le fonctionnement nominal du système.

L’observateur doit recevoir les valeurs des variables d’états de tous les composants du système à analyser. Une variable d’état est dite atteignable quand le composant matériel comprenant l’origine de cette variable est atteignable physiquement par le composant matériel sur lequel est implanté l’observateur.

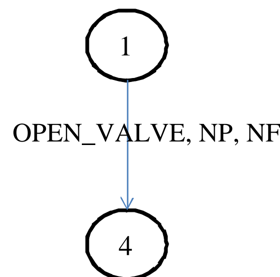

Figure 20. Exemple d’événement
OPEN_VALVE : variable représentant l’événement
NP, NF : variables d’états représentant l’information issue du capteur

$\forall v \in V, O \in C, \forall m \in M,$
(Conn_I/O (Origine(v), O) ∨ F_Data (Origine(v), O)) ^ Source (m)=Origine(v) ^ Destination(m)=O ^ Contenu (m) = Valeur(v) → Atteignabilité (v)=1

Représentation algorithmique de la propriété :

Soient :

Struct_M : Structure de données [Message, Source, destination, Contenu], représentant un message, sa source, sa destination et son contenu.

Struct_W : Structure de données [Variable_état, Origine], représentant une variable d’état, son origine et le message qui l’envoie à l’observateur.

Tab_W : Tableau représentant V

Var_O : Variable représentant O

Conn_I/O : Fonction se basant sur l’analyse de la description de l’architecture (parser le code)

Chercher_source(Tableau_messages, S) : Fonction se basant sur l’analyse du code, cherchant la liste des massages parmi « Tableau_message » ayant comme source « S ».

Chercher_destination(Tableau_messages, D) : Fonction se basant sur l’analyse du code, cherchant la liste des massages parmi « Tableau_message » ayant comme destination « D ».

Chercher_contenu(Tableau_messages, C) : Fonction se basant sur l’analyse du code, cherchant la liste des massages parmi « Tableau_message » ayant comme contenu « C ».

```
Booléen Atteignabilité (Tab_M : Tableau de Struct_M, Tab_W : Tableau
de Struct_W, Var_O : chaines de caractères)
i : entier ;
Att : Booléen ;
Pour i=1 à Q Faire
     Tab1 = Chercher_source(Tab_M, Tab_W[i].Origine) ;
     Tab2 = Chercher_destination(Tab_M, Var_O) ;
     Tab3 = Chercher_contenu(Tab_M, Tab_W[i].Variable_état) ;
     Message =Tab1 intersection Tab2 intersection Tab3 ;
    Si ((Conn_I/O (Tab_W[i].Origine,Var_O)==1 OU
```

```
          F_Data(Tab_W[i].Origine,Var_O))
          ET (Message !=null)
         Alors Att=1;
         Sinon Afficher (Tab_W [i]); Return 0; Sortir;
         Fin si;
     Fin pour ;
     Return 1;
     Fin Atteignabilité ;
```

Propriété 4 : Accessibilité

L'observateur doit recevoir des notifications sur les événements de tous les composants du système à analyser. Une variable représentant un événement est dite accessible quand le composant matériel source de cette variable est accessible physiquement par le composant matériel sur lequel est implanté l'observateur. La différence avec l'atteignabilité est que cette dernière s'intéresse aux variables d'état représentant les informations issues de capteurs et non pas aux variables représentant les événements.

$\forall e \in E, \forall c \in C, \forall m \in M,$
(Conn_I/O (Origine(e), O) $\vee$ F_Data (Origine(e), O)) ^ Source (m)=Origine(e) ^ Destination(m)=O ^ Contenu(m)=Valeur(e) → Accessibilité (e)=1

Représentation algorithmique de la propriété :

Soient :

Struct_M : Structure de données [Message, Source, destination, Contenu], représentant un message, sa source, sa destination et son contenu.

Struct_E : Structure de données [Evenement, Origine], représentant une variable d'état, son origine et le message qui l'envoie à l'observateur.

Tab_E : Tableau de Struct_E représentant E

Var_O : Variable représentant O

Conn_I/O : Fonction se basant sur l'analyse de la description de l'architecture (parser le code)

Chercher_source(Tableau_messages, S) : Fonction se basant sur l'analyse du code, cherchant la liste des massages parmi « Tableau_message » ayant comme source « S ».

Chercher_destination(Tableau_messages, D) : Fonction se basant sur l'analyse du code, cherchant la liste des massages parmi « Tableau_message » ayant comme destination « D ».

Chercher_contenu(Tableau_messages, C) : Fonction se basant sur l'analyse du code, cherchant la liste des massages parmi « Tableau_message » ayant comme contenu « C ».

```
     Booléen Atteignabilité (Tab_M : Tableau de Struct_M, Tab_E : Tableau
     de Struct_E, Var_O : chaines de caractères)
     i : entier;
     Att : Booléen ;
     Pour i=1 à R Faire
          Tab1 = Chercher_source(Tab_M, Tab_E[i].Origine) ;
          Tab2 = Chercher_destination(Tab_M, Var_O) ;
          Tab3 = Chercher_contenu(Tab_M, Tab_E[i].Evenement) ;
          Message =Tab1 intersection Tab2 intersection Tab3 ;
         Si ((Conn_I/O (Tab_E[i].Origine, Var_O)==1 OU
          F_Data(Tab_E[i].Origine, Var_O ))
```

```
        ET (Message!=null)
       Alors Att=1;
       Sinon Afficher (Tab_E[i]); Return 0; Sortir;
       Fin si;
   Fin pour ;
   Return 1;
   Fin Atteignabilité ;
```

Propriété 5 : Disponibilité temporelle

Cette métrique prend la valeur « VRAI » lorsque le temps disponible sur le calculateur prévu pour le diagnostic est suffisant pour un diagnostiqueur (ou sous-fonction du diagnostiqueur dans le cas du diagnostic décentralisé). Par conséquent, pour déterminer les intervalles de temps disponible pour le diagnostiqueur, nous cherchons les cycles libres non exploités par le fonctionnement nominal du système. Pour ceci, nous devons comparer la durée des cycles de temps disponibles avec le temps nécessaire pour exécuter le diagnostiqueur en utilisant les mêmes ressources matérielles (Figure 21).

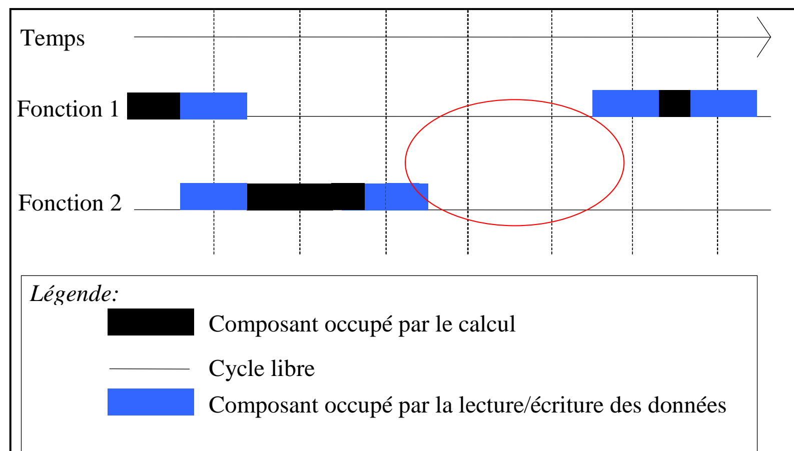

Figure 21. Cycles d'exécution de processus

Propriété 6 : Observabilité [64]
Toute variable de chaque événement dans l'automate fini (Figure 20), représentant le système, doit être observable pour que la totalité de l'événement soit observable. Pour satisfaire cette propriété, il faut :

- vérifier qu'il existe une disponibilité temporelle au niveau du système pour implémenter un diagnostiqueur,
- vérifier l'atteignabilité de l'observateur aux variables d'états portant des informations issues des capteurs ou autres composants
- et vérifier l'accessibilité des événements par l'observateur

Règle 2 :

Toute propriété de la diagnosticabilité fonctionnelle-architecturale est vérifiable grâce à la représentation ou la modélisation du système. Les propriétés {1.1, 1.2, 1.3, 2, 3, 4} sont vérifiables au niveau de la description de l'architecture, alors que les propriétés {5 et 6} sont vérifiables au niveau de l'interaction fonctions- architecture :

- Vérification de la description de l'architecture : non temporelle, faite à partir de la description (ou la documentation) de l'architecture.

•	Vérification de l'interaction des fonctions avec l'architecture : prend en considération l'avancement de l'état du système matériel-logiciel en fonction du temps

Nous avons effectué une synthèse des propriétés sous forme de différentes classifications suivant différents critères :

- La dépendance de chaque propriété des autres propriétés
- Le mode de vérification de chacune des propriétés
- Les propriétés qui concernent le diagnostiqueur
- Les propriétés qui concernent l'observateur (Figure 22).

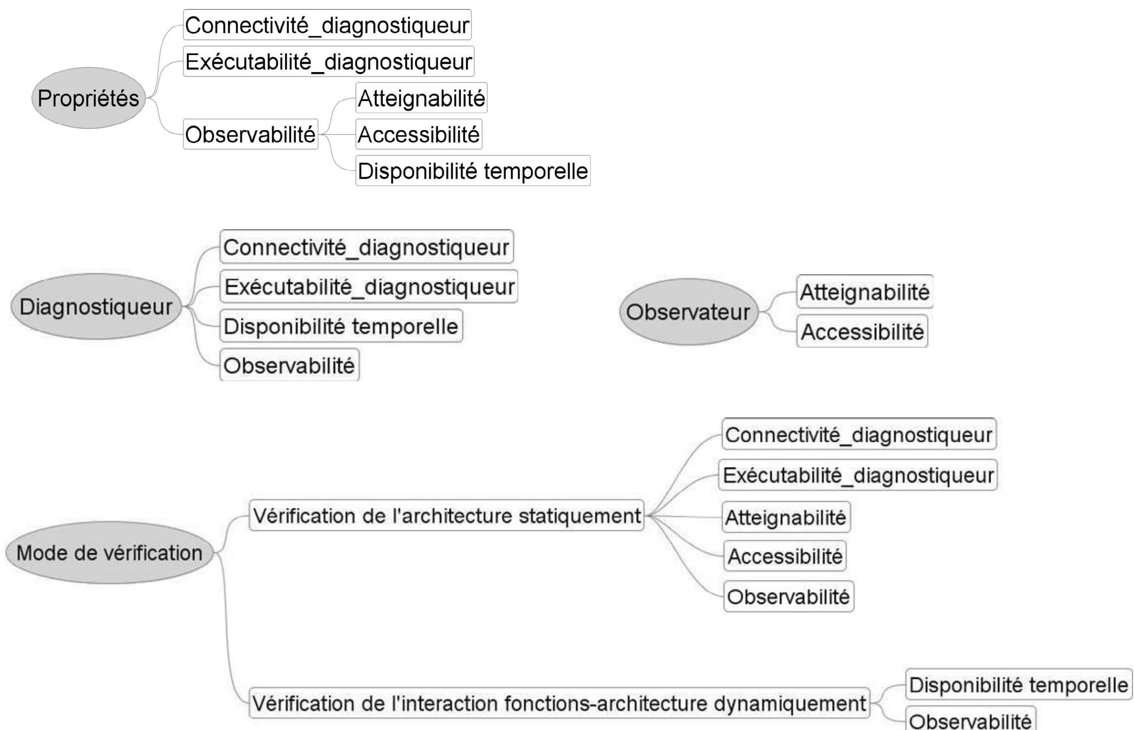

Figure 22. Classification des propriétés selon différents critères

2. Vérification des propriétés

Afin de vérifier ces propriétés de la diagnosticabilité fonctionnelle-architecturale, il faudrait d'abord décrire l'architecture dans un langage de description d'architecture tel que AADL (*Architecture Analysis and Description Language*) [65], ou un langage de description du matériel tel que VHDL (*Very high speed integrated circuit Hardware Description Language*) [66] ou SystemC. Ensuite, nous procédons à l'analyse des propriétés qui dépendent de l'évolution de l'état du système dans le temps. Pour ce faire nous avions le choix entre développer notre propre moteur d'analyse ou de se baser sur des moteurs existants. Nous avons choisi de nous baser sur les moteurs de simulation compatibles avec les formalismes d'expression de modèles les plus pratiques en industrie tel que Matlab/Simulink, etc. Nous demandons donc aux différents moteurs de simulation de « dérouler » le comportement du système selon son modèle et selon des scénarios représentant des scénarios critiques. Par conséquent, nous avons orienté nos efforts vers l'interprétation des résultats de simulation d'un modèle. En effet, quand nous simulons un modèle, nous pouvons produire un fichier de traces enregistrant toutes les variations, datées précisément des valeurs des signaux.

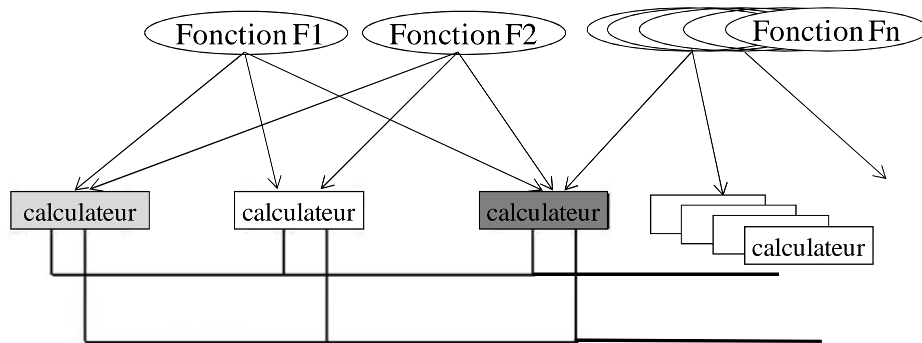

Figure 23. Répartition des fonctions sur le matériel

Afin de pouvoir simuler le système dont on souhaite vérifier ses propriétés de l'interaction fonctions-architecture, nous utilisons une co-modélisation matérielle-logicielle. La co-modélisation permet de représenter la répartition des différentes fonctions sur les différents calculateurs (Figure 23), d'une manière cohérente, offrant la possibilité que le modèle soit connecté à d'autres modèles qui peuvent être exprimés avec des langages différents. Notre approche doit permettre au concepteur du système d'intégrer les différents modèles afin de les simuler et les tester. Ensuite, la vérification des propriétés de l'interaction fonctions-architecture, se fait grâce à notre méthode d'analyse des traces de co-simulation de modèles logiciels/matériels (Figure 24).

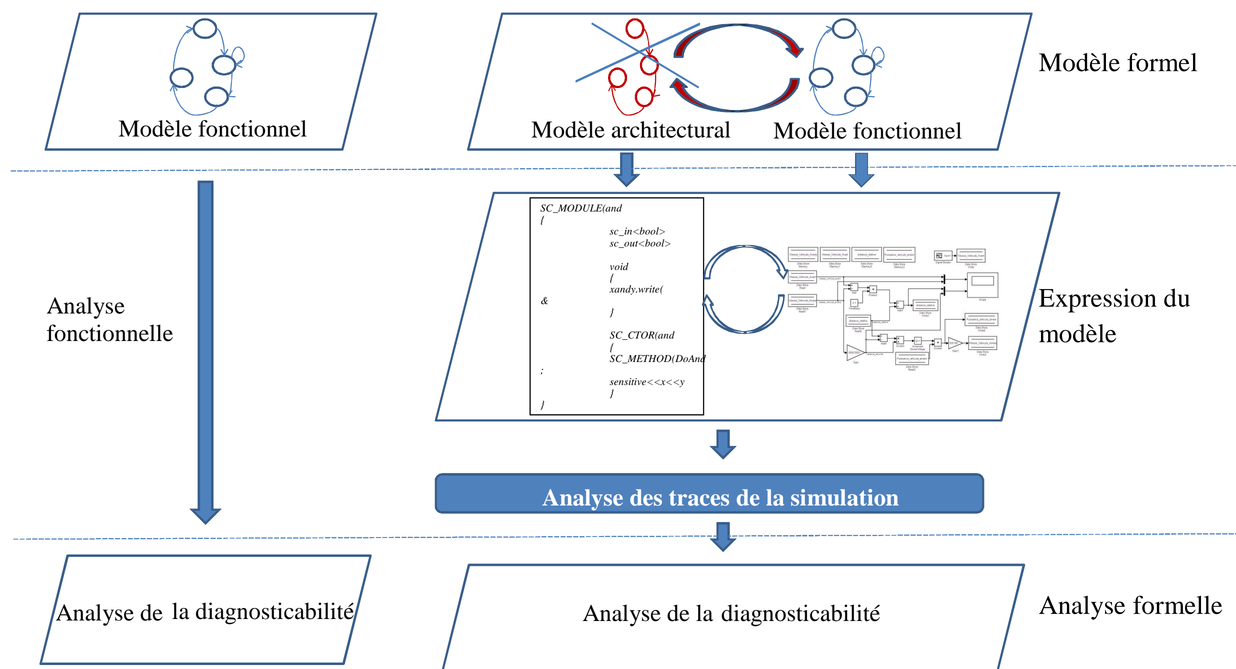

Figure 24. Analyse de la diagnosticabilité fonctionnelle-architecturale

3. La co-modélisation matérielle-logicielle

Le modèle du système représente l'élément d'entrée de l'étape de la simulation, qui précède l'étape de la vérification de la propriété. Il est possible pour cette fin de modéliser le système de deux manières différentes : soit avec une modélisation homogène à la fois architecturale et fonctionnelle en utilisant par exemple le langage SystemC, soit avec une modélisation hétérogène en utilisant un langage pour le matériel et un langage pour le logiciel. En effet,

SystemC est une extension du langage C + + avec l'utilisation de bibliothèques de classes. Ses avantages sont les suivants :

- Il permet la modélisation matérielle-logicielle avec le même langage de modélisation,
- Les modèles peuvent être facilement reliés à d'autres modèles matériels [67], ou à des modèles fonctionnels (par exemple modèles Simulink) [68],
- L'environnement SystemC possède également un simulateur : il se compose de la bibliothèque C++ et d'un moteur de simulation événementiel,
- Toute librairie C ou C + + peut être incluse dans un co-modèle matériel-logiciel donc, nous pouvons décrire le comportement approprié du système avec différents niveaux de hiérarchie.

Malgré tous les avantages d'une co-modélisation matérielle-logicielle, nous allons limiter dans notre approche le SystemC à la modélisation matérielle et utiliser Simulink pour la modélisation logicielle. En effet, la modélisation en Simulink est largement pratiquée dans l'industrie. Ainsi, le modèle, qui sera utilisé par la suite pour la simulation, devient encore plus conforme à la réalité. Comme le matériel et le logiciel sont interdépendants, un interfaçage entre l'environnement de SystemC et celui de Simulink est primordial pour assurer la modélisation complète d'un système : modélisation matérielle, modélisation logicielle et interconnexion entre les deux.

4. La co-simulation

L'interconnexion entre SystemC et Simulink ne peut pas être établie d'une façon directe. Le passage par l'environnement Matlab est nécessaire. Par conséquent, ces derniers communiquent entre eux en passant par deux étapes : d'abord SystemC se connecte à l'environnement Matlab qui est relié à Simulink nativement.

Grâce à la librairie « engine.h » (*Matlab engine library*), il est possible d'appeler et de commander Matlab depuis un programme C, C++, Fortran, etc. Puisque SystemC est une extension de C++, l'interconnexion SystemC-Matlab sera ainsi établie via cette librairie : il faut ajouter chacune des libraires «systemc.h» et «engine.h» au standard C++. (Annexe I). La librairie « engine.h » contient plusieurs routines qui permettent de :

- Lancer ou arrêter un processus Matlab communicant avec un programme C++,
- Echanger des données entre Matlab et C++,
- Exécuter des commandes Matlab à partir d'un environnement C++.

Le Tableau 3 présente les principales routines de la librairie « engine.h ».

Tableau 3. Quelques routines de la librairie "engine.h"

Routines C++	Description
engOpen	Lancer un processus Matlab
engClose	Fermer un processus Matlab
engGetVariable	Lire une matrice de Matlab
engSetVariable	Ecrire dans une matrice de Matlab
engEvalString	Exécuter une commande Matlab

Contrairement à C++, Matlab ne dispose que d'un seul type de données : les matrices (*Matlab Arrays*). Ainsi, pour que Matlab puisse "comprendre" les variables de C++, la librairie « engine.h » raccorde à C++ un nouveau type de donnée « mxArray », reconnu par les deux environnements. Il s'agit d'une structure contenant plusieurs champs dont un pour le type de

la matrice, ses dimensions, la donnée associée (data), etc. Ainsi, la librairie « engine.h » offre une liste de routines préfixées par « mx » qui permettent de créer, accéder, manipuler et détruire les mxArrays.

Puisque Simulink est intégré dans l'environnement Matlab, l'interconnexion entre ces deux environnements nécessite des blocs Simulink prédéfinis et dédiés à la communication entre eux.

SystemC doit mener la co-simulation. Il instancie tout d'abord l'architecture matérielle du système, détermine les paramètres de la simulation. Suivant ces paramètres, il envoie périodiquement de nouvelles données vers les modèles Simulink, contrôle l'avancement de leurs simulations et récupère les résultats de leurs calculs [69].
D'une façon itérative, il est possible de fournir de nouvelles entrées au modèle Simulink, avancer la simulation d'un pas et récupérer les nouvelles sorties. L'un des principaux inconvénients de l'analyse des résultats de simulation est la dépendance de la simulation des « conditions initiales ». En effet, la simulation peut être considérée comme une « exécution symbolique instanciée » nécessitant des valeurs initiales aux entrées et variables du système. Le prochain paragraphe présente la solution que nous avons retenue.

5. Couverture des conditions

Afin de garantir une certaine exhaustivité des cas de figure, nous nous somme inspirés des techniques de couvertures de tests, pour définir les valeurs initiales et les scénarios de simulation. Plusieurs critères et méthodes de couverture de tests de code sont utilisés dans le monde du génie logiciel. Parmi les différents types de couvertures nous citons ci-après des moins complexes aux plus complexes :

- ***Decision Coverage*** **(Couverture des décisions) :** Chaque décision dans le système représente une situation particulière. Chaque décision doit être pratiquée au moins une fois pour garantir la couverture.
- ***Path Coverage*** **(Couverture des chemins) :** Dans ce type de couverture, tous les chemins d'un programme doivent être couverts et exécutés.
- ***Condition Coverage*** **(Couverture des conditions) :** Ce type de couverture évalue chaque sous-expression booléenne à Vrai ou Faux. La couverture des conditions exige que toutes les conditions dans un programme aient pris toutes les valeurs possibles au moins une fois, mais sans exiger le changement de la valeur de décision au moins une fois. Donc les valeurs des conditions seront libres de permuter entre Vrai et Faux.
- ***Condition/Decision Coverage*** **(Couverture des Conditions/Décisions) :** Ce type de couverture assemble les exigences des deux derniers critères de couverture des décisions et celle des conditions. Donc il faut avoir un nombre de cas de tests suffisant à faire basculer les valeurs de toutes les décisions et toutes les conditions entre vrai et faux.
- ***Multiple Condition Coverage*** **(Couverture multiple des conditions) :** La couverture multiple des conditions exige des cas de test qui assurent que chaque combinaison possible d'entrées à une décision soit exécutée au moins une fois. Ce type de couverture exige des tests exhaustifs des conditions d'une décision. En théorie, ce critère de couverture est le type de couverture structurelle le plus souhaité puisqu'il est le plus puissant. Mais cette couverture n'est pas applicable puisque pour une décision avec 'n' conditions, nous avons besoin de 2^n cas de tests ce qui peut nous

emmener à une explosion combinatoire. Les constructeurs d'automobiles désirent appliquer des tests les plus efficaces que possible au moindre coût.

- ***Modified Condition/Decision Coverage*** **(MC/DC) :** Pour appliquer la couverture MC/DC d'une décision, il faut effectuer une couverture de décision et de condition. Donc toutes les conditions dans les décisions doivent prendre toutes les valeurs possibles au moins une seule fois et la décision doit au moins prendre les deux valeurs Vrai et Faux. Mais d'autre part, les tests utilisés pour la couverture doivent montrer que chaque condition peut être la seule responsable du basculement de la sortie de la décision entre Vrai et Faux. Le MC/DC est un critère de couverture structurelle où le comportement du programme est comparé face à l'objectif du code source ; c'est de type boîte blanche "white box testing" puisque le code source est directement étudié.

La technique est utilisée dans le standard DO-178B pour assurer que le niveau logiciel A (Catastrophique) est testé adéquatement.

Afin de mettre au point des scénarios pertinents pour les cycles de simulation, nous nous sommes inspirés de certaines des techniques MC/DC pour la sélection des valeurs d'initialisation, ainsi que pour le déroulement des événements [70].

6. Analyse des traces : l'outil COSITA

Le simulateur SystemC, génère à la fin de la simulation un fichier de traces de simulation au format VCD (*Value Change Dump*), spécifié avec Verilog par le standard IEEE 1364-1995, en ajoutant l'instruction "sc_create_vcd_trace_file()" au modèle. Le fichier de traces reprend de façon chronologique l'ensemble des événements qui ont affecté le système et est structuré dans un format libre. Le fichier commence par l'information d'en-tête donnant la date, le numéro de la version du simulateur, et l'unité de temps utilisée. Ensuite, le fichier contient les définitions des types des variables enregistrées, suivies des changements de leurs valeurs à chaque incrémentation du temps de simulation. Seules les variables qui changent de valeur au moment d'une incrémentation du temps apparaissent dans le fichier (Figure 25).

```
$date
     Mar 17, 2008       10:03:45
$end

$version
              SystemC 2.1.v1 --- Jun 11 2007 17:21:36
$end

$timescale
     1 ps
$end

$scope module SystemC $end
$var wire    1  aaa  clk        $end
$var wire   32  aab  vir_in [31:0]  $end
$var wire   32  aac  vir_out [31:0]  $end
$var wire   16  aad  logic_in [15:0]  $end
$var wire   16  aae  logic_out [15:0]  $end
$upscope $end
$enddefinitions  $end

$comment
All initial values are dumped below at time 0 sec = 0 t
$end

$dumpvars
1aaa
b1 aab
b0 aac
bxxxxxxxxxxxxxxxx aad
bxxxxxxxxxxxxxxxx aae
$end

#1000
0aaa

#2000
1aaa
b10 aab
b111 aad
b111 aae
```

Figure 25. Format du fichier VCD

Notre approche consiste à analyser le contenu de chaque ligne du fichier trace. Pour vérifier les propriétés de la diagnosticabilité, l'analyse des fichiers des traces nécessite comme paramètres d'entrée :

- Le contenu du fichier VCD,
- La durée du temps nécessaire par cycle (cycle de diagnostic) pour le processus de l'observation,
- La durée de temps requise par le composant matériel (qui nécessite l'observation) pour le calcul,
- Le temps nécessaire pour accéder aux ports matériels en lecture/écriture,
- La durée d'une période d'observation, exprimée comme un multiple du cycle d'horloge.

Nous avons développé notre outil COSITA (*CO-SImulation Trace Analysis*) [71] qui permet d'analyser les fichiers de traces de co-simulation. L'outil lance la co-simulation SystemC-Simulink en générant des scenarios pré-sélectionnés pour la co-simulation. Ensuite, COSITA regroupe les fichiers de traces de co-simulation et analyse les métriques de la diagnosticabilité fonctionnelle-architecturale en analysant le contenu des fichiers traces.
De la même manière que les méthodes formelles de vérification de propriétés, nous utilisons la méthode d'analyse de traces de simulation pour vérifier les propriétés matérielles-logicielles du système modélisé en SystemC-Simulink. COSITA intervient alors exclusivement lors de la vérification des propriétés de l'interaction fonctions-architecture, définies par notre approche.

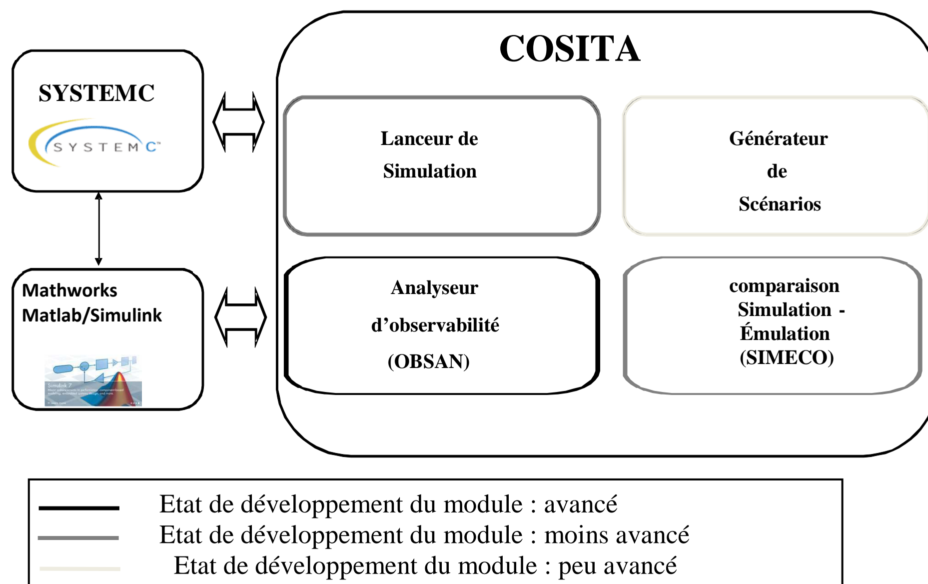

Figure 26. Les différents modules de l'outil COSITA

Les différents modules de COSITA (Figure 26) (Annexe D) sont :

- le générateur de scénarios de co-simulation, est un module qui permet d'utiliser un fichier contenant des différentes valeurs d'entrée du test, représentant le scénario choisi pour la co-simulation, les valeurs choisies dépendent de la technique de couverture des conditions choisie (paragraphe 5). Ensuite, le module affecte les valeurs une à une au modèle au moment de la co-simulation,

- le lanceur de la simulation, permet de lancer l’environnement maitre de la co-simulation, à savoir l’environnement SystemC.
- l’analyseur de l’observabilité (ou de la disponibilité temporelle) appelé « OBSAN » pour (*OBServability ANalyzer*), permet d'effectuer une analyse sur le fichier de traces généré par la simulation, pour détecter les cycles non occupés par le comportement nominal du système. Ceci signifie que nous devions détecter les cycles non occupés par les variations de valeurs pendant la simulation. Ensuite, nous vérifions si le système est suffisamment prêt à implanter le diagnostiqueur dans les cycles libres (pour un diagnostic en temps réel). Nous devons ainsi connaitre certaines caractéristiques du diagnostiqueur tel que le temps de calcul nécessaire par cycle (cycle de diagnostic) et les caractéristiques du calculateur sur lequel devra être implanté le diagnostiqueur. Le résultat symbolique obtenu est un indicateur global de la disponibilité temporelle du système, représenté par la durée maximale d'un diagnostiquer ayant un processus d'observation indépendant pouvant être inséré dans les cycles libres de l'opération du système indiquée en nombre de périodes de l'horloge-système (Annexe A). Si le nombre des cycles libres n'est pas suffisant, nous devrions proposer des améliorations au co-modèle de base du système.
- et finalement un outil de comparaison des résultats issus de la co-simulation et de ceux issus de l’émulation, qui nous servira lors de la phase de validation.

7. Interprétation de l'analyse des métriques

Toutes les propriétés doivent être satisfaites (c'est-à-dire doivent être égales à « vrai». Si le modèle du système ne vérifie pas l’une des propriétés, il faut itérer la modélisation et chercher l’origine du problème ou modifier la conception du système. Par exemple, pour la propriété « observabilité », il existe un lien direct entre l’automate qui représente le fonctionnement du système et la vérification de cette propriété. En effet, il suffit que l’une des variables issues des capteurs, reliée à un événement observable, ne soit pas accessible par le calculateur qui contient le processus observateur, pour que l’événement en question ne soit plus observable (Figure 27). Ainsi les propriétés accessibilité et atteignabilité sont nécessaires pour la propriété observabilité, mais ne sont pas suffisantes car il faut que la disponibilité temporelle soit satisfaite également.

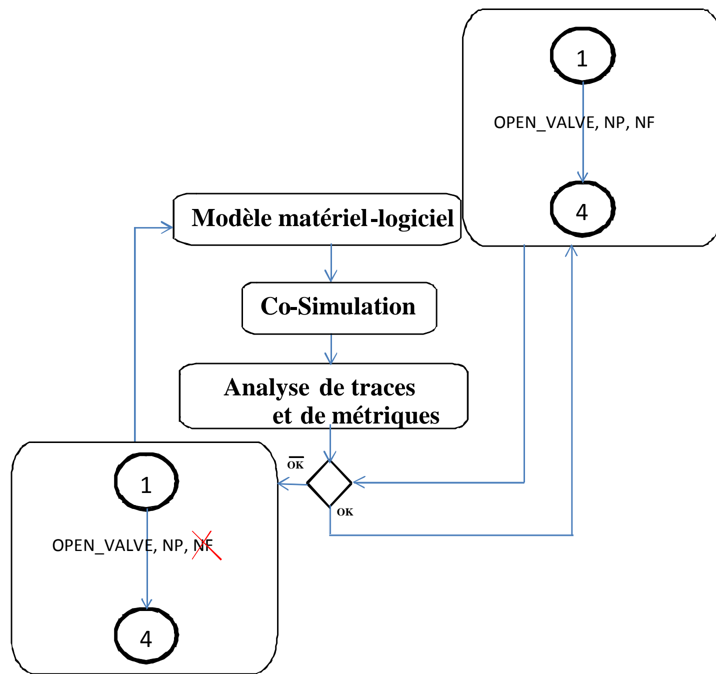

Figure 27. Processus itératif de vérification de la diagnosticabilité basé co-modélisation matérielle-logicielle

IV. Conclusion

Dans ce chapitre, nous avons présenté l'exemple du système HVAC pour l'analyse de la diagnosticabilité des systèmes à événements discrets, telle qu'elle est réalisée dans la littérature. Ensuite, nous avons présenté notre approche d'analyse de diagnosticabilité, qui prend en considération l'interaction architecture-fonctions du système à concevoir. Notre approche nécessite d'abord la réalisation du modèle du système (sous forme d'automates) qui sert à la fois pour l'analyse de la diagnosticabilité fonctionnelle et pour la vérification de quelques propriétés de la diagnosticabilité fonctionnelle-architecturale (ex : accessibilité, atteignabilité). Dans le chapitre suivant nous présenterons une étude de cas réelle, qui fournira une illustration concrète de notre approche.

Etude de Cas : la fonction SDK (Smart Distance Keeping)

I. Introduction

Dans ce chapitre, nous présenterons d'abord les architectures électroniques embarquées dans les véhicules routiers. Ensuite, nous aborderons la fonction SDK (*Smart Distance Keeping*), telle qu'elle est a été présentée par RENAULT TRUCKS[3] ainsi que son modèle mathématique simplifié, que nous avons réalisé. Cette fonction nous servira à découvrir la méthode d'analyse de la diagnosticabilité fonctionnelle-architecturale que nous avons mis au point dans cette thèse. Ce chapitre présente les diverses modélisations des architectures ou des algorithmes que nous avons effectués. Un choix s'est posé lors de la rédaction de ce chapitre : Soit nous incluons dans le corps du texte les modèles exprimés en leur langage correspondant (AADL, SystemC, etc.), soit nous les intégrons aux annexes. Nous avons choisi d'inclure les sources des modèles nécessitant des commentaires circonstanciés. Quant aux restants des modèles, ils ont été reportés en annexes du rapport.

II. Les architectures électroniques embarquées des véhicules

Les fonctions électriques/électroniques embarquées dans les moyens de transport sont en pleine croissance. Nous observons dans la Figure 28 que l'avènement des microprocesseurs dans les années 1980 a été un déclencheur du développement des systèmes embarqués. Les réseaux de communications multiplexés ainsi que le développement des communications sans fil ont contribué à l'accroissement de ces systèmes dans les années 1990 [72].

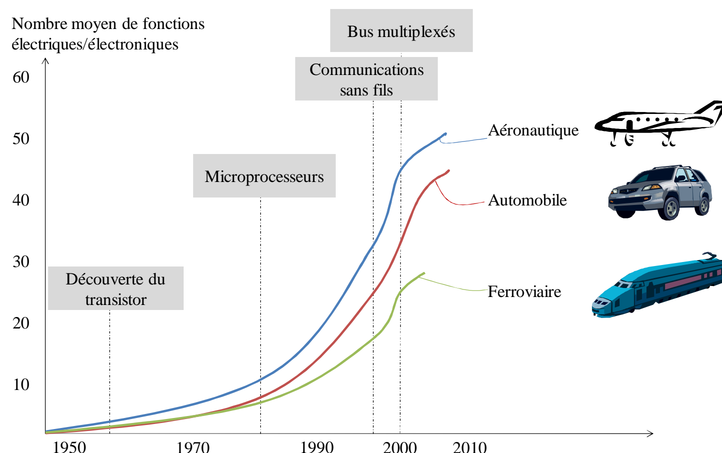

Figure 28. Evolution du nombre de fonctions électriques dans le temps

Dans ce chapitre, l'étude de cas concerne les systèmes embarqués automobile. Dans ce domaine, l'électronique et les logiciels se sont hybridés avec les composants mécaniques ou hydrauliques, afin d'en augmenter l'efficacité ou la réactivité, tout en diminuant souvent les coûts. Progressivement, d'autres fonctions plus évoluées pour l'assistance avancée à la

[3] RENAULT TRUCKS : groupe Volvo AB, assemble et vend des véhicules industriels et utilitaires.

conduite (*Advanced Driving Aid Systems*) ont vu le jour, allant de fonctions simples comme le ABS, le ESP, jusqu'à des fonctions plus élaborées comme le régulateur de vitesse avec radar, voire même la détection et évitement de piéton. Ainsi, de nombreuses fonctions ou composants des systèmes automobiles sont pilotés électroniquement ; l'architecture informatique matérielle s'articule autour de calculateurs (appelés aussi *E.C.U Electronic Control Unit*). Les calculateurs se composent généralement d'un microprocesseur/microcontrôleur, une mémoire et des périphériques d'entrées/sorties [73]. Un calculateur n'est aujourd'hui plus un élément isolé, il exploite les signaux provenant des capteurs placés à divers endroits et dans différents composants du véhicule (Figure 29).

Figure 29. Architecture interne d'un ECU

Tous les calculateurs sont interconnectés entres eux directement ou indirectement par un bus de communication (Figure 30).

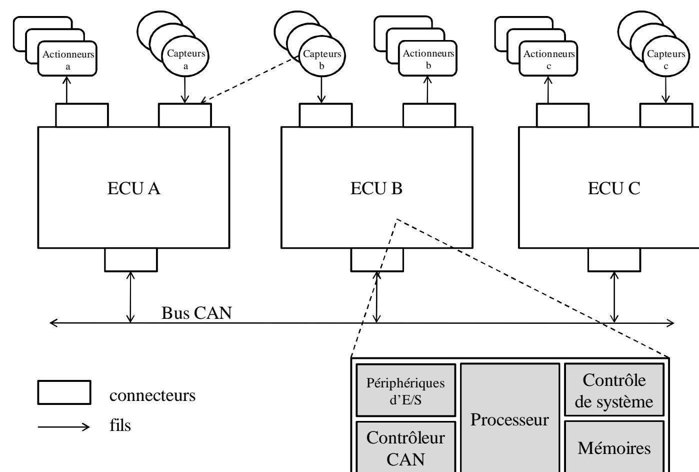

Figure 30. Liaisons entre capteurs, ECUs et actionneurs

Celui-ci est physiquement constitué par des conducteurs électriques (2 fils). Les informations (messages) y circulent avec une vitesse de transmission plus ou moins rapide en fonction du type de bus utilisé. Par exemple le bus CAN (*Controller Area Network*) J1939 (haut débit), est fréquemment utilisé dans les automobiles et les camions. Les informations circulant sur ce bus permettent la gestion en temps réel des fonctionnalités partagées. Ils existent aussi d'autres types de bus utilisés dans le domaine automobile tels que le bus LIN et le bus FlexRay [74]. Le bus de communication permet également l'interfaçage avec les capteurs et les actionneurs.

L'architecture embarquée automobile à laquelle nous nous intéressons pour réaliser l'étude de cas, est basée sur le bus CAN. Le bus CAN utilise la technique de multiplexage qui consiste à raccorder à un même câble un grand nombre de calculateurs qui interagissent entre eux suivant un protocole de communication série bien déterminé [75] [76] (Annexe J).

L'introduction des bus multiplexés (principalement le bus CAN) dans l'automobile, avait pour objectif de réduire la longueur de câbles dans les véhicules ainsi que de réduire leurs coûts élevés en masse, en matériaux et en main d'œuvre pour l'assemblage. Néanmoins, ceci a contribué à l'augmentation du nombre de calculateurs pour l'encodage/décodage et la mise en forme des trames de données du bus CAN.

III. La fonction SDK

1. Aperçu sur la fonction SDK

La fonction SDK est une fonction qui a fait partie du projet européen « CHAUFFEUR» réalisé par RENAULT TRUCKS et qui a été analysé ensuite dans le projet ANR-Predit « DIAFORE » [77] [78].
La fonction SDK, utilisée sur les routes, assiste le conducteur de camion à gérer son environnement longitudinal : elle l'aide à conserver automatiquement une distance de sécurité avec les véhicules qui le précèdent. Cette distance est proportionnelle à sa vitesse et lui assure deux secondes de réflexion dans le cas où des changements brusques se manifestent devant lui. La fonction SDK est le nom donné à la fonction AAC (*Adaptative Cruise Control*) avancée. SDK peut être définie ainsi comme un régulateur de vitesse qui prend en considération la dynamique relative des véhicules.

Pour réaliser la fonction SDK, un radar installé à la façade du véhicule, observe d'une façon continue l'espace devant le véhicule et fournit à l'ECU auquel il est connecté, la valeur de la distance entre le véhicule équipé par SDK et le véhicule qui le précède, la vitesse relative entre les deux véhicules et le type de l'objet détecté (Figure 31).

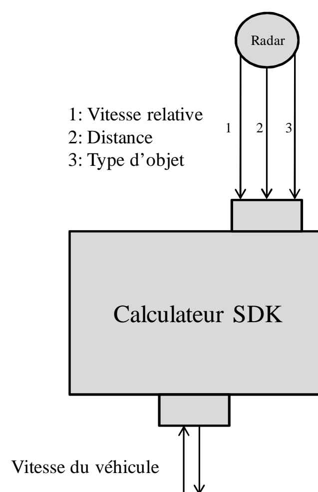

Figure 31. Valeurs issues du radar

Il s'agit d'un radar à effet doppler, fabriqué par A.D.C (Automotive Distance Control) de type ARS 100 et de bande de fréquence 76-77GHz. La fonction SDK utilise en plus, la valeur de la vitesse du véhicule équipé par la fonction pour effectuer le calcul nécessaire. Ensuite, un calcul de régulation interne effectué par un des ECUs permet de gérer automatiquement les actionneurs du véhicule comme le frein-moteur, le moteur et la boîte de vitesse. La fonction SDK permet l'intervention du chauffeur : à tout instant, ce dernier peut prendre en charge le contrôle du véhicule via les commandes conventionnelles de conduite.

La fonction SDK prend en considération plusieurs scenarii qui couvrent toutes les possibilités pouvant se présenter dans l'environnement en face du véhicule. Dès qu'une modification dans l'environnement devant le véhicule est détectée, ce dernier doit adapter sa vitesse suivant celles des véhicules en avant pour maintenir la distance de sécurité. Parmi les scénarii auxquels peut s'adapter la fonction SDK, nous citons :

- La détection d'un véhicule sur la trajectoire actuelle (Figure 32)
- La détection d'un véhicule qui provient d'une voie d'accélération (Figure 33)
- La détection des véhicules dans un trafic dense (Figure 34)

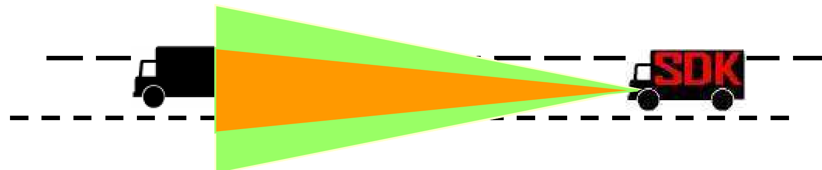

Figure 32. Détection d'un véhicule dans le chemin actuel

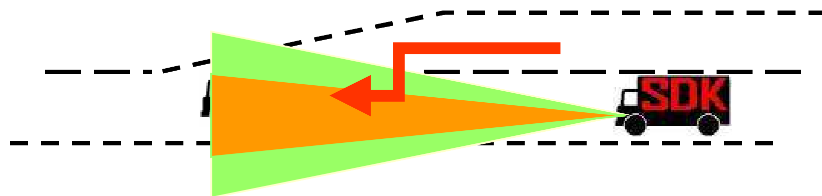

Figure 33. Détection d'un véhicule qui provient de la voie de coté

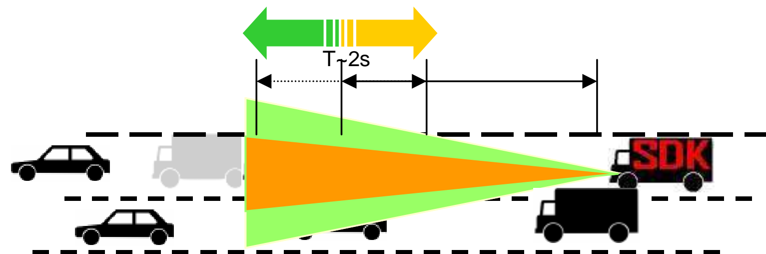

Figure 34. Trafic dense

Dans la suite nous allons présenter une modélisation de la fonction SDK. Nous avons simplifié l'algorithme de la fonction puisque le but n'est pas de concevoir complètement cette fonction, ni de vérifier les calculs mais plutôt de l'utiliser pour tester l'analyse de la diagnosticabilité fonctionnelle-architecturale. Ainsi, notre modèle prend comme entrée une seule valeur du radar au lieu de trois, ne tient pas compte du contrôle des freins ni de la boite de vitesse. Il régule simplement la puissance du moteur du véhicule ; la vitesse lui sera proportionnelle (Figure 35).

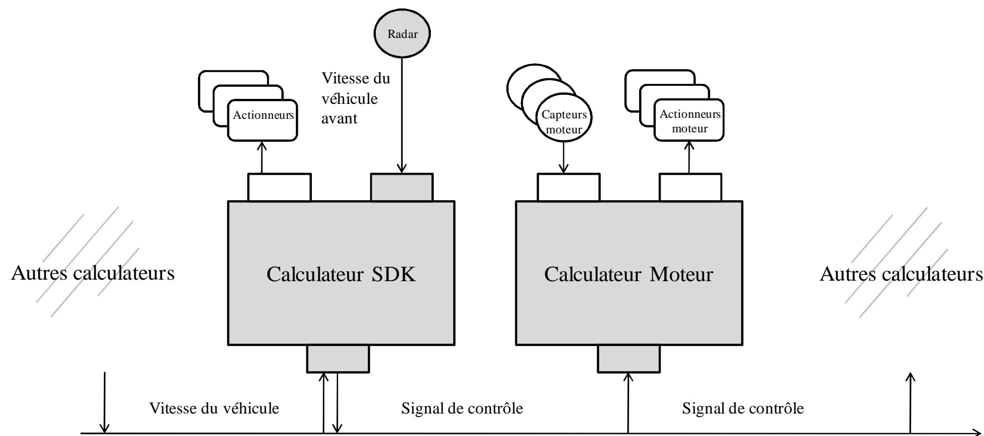

Figure 35. Composants de l'architecture électronique participants à la SDK

2. Le modèle mathématique de la fonction SDK

Le radar mesure la distance qui sépare les deux véhicules en émettant une courte impulsion de signal radio, et mesure le temps que prend l'onde pour revenir après avoir été réfléchie. La distance est la moitié du temps de retour de l'onde multipliée par la vitesse du signal (qui est proche de la vitesse de la lumière dans le vide si le milieu traversé est l'atmosphère). Ainsi le temps de retour varie selon la variation de la distance qui sépare les deux véhicules, plus la distance est courte, plus le retour de signal est rapide.

vitesse_relative(t)=vitesse_vehicule_avant(t) – vitesse_vehicule_arriere(t)

A chaque retour du signal, nous obtenons la valeur de la vitesse relative ainsi que la distance relative entre les deux véhicules.

distance_relative(t2)=

distance_relative(t1) + (t2-t1)×(vitesse_vehicule_avant – vitesse_vehicule_arriere)

t1et t2 sont en secondes

La distance de sécurité dépend de la vitesse du véhicule suiveur, c'est la distance nécessaire pour que le conducteur du véhicule d'arrière ait deux secondes de réflexion afin d'éviter tout risque dans son environnement frontal. Ainsi à chaque pas de calcul, la fonction SDK doit obtenir la distance de sécurité selon la formule suivante :

distance_securite = 2 × vitesse_vehicule_arriere × (1000/3600)

N.B : La vitesse étant en km/h, le facteur 1000/3600 est nécessaire pour avoir la distance en mètres.

Selon la différence entre la distance relative et la distance de sécurité, la fonction SDK doit réduire ou augmenter la puissance du moteur du véhicule et par la suite agir sur sa vitesse. En

effet, pour notre modèle simple nous considérons que la vitesse est directement proportionnelle à la puissance du moteur donnant les deux dernières équations du modèle :

$$\textit{Puissance}(t2) = \textit{puissance}\ (t1) \times \left(1 + \frac{distance_relative - distance_securite}{distance_securité}\right)$$

$$\textit{Vitesse_vehicule_arriere} = \textit{puissance} \times (150/100)$$

N.B. La puissance du moteur étant exprimée en pourcentage, le facteur 150/100 provient de l'application d'une règle de trois qui fait correspondre à une puissance maximale de 100% une vitesse maximale de 150km/h.

IV. Analyse de la diagnosticabilité fonctionnelle-architecturale de SDK

1. Analyse de la diagnosticabilité fonctionnelle de SDK

Notre approche de vérification de diagnosticabilité d'architecture nécessite d'abord la construction du modèle fonctionnel du système (sous forme d'automates), ensuite la construction d'un modèle de description de l'architecture matérielle (en AADL). La troisième étape de l'approche est la vérification des propriétés de l'architecture et la dernière étape et la vérification des propriétés de l'interaction fonctions/architecture.
Nous utiliserons l'approche du « diagnostiqueur » [48] pour l'analyse de diagnosticabilité des systèmes à évènements discrets. Ainsi, nous modélisons la fonction SDK à l'aide d'un ensemble d'automates finis, de la même manière que pour l'exemple de PVC dans le chapitre précédent. Un partitionnement de l'étude de la diagnosticabilité du système doit être fait afin de cerner l'ensemble des composants à analyser. Les autres composants en interactions peuvent faire partie d'une autre analyse de diagnosticabilité chacun par exemple une analyse de la diagnosticabilité de la fonction moteur, ou de la fonction freinage, …etc. Pour l'exemple de la SDK comportant les composants suivant : le radar, les calculateurs, la valeur de la vitesse du véhicule et le moteur, les événements « fautes » attendus sont les suivants :

- « Panne radar » : F1 qui mène à l'état « Panne 1»
- « Panne mesure vitesse » : F2 qui mène à l'état « Panne 2» (Figure 36).

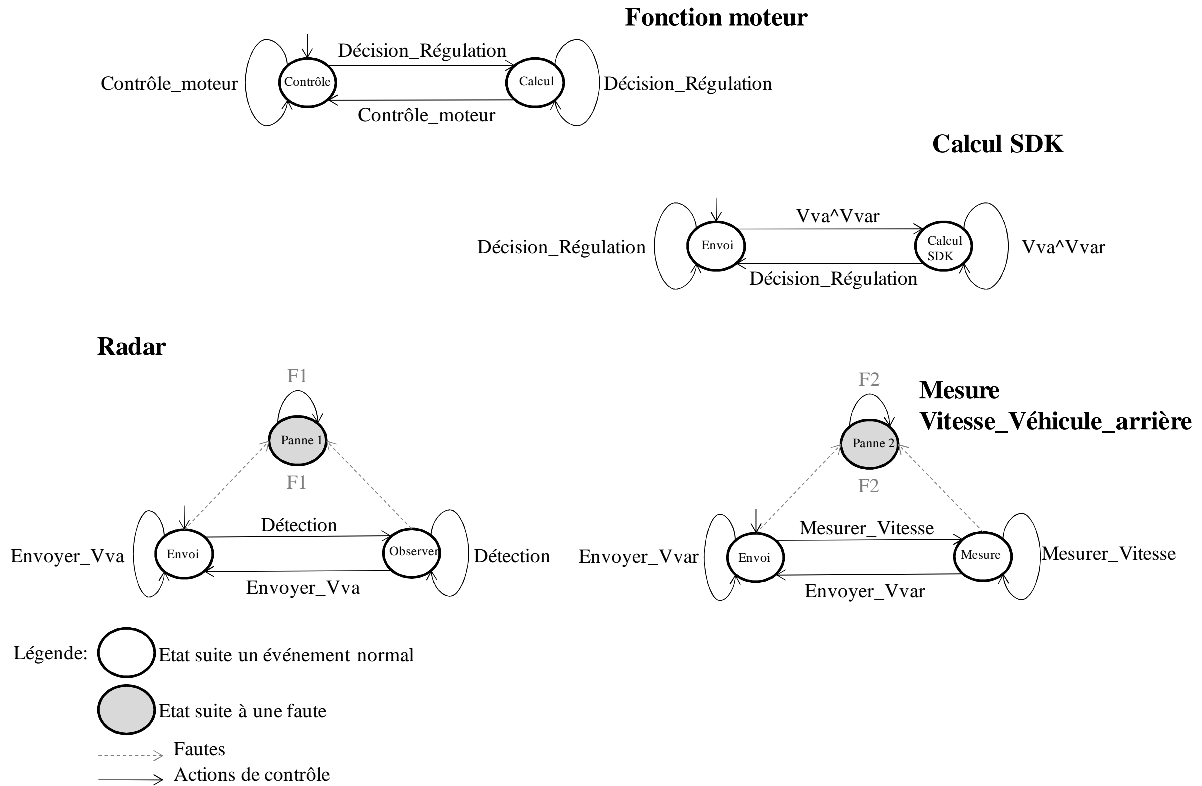

Figure 36. Modèle de comportement des composants de la fonction SDK
Vvar : Vitesse véhicule arrière
Vva : Vitesse véhicule avant

Les contrôleurs doivent ainsi contrôler l'ensemble des événements du système (Figure 37):

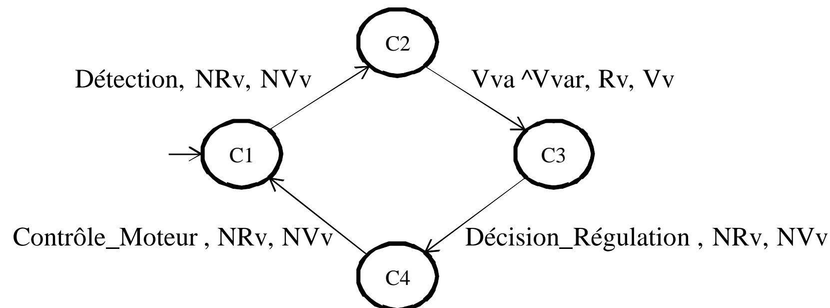

Figure 37. Automate fini du comportement des calculateurs

Ensuite les déviations du comportement nominal font partie du mode dégradé, qui à son tour doit être modélisé (Figure 38).

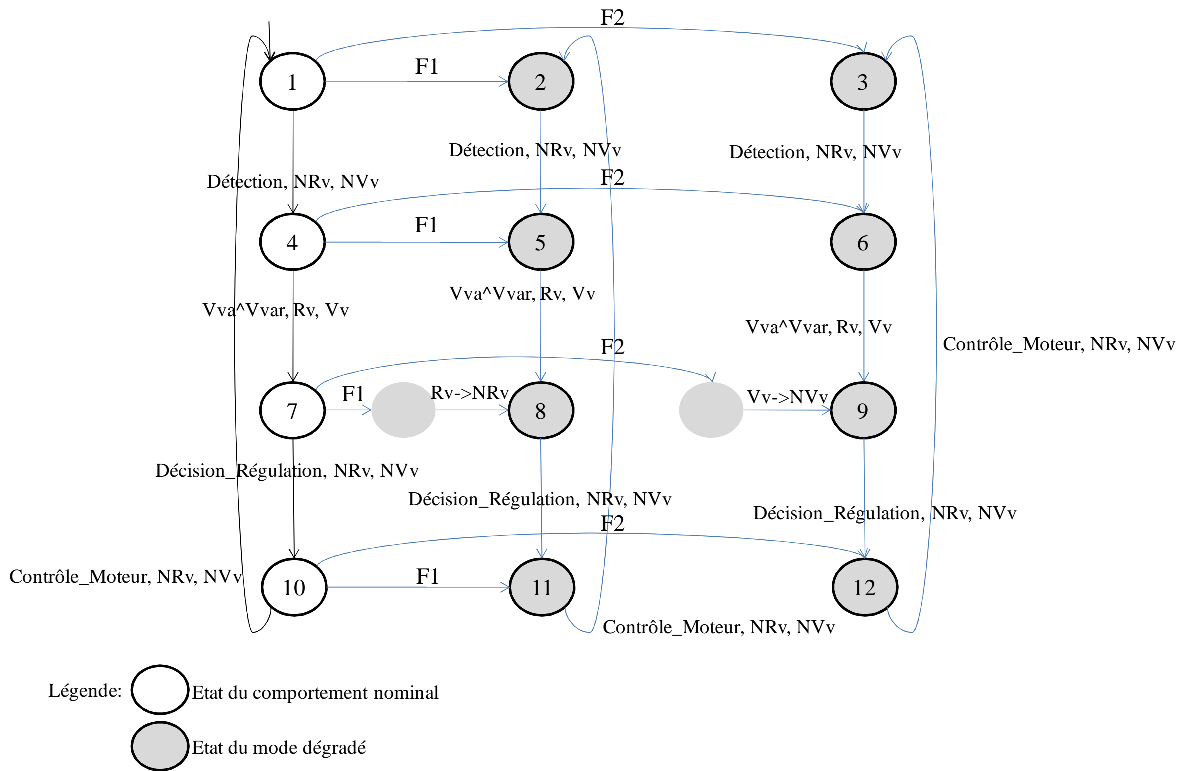

Figure 38. Automate fini complet de la fonction SDK

Capteur de valeur radar :
-Rv : Radar Value
-NRv : No Radar Value

Capteur de valeur Vvar :
-Vv : Velocity Value
-NVv :No Velocity Value

Par la suite, à partir du modèle global du système, l'approche de M. Sampath génère un « Observateur » à partir des séquences d'événements observables du système (Figure 39).
Par exemple dans le cas de SDK, O= <Détection,NRv,NVv>, <Vva^Vvar,Rv,Vv> est une séquence d'événements observables, elle est possible dans les deux cas de fonctionnement nominal et dégradé. Ensuite, l'expression de cette séquence d'événements observables en termes d'états, produit un ensemble Δ(O) de toutes les possibilités : Δ(O)= {(1,4,7) (1,4,7,8) (1,2,5,8) (1,4,5,8) (1,4,7,9)}.

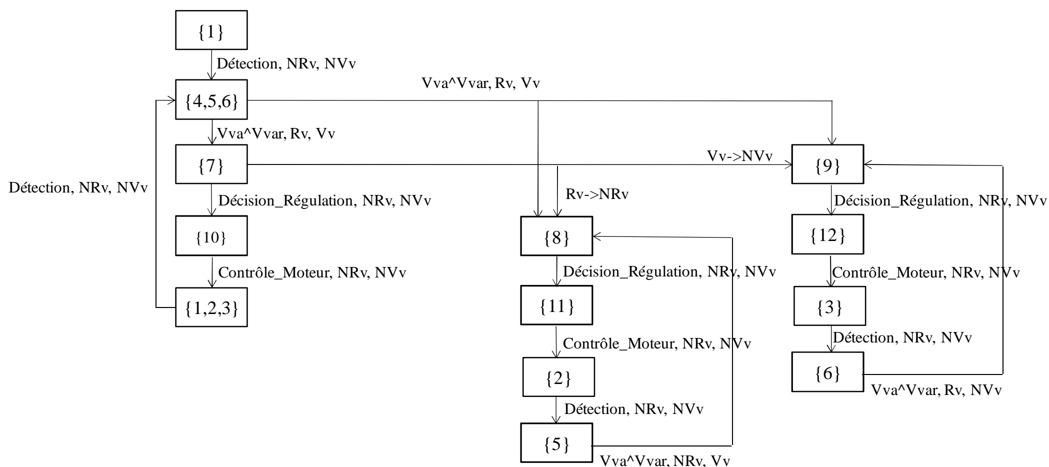

Figure 39. Observateur des événements de la fonction SDK

A la fin, vient la dernière étape qui est la réalisation du « diagnostiqueur » (Figure 40).

Un état (ou un nœud) dans le diagnostiqueur peut être :

- N (Normal): si tous ses candidats sont étiquetés N
- F*i* certain : si tous ses candidats sont étiquetés F*i*
- F*i* incertain : si autre

Dans le cas du système SDK, il existe deux fautes possibles :

- F1 : Panne Radar
- F2 : Panne Mesure_Vitesse

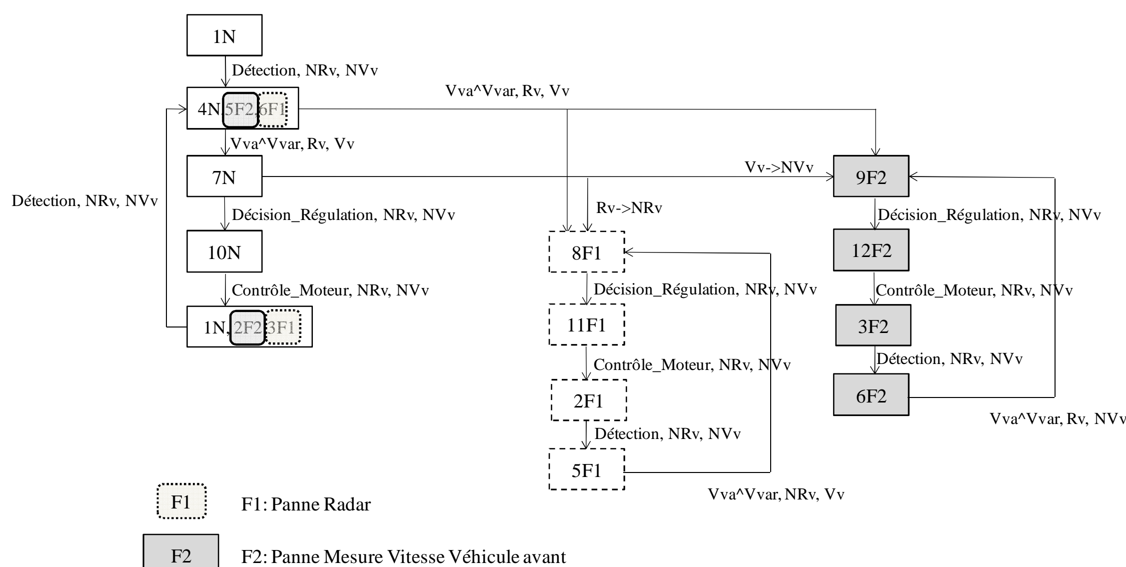

Figure 40. Diagnostiqueur de la fonction SDK

D'après l'approche suivie, un système à événements discrets est diagnosticable si et seulement si son diagnostiqueur ne contient pas des cycles indéterminés. Un cycle est indéterminé si :

- Il existe un cycle d'états F*i* incertains dans le diagnostiqueur
- et les états correspondants, du cycle des Fi incertains, forment aussi un cycle observable dans le modèle global du système.

Dans le cas du système SDK, il n'existe que deux états incertains (4N, 5F2, 6F1) et (1N, 2F2, 3F1) mais ne forment pas un cycle d'états F*i* incertains dans le diagnostiqueur. Ainsi, le système SDK est diagnosticable.

2. Description de l'architecture embarquée

Dans ce paragraphe nous présentons une description de l'architecture utilisée pour la fonction SDK. Cette description faite en AADL, montre les détails de chaque composant de l'architecture ainsi que leurs interactions.

Le but de cette étape est de réaliser un modèle simple, complémentaire au modèle réalisé avec des automates finis dans le paragraphe précédent.

Ainsi, la description de l'architecture en AADL est la suivante :

```
---------------------------------------------------------------------------
----
-- Definition des composants des calculateurs -----------------------------
----
---------------------------------------------------------------------------
----

processor MPC563
  features
    bus01: requires bus access membus.impl1;
    bus_can : requires bus access CAN.impl;
end MPC563;

processor MPC555
  features
    bus02: requires bus access membus.impl2;
    bus_can : requires bus access CAN.impl;
end MPC555;

processor implementation MPC563.Motorola
properties
Clock_Period => 25 ns;
end MPC563.Motorola;

processor implementation MPC555.Motorola
properties
Clock_Period => 25 ns;
end MPC555.Motorola;

memory RAM1
  features
    bus01: requires bus access membus.impl1;
    bus_can : requires bus access CAN.impl;
end RAM1;


memory RAM2
  features
    bus02: requires bus access membus.impl2;
    bus_can : requires bus access CAN.impl;
end RAM2;

memory implementation RAM1.Motorola_RAM
end RAM1.Motorola_RAM;

memory implementation RAM2.Motorola_RAM
end RAM2.Motorola_RAM;

bus membus
end membus;

bus implementation membus.impl1
end membus.impl1;

bus implementation membus.impl2
end membus.impl2;
```

```
----------------------------------------------------------------------------
----
-- Definition des calculateurs ---------------------------------------------
----
----------------------------------------------------------------------------
----

system Calculateur_radar
  features
  interface : requires bus access CAN.impl;
end Calculateur_radar;

system implementation Calculateur_radar.imp
  subcomponents
    HSRAM1: memory RAM1.Motorola_RAM;
    Proc : processor MPC563.Motorola {Clock_Period => 25 ns;};
      high_speed_bus : bus membus.impl1;
  connections
  bus access high_speed_bus -> proc.bus01;
  bus access interface -> proc.bus_can;
  bus access interface -> HSRAM1.bus_can;
end Calculateur_radar.imp;


system Calculateur_moteur
 features
 interface : requires bus access CAN.impl;
end Calculateur_moteur;

system implementation Calculateur_moteur.imp
  subcomponents
    HSRAM2: memory RAM2.Motorola_RAM;
    Proc : processor MPC555.Motorola{Clock_Period => 25 ns;};
    high_speed_bus : bus membus.impl2;
  connections
  bus access high_speed_bus -> proc.bus02;
  bus access interface -> proc.bus_can;
  bus access interface -> HSRAM2.bus_can;
end Calculateur_moteur.imp;



----------------------------------------------------------------------------
----
-- Definition du bus CAN ---------------------------------------------------
----
----------------------------------------------------------------------------
----

bus CAN
  properties
    -- Ce bus permet de raccorder des equipements
    Allowed_Access_Protocol => Device_Access;
    Allowed_Connection_Protocol => Data_Connection;
    Allowed_Message_size => 40b..40b;
    SEI ::BandWidthCapacity => 1.0 Mbps;
end CAN;

bus implementation CAN.impl
end CAN.impl;
```

```
---------------------------------------------------------------------------
----
-- Definition des données -------------------------------------------------
----
---------------------------------------------------------------------------
----

data trame
      properties
      Source_Data_Size => 40b;
end trame;

data state_or_event_variable
      properties
      Source_Data_Size => 1b;
end state_or_event_variable;


---------------------------------------------------------------------------
----
-- Definition des périphériques -------------------------------------------
----
---------------------------------------------------------------------------
----

device speed_sensor
  features
    CAN_interface : requires bus access CAN.impl;
    output_vvar : out data port ;
      flows
      Flow1: flow source output_vvar;
end speed_sensor;


device radar
  features
    CAN_interface : requires bus access CAN.impl;
    output_vva : out data port ;

      flows
      Flow1: flow source output_vva;
end radar;

device engine_actuator
  features
    CAN_interface : requires bus access CAN.impl;
    input_new_speed : in data port ;
      flows
      Flow1: flow sink input_new_speed;
end engine_actuator;



---------------------------------------------------------------------------
----
-- Definition des processus et des threads --------------------------------
----
---------------------------------------------------------------------------
----
```

```
process processus_radar
features
input_vva : in data port trame;
output_vva : out data port;
output_RV_diag : out data port state_or_event_variable;
output_detection_diag : out data port state_or_event_variable;
e1: out event port;
flows
flow1 : flow path input_vva -> output_vva;
RV : flow source output_RV_diag;
Detection : flow source output_detection_diag;
end processus_radar;


process implementation processus_radar.impl
subcomponents
donnee : data trame;
detection_data : data state_or_event_variable;
RV_data : data  state_or_event_variable;
properties
Period => 25 ns;
end processus_radar.impl;


process processus_sdk
features
input_vva : in data port;
input_vvar : in data port trame;
output_new_speed : out data port trame;
output_VV_diag : out data port state_or_event_variable;
output_VVA_AND_VVAR_diag : out data port state_or_event_variable;
e1: in event port;
flows
flow1 : flow path input_vva -> output_new_speed;
flow2 : flow path input_vvar -> output_new_speed;
VV : flow source output_VV_diag;
VVA_AND_VVAR : flow source output_VVA_AND_VVAR_diag;
end processus_sdk;


process implementation processus_sdk.impl
subcomponents
donnee : data trame;
VV_data : data  state_or_event_variable;
vva_and_vvar_data : data  state_or_event_variable;
Decision_de_regulation_data : data  state_or_event_variable;
end processus_sdk.impl;


process processus_moteur
features
input_new_speed : in data port trame;
output : out data port trame;
output_Decision_de_regulation_diag : out data port state_or_event_variable;
output_controle_moteur_diag : out data port state_or_event_variable;
flows
flow1 : flow path input_new_speed -> output;
Decision_de_regulation : flow source output_Decision_de_regulation_diag;
Controle_moteur : flow source output_controle_moteur_diag;
end processus_moteur;
```

```
process implementation processus_moteur.impl
subcomponents
controle_moteur_data : data  state_or_event_variable;
properties
Period => 25 ns;
end processus_moteur.impl;


process processus_diagnostiqueur
features
input_RV_diag : in data port state_or_event_variable;
input_VV_diag : in data port state_or_event_variable;
input_detection_diag : in data port state_or_event_variable;
input_VVA_AND_VVAR_diag : in data port state_or_event_variable;
input_Decision_de_regulation_diag : in data port state_or_event_variable;
input_controle_moteur_diag : in data port state_or_event_variable;

flows
RV : flow sink input_RV_diag;
VV : flow sink input_VV_diag;
Detection : flow sink input_detection_diag;
VVA_AND_VVAR : flow sink input_VVA_AND_VVAR_diag;
Decision_de_regulation : flow sink input_Decision_de_regulation_diag;
Controle_moteur : flow sink input_controle_moteur_diag;
end processus_diagnostiqueur;


process implementation processus_diagnostiqueur.impl
end processus_diagnostiqueur.impl;


---------------------------------------------------------------------------
----
-- Definition du systeme complet ------------------------------------------
----
---------------------------------------------------------------------------
----

system systeme_complet
end systeme_complet;

system implementation systeme_complet.impl
  subcomponents
    ECU1: system Calculateur_radar.imp;
    ECU2: system Calculateur_moteur.imp;
    speed_sensor : device speed_sensor;
    engine_actuator : device engine_actuator;
    radar : device radar;
    reseau_CAN : bus CAN.impl;
    p_radar : process processus_radar.impl;
    p_sdk : process processus_sdk.impl;
    p_diag : process processus_diagnostiqueur.impl;
    p_moteur : process processus_moteur.impl;

      connections
      -- en entrées/sorties
      cnx0 : data port radar.output_vva -> p_radar.input_vva
      {Actual_Connection_Binding => reference reseau_CAN;};
      cnx1 : data port p_radar.output_vva -> p_sdk.input_vva      ;
      cnx2 : data port speed_sensor.output_vvar -> p_sdk.input_vvar
      {Actual_Connection_Binding => reference reseau_CAN;};
```

```
cnx3 : data port p_sdk.output_new_speed -> p_moteur.input_new_speed
{Actual_Connection_Binding => reference reseau_CAN;};
cnx4: data port p_moteur.output -> engine_actuator.input_new_speed
{Actual_Connection_Binding => reference reseau_CAN;};

-- vers le processus diagnostiqueur
cnx5: data port p_radar.output_RV_diag -> p_diag.input_RV_diag;
cnx6: data port p_sdk.output_VV_diag -> p_diag.input_VV_diag;
cnx7: data port p_radar.output_detection_diag ->
p_diag.input_detection_diag;
cnx8: data port p_sdk.output_vva_and_vvar_diag ->
p_diag.input_vva_and_vvar_diag;
cnx9: data port p_moteur.output_Decision_de_regulation_diag ->
p_diag.input_Decision_de_regulation_diag
{Actual_Connection_Binding => reference reseau_CAN;};
cnx10: data port p_moteur.output_controle_moteur_diag ->
p_diag.input_controle_moteur_diag
{Actual_Connection_Binding => reference reseau_CAN;};

-- en évenements
cnxe1: event port p_radar.e1 -> p_sdk.e1;


-- Connexions au bus CAN
bus access reseau_CAN -> ECU1.interface;
bus access reseau_CAN -> ECU2.interface;
bus access reseau_CAN -> speed_sensor.CAN_interface;
bus access reseau_CAN -> engine_actuator.CAN_interface;

-- Flux de données
flows
Flux1: end to end flow radar.Flow1-> cnx0 -> p_radar.Flow1 -> cnx1 ->
p_sdk.flow1 -> cnx3 -> p_moteur.flow1-> cnx4 ->
engine_actuator.Flow1;
Flux2: end to end flow speed_sensor.Flow1 -> cnx2 -> p_sdk.flow2 ->
cnx3 -> p_moteur.flow1 -> cnx4 -> engine_actuator.Flow1;
Message1: end to end flow p_radar.RV-> cnx5 -> p_diag.RV;
Message2: end to end flow p_sdk.VV-> cnx6 -> p_diag.VV;
Message3: end to end flow p_radar.Detection-> cnx7 ->
p_diag.Detection;
Message4: end to end flow p_sdk.vva_and_vvar-> cnx8 ->
p_diag.vva_and_vvar;
Message5: end to end flow p_moteur.Decision_de_regulation -> cnx9 ->
p_diag.Decision_de_regulation;
Message6: end to end flow p_moteur.Controle_moteur -> cnx10 ->
p_diag.Controle_moteur;

-- Liaisons
properties
Actual_processor_Binding => reference ECU1.proc applies to p_radar; -
- mapping du processus p_radar sur le processeur
Actual_memory_Binding => reference ECU1.HSRAM1 applies to p_radar; --
mapping du processus p_radar sur la mémoire
Actual_processor_Binding => reference ECU1.proc applies to p_sdk; --
mapping du processus p_sdk sur le processeur
Actual_memory_Binding => reference ECU1.HSRAM1 applies to p_diag; --
mapping du processus p_diag sur la mémoire
Actual_processor_Binding => reference ECU1.proc applies to p_diag; --
mapping du processus p_diag sur le processeur
Actual_memory_Binding => reference ECU1.HSRAM1 applies to p_sdk; --
mapping du processus p_sdk sur la mémoire
```

```
    Actual_processor_Binding => reference ECU2.proc applies to p_moteur;
    -- mapping du processus p_moteur sur le processeur
    Actual_memory_Binding => reference ECU2.HSRAM2 applies to p_moteur; -
    - mapping du processus p_moteur sur la mémoire

end systeme_complet.impl;
```

Les messages entre les différents composants de l'architecture reflètent une certaine vue comportementale sur l'architecture. Ils sont décrits ci-après en pseudo-code, il s'agit de :

- Deux messages contenant les valeurs des variables d'état,

```
Message_1 :
Source : Calculateur_Radar
Destination : Calculateur_Radar
Contenu : Valeur(RV)

Message_2 :
Source : Calculateur_Radar
Destination : Calculateur_Radar
Contenu : Valeur (VV)
```

- Ainsi que quatre messages pour notifier les événements « détection », « VVa^VVar », « Décision_de_régulation » et « contrôle_Moteur »:

```
Message_3 :
Source : Calculateur_Radar
Destination : Calculateur_Radar
Contenu : Valeur (Détection)

Message_4 :
Source : Calculateur_Radar
Destination : Calculateur_Radar
Contenu : Valeur (VVa ^ VVar)

Message_5 :
Source : Calculateur_Moteur
Destination : Calculateur_Radar
Contenu : Valeur (Décision de régulation)

Message_6 :
Source : Calculateur_Moteur
Destination : Calculateur_Radar
Contenu : Valeur(Contrôle_Moteur)
```

Le diagnostic prévu suivant le cahier des charges (défini par des experts), est un diagnostic centralisé implanté sur l'ECU1 (RAM : 26Ko, ROM : 10Mo, Fréquence : 40MHz) à l'aide de la fonction « *p_Diag*» de taille 600Ko. Le WCET de « *p_Diag* » sur ECU1 représente 160 cycles d'horloge, correspondant à 4000ns. Le WCET de « *p_SDK* » sur ECU1 vaut 16 cycles d'horloge, correspondant à 400ns. La périodicité souhaitée pour éxecuter « *p_sdk* » ensuite « *p_diag* » sur ECU1 est de 4200ns.

3. Vérification des propriétés de la diagnosticabilité fonctionnelle-architecturale

Nous vérifions ci-après que la description du système répond aux propriétés de la diagnosticabilité fonctionnelle-architecturale, exigées au niveau de la description de l'architecture.

Soient les variables :
A : L'ensemble des calculateurs (ou composants) contenant une ou plusieurs fonctions à analyser -> A= {ECU1, ECU2}
B : L'ensemble des composants sources de variables d'état -> B = {ECU1}
C : L'ensemble de tous les composants calculateurs -> C= {ECU1, ECU2}
D_principal : Le calculateur (ou composant) diagnostiqueur -> D_principal= ECU1
E : L'ensemble des évènements à diagnostiquer -> E = {Détection, (Vva^Vvar), Décision_Régulation, Contrôle_Moteur}
F : L'ensemble de toutes les fonctions
F_Diag : L'ensemble des fonctions de diagnostic -> F_Diag= {p_Diag}
M : L'ensemble des messages envoyés contenant la valeur des variables d'état ou la notification des événements de l'automate représentant le système-> M= {Message_1, Message_2, Message_3, Message_4, Message_5, Message_6}
O : Le calculateur (ou composant) sur lequel est implanté le processus observateur
V : L'ensemble des variables d'états issues de capteurs ou autres composants du système -> V= {Rv, Vv}

Soit le type :
void : Type de retour non défini différent de null

Soient les fonctions :
F_Data : (x,y)→{0,1}\ (x,y) ∈ C^2 retourne 1 si x et y sont connectés par une connexion de type données, retourne 0 sinon
Size_F : x→y\ x ∈ F_Diag, y ∈ ℝ retourne la taille de x
Size_Mem : x→y\ x ∈ C, y ∈ ℝ retourne la taille mémoire de x
ΔT : x→y\ x ∈ F_Diag, y ∈ ℝ Fonction qui retourne la périodicité souhaitée pour l'exécution de la fonction x
WCET : (x,y)→z\ x ∈ F_Diag, y ∈ C, z ∈ ℝ retourne le temps maximal d'exécution que x peut prendre sur la plateforme y, correspond au WCET (W*orst-Case Execution Time*) de x sur y. Nous présumons l'existence de cette fonction.
Conn_I/O : (x,y)→z\ x∈ C, y ∈ B, z ∈{0,1} retourne 1 si x et y sont connectés
Implemented : (x,y)→z\x∈ F, y ∈ C, z ∈ {0,1}retourne 1 si x est implémentée sur y
Source : x→y\ x∈ M, y∈ C retourne le nom (ou adresse) du calculateur émetteur du message x.
Destination : x→y\ x∈ M, y∈ C retourne le nom (ou adresse) du calculateur récepteur du message x.
Valeur : x→y\ x∈ M, y∈ void retourne la valeur du message M.

Vérification de la Règle1 **:**

Comme la structure de diagnostic employée est centralisée, il faut vérifier les propriétés 1.1, 2, 3, 4, 5, 6 et 7

Vérification de la Propriété 1.1 : Connectivité-Diagnostiqueur

Nous avons :

ECU1 ∈ A, D_principal= ECU1

Ainsi :
Connectivité_Diagnostiqueur (ECU1)=1

-------------------***************-------------------*****************---------------------
Nous avons :

ECU2 ∈ A, F_Data (ECU2, D_principal) =1,

Ainsi :
Connectivité_Diagnostiqueur (ECU2) =1

-------------------***************-------------------*****************---------------------

Donc :
∀a ∈ A, Connectivité_Diagnostiqueur (a)=1

Vérification de la Propriété 2 : Exécutabilité diagnostiqueur

Nous avons :

F_Diag= p_diag,
Size_Mem (D_principal) = Size_Mem (ECU1) =10Mo,
Size_F (p_diag) =600Ko,
ΔT (p_diag) = 4200ns
WCET (p_Diag, ECU1) = 4000ns

Ainsi :

∀d ∈ D, ∀f ∈ F_Diag,

((Implemented (f,d)==1) ^ (Size_Mem (d)≥ Size_F (f)) ^ (ΔT (f) >= WCET (f,d)))

Ainsi :

Exécutabilité (f,d)=1

Vérification de la Propriété 3: Atteignabilité

Nous avons :
RV ∈ V,
Origine (RV)=ECU1 ∈ B,
Message_1 ∈ M,
Source(Message_1)=ECU1,
Destination(Message_1)=ECU1,
Contenu (Message_1)=Valeur(RV),
O= ECU1,

Alors :

(Conn_I/O(Origine(RV), O) ∨ F_Data (Origine(RV), O)) ^
Source (Message_1)=Origine(RV) ^
Destination(Message_1)=O ^ Contenu (Message_1) = Valeur(RV)

Ainsi :

Atteignabilité (RV)=1

-------------------********---------------*******--------------******----------------*********---

Nous avons :
VV ∈ V,
Origine (VV)=ECU1 ∈ B,
Message_2 ∈ M,
Source(Message_2)=ECU1,
Destination(Message_2)=ECU1,
Contenu (Message_2)=Valeur(VV),
O= ECU1,

Alors :
(Conn_I/O(Origine(VV),O) ∨ F_Data (Origine(VV), O)) ^
Source (Message_2)=Origine(VV) ^ Destination(Message_2)=O ^
Contenu (Message_2) = Valeur(VV)

Ainsi :

Atteignabilité (VV)=1

Vérification de la Propriété 4 : Accessibilité

Nous avons :

detection ∈ E,
Origine (detection)=ECU1 ∈ B,
Message_3 ∈ M,

Source(Message_3)=ECU1,
Destination(Message_3)=ECU1,
Contenu (Message_3)=Valeur (detection),
O= ECU1,

Alors :

(Conn_I/O(Origine(detection),O) ∨ F_Data(Origine(detection),O)) ^
Source(Message_3)=Origine(detection) ^ Destination(Message_3)=O ^
Contenu(Message_3)=Valeur (detection)

Ainsi :
Accessibilité (detection)=1

-----------------********-------------*******-------------******---------------*********---

Nous avons :

(VVa ^ VVar) ∈ E,
Origine ((VVa ^ VVar))=ECU1 ∈ B,
Message_4 ∈ M, Source(Message_4)=ECU1,
Destination(Message_4)=ECU1,
Contenu (Message_4)=Valeur ((VVa ^ VVar)),
O= ECU1,

Alors :

(Conn_I/O (Origine ((VVa ^ VVar)), O) ∨ F_Data (Origine ((VVa ^ VVar)), O)) ^ Source (Message_4)=Origine((VVa^VVar))^Destination(Message_4)=O ^
Contenu(Message_4)=Valeur ((VVa ^ VVar))

Ainsi :
Accessibilité ((VVa ^ VVar))=1

-----------------********-------------*******-------------******---------------*********---
Nous avons :

Décision_de_régulation ∈ E,
Origine (Décision_de_régulation)=ECU1 ∈ B,
Message_5 ∈ M,
Source(Message_5)=ECU1,
Destination(Message_5)=ECU1,
Contenu (Message_5)=Valeur(Décision_de_régulation),
O= ECU1,

Alors :

(Conn_I/O (Origine(Décision_de_régulation), O) ∨ F_Data (Origine(Décision_de_régulation), O)) ^ Source (Message_5)=Origine(Décision_de_régulation)^Destination(Message_5)=O ^

Contenu(Message_5)=Valeur(Décision_de_régulation)

Ainsi :
Accessibilité (Décision_de_régulation)=1

-----------------*******-------------*******-------------******--------------*********---

Nous avons :

Contrôle_Moteur ∈ E,
Origine (Contrôle_Moteur)=ECU1 ∈ B,
Message_6 ∈ M,
Source(Message_6)=ECU1,
Destination(Message_6)=ECU1,
Contenu (Message_6)=Valeur(Contrôle_Moteur),
O= ECU1,

Alors :

(Conn_I/O (Origine(Contrôle_Moteur), O) ∨ F_Data (Origine(Contrôle_Moteur), O)) ^
Source (Message_6)=Origine(Contrôle_Moteur)^Destination(Message_6)=O ^
Contenu(Message_6)=Valeur(Contrôle_Moteur)

Ainsi :
Accessibilité (Contrôle_Moteur)=1

Les propriétés 5 et 6 nécessitent une vérification qui tien compte du facteur temps et de l'avancement de l'état du système par rapport à ce facteur.

Vérification de la Règle 2:

Toute propriété de la diagnosticabilité fonctionnelle-architecturale est vérifiable grâce à la représentation ou la modélisation du système. Les propriétés {1.1, 1.2, 2, 3, 4} sont vérifiables au niveau de la description de l'architecture, alors que les propriétés {5 et 6} sont vérifiables au niveau de l'interaction des fonctions avec l'architecture.

4. Co-Modélisation et co-simulation matérielle-logicielle

a. Modélisation de SDK en Simulink

Le modèle Simulink est un modèle fonctionnel et doit être conforme au modèle à automates finis de la fonction. Il est constitué par des blocks Simulink et fait abstraction de l'architecture matérielle (Figure 41).

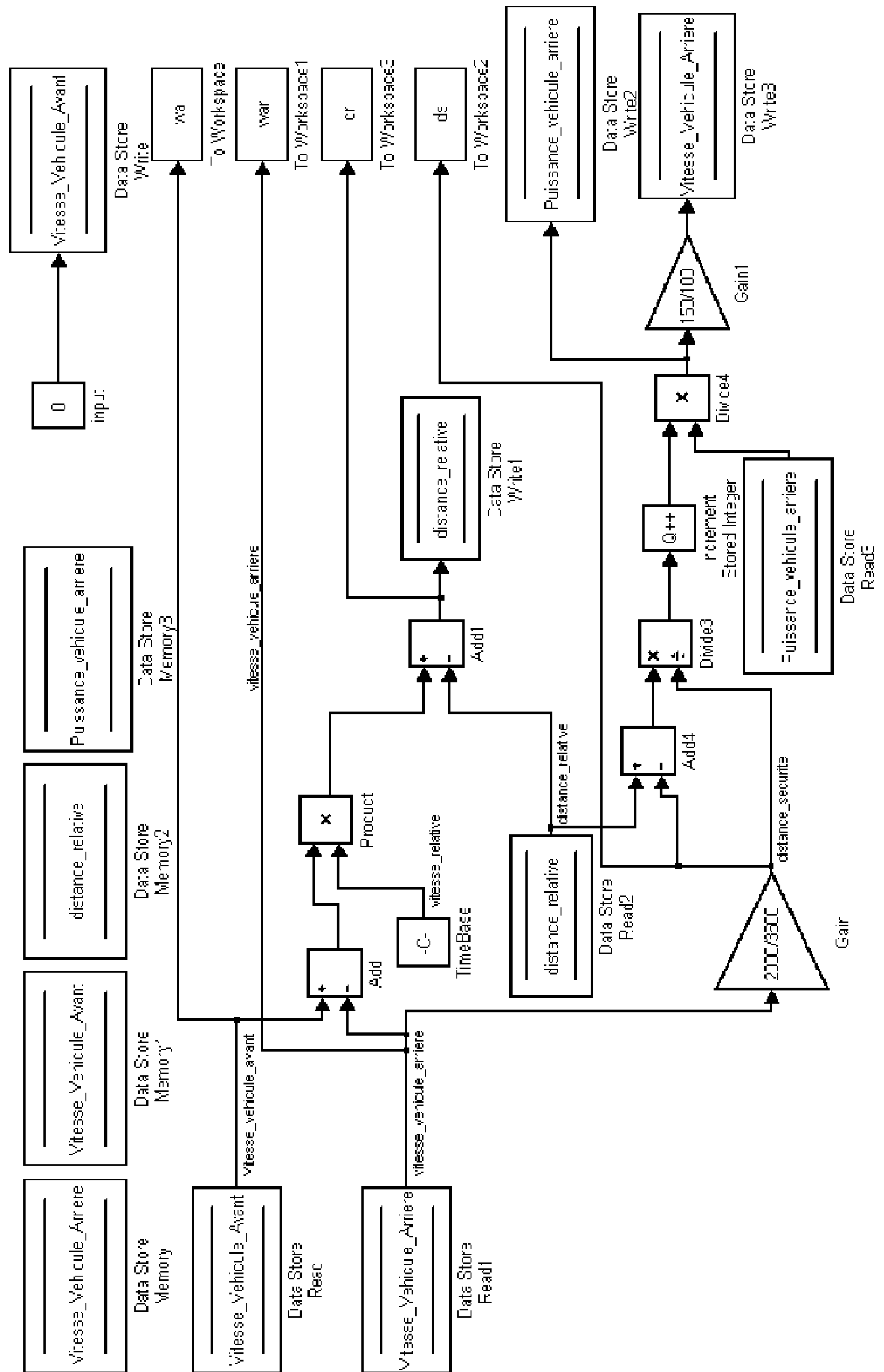

Figure 41. Modèle Simulink de la fonction SDK

Variables Simulink :

Vitesse_Vehicule_Avant
Vitesse_Vehicule_Arriere
Distance_Relative

Variables Matlab workspace :

vva, vvar, dr, ds

Au niveau de notre laboratoire, nous ne disposons pas d'informations suffisantes sur les caractéristiques du radar utilisé par la fonction SDK afin de simuler les valeurs que peut prendre les signaux « vitesse relative » et « distance relative », pour cela et afin de simplifier les calculs de la fonction SDK, cette dernière prend comme entrées la vitesse du véhicule avant, fournie par le block « Signal Builder » (Figure 42) sous forme d'un scénario prédéfini ou généré aléatoirement (Figure 43), et la vitesse du véhicule arrière (véhicule équipé par la fonction). Ensuite, l'algorithme de la fonction est appliqué à ces deux entrées et régule la puissance du moteur et donc la vitesse du véhicule suivant les valeurs des entrées. Le modèle Simulink doit décrire seulement le comportement nominal de la fonction sans tenir compte du déploiement prévu pour le diagnostiqueur.

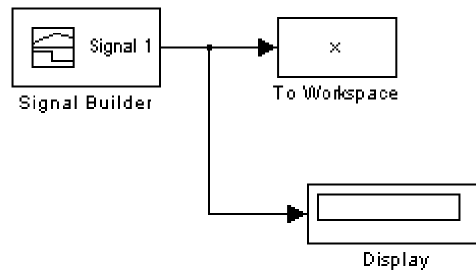

Figure 42. Block « Signal Builder » pour la génération des entrées de la fonction SDK

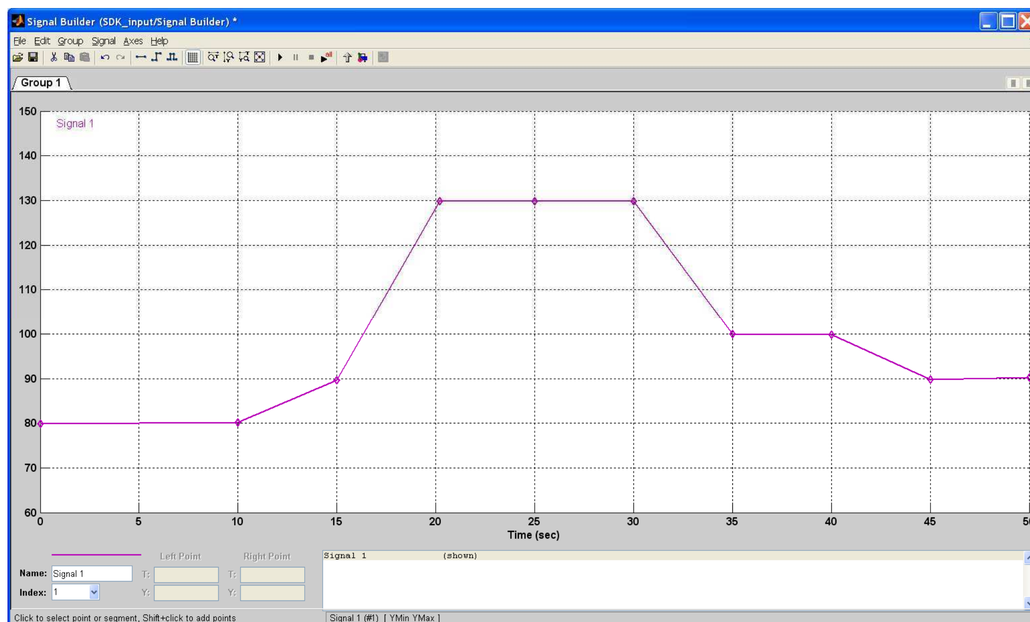

Figure 43. Exemple de scénario prédéfini pour les valeurs d'entrée de SDK

b. Modélisation des ECUs et du bus CAN en SystemC

Le modèle de l'architecture (Calculateurs et bus de communication) doit être conforme à la description de l'architecture faite précédemment et qui respecte déjà les propriétés que nous avons définies (sauf propriétés 5 et 6, car nous devons vérifier la faisabilité de l'insertion d'un observateur) pour la diagnosticabilité fonctionnelle/architecturale. Ainsi, il suffit de « convertir » la description du système en un modèle SystemC.
Chaque ECU dispose d'un port bidirectionnel dans chaque module. Il est utilisé pour envoyer des ordres sur le bus (demandes) et l'obtention de données et des informations du bus

(réponses). Une seule horloge est utilisée pour tous les processeurs ; si nous souhaitons augmenter le niveau de granularité, la précision du modèle de simulation est réglée sur le cycle d'horloge des ECUs. Il est important de noter que le protocole CAN complet est utilisé uniquement dans les modèles avec un haut niveau de granularité, exprimant les transactions entre calculateurs. Avec un niveau de granularité plus précis, le processeur et la mémoire de tous les modèles d'ECUs sont enveloppés dans des modules SystemC afin de communiquer avec d'autres dispositifs tels que les contrôleurs CAN.

Tout d'abord, nous modélisons l'architecture au niveau matériel. La Figure 44 rappelle comment est constituée cette architecture.

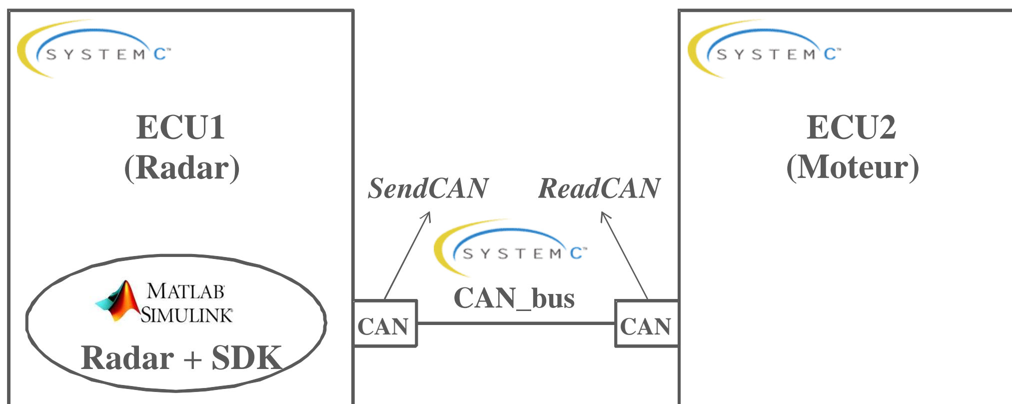

Figure 44. Aperçu du co-modèle SystemC-Simulink

Cette architecture se compose de deux modules : ECU1 et ECU2. Ces modules sont différents c'est à dire ils ne sont pas des instances d'un même module ECU : même s'ils sont identiques au niveau matériel, leur comportement fonctionnel est différent.
Nous avons créé les deux modules, ECU1 et ECU2, indépendamment l'un de l'autre. A chaque pas de simulation, la nouvelle valeur de la vitesse du véhicule d'avant est fournie par le modèle Simulink « *SDK_input* ». A chaque pas de la simulation, ce modèle envoie à SystemC, via la technique d'interfaçage déjà décrite, une nouvelle valeur de la variable « *vitesse_vehicule_avant* » suivant un signal prédéterminé.

Par contre, ECU2 doit, à chaque pas de simulation, lire du bus CAN la nouvelle valeur de la vitesse du véhicule arrière et l'utiliser par la suite pour commander les actionneurs du moteur. Dans le modèle SDK, nous relions les signaux vitesse_vehicule_avant, vitesse_vehicule_arriere, distance_securite et distance_relative à des blocs « *ToWorkspace* » appelés respectivement « *vva* », « *vvar* », « *ds* » et « *dr* » pour récupérer les valeurs de ces signaux depuis Matlab vers l'environnement SystemC.

Pour modéliser la communication entre les deux ECUs suivant le protocole CAN, nous avons attribué à chaque ECU un port d'entrée-sortie (sc_inout) et nous l'avons relié via un signal de type sc_uint<40>. En effet, nous avons adopté une trame CAN sur 40 bits formée d'un identificateur de 5 bits, d'un champ de 3 bits qui précise le nombre d'octets sur lesquels l'information envoyée s'étend, et un champ de 32 bits contenant cette information (Figure 45).

TRAME		
Identificateur	Longueur	Donnée
5 bits	3 bits	32bits

Figure 45. Format de la trame CAN

Cette trame CAN peut être envoyée ou reçue à l'aide de deux fonctions que nous avons développées :

- *La fonction SendCAN(sc_uint<5> id, sc_uint<3> length, sc_uint<8> *data)*
Il faut lui fournir les informations suivantes :

- l'identificateur qu'on attribue à la trame,
- la longueur de la donnée (en nombre d'octets),
- ainsi que la donnée sous forme d'un tableau de 4 champs de type sc_uint<8>. Cette fonction, concatène ces informations et forme ainsi la trame puis l'envoie sur le bus.

- *La fonction ReadCAN(sc_uint<5> *id, sc_uint<3> *length, sc_uint<8> *data)*
Il faut lui fournir la valeur de l'identificateur de la trame qu'on souhaite lire. Ainsi, cette fonction lit chaque trame se trouvant sur le bus et la décompose pour extraire ses différents champs. Si la trame lue possède l'identificateur demandé, les informations seront passées à un niveau plus haut.

Après la co-modélisation du module « *ECU1* » et celle du module « *ECU2* » de notre architecture (Annexe A), nous écrivons le « *sc_main* » dans lequel une horloge ainsi que les modules « *ECU1* » et « *ECU2* » sont instanciés, ensuite une connexion est établie entre eux une fois la simulation est lancée.
La fréquence de l'horloge dans cet exemple correspond à la puissance des horloges des deux microprocesseurs de « *ECU1* » et « *ECU2* », la même dans les deux calculateurs, égale à 40 MHz donc $40.\ 10^6$ tops (ou cycles) d'horloge par seconde. La durée d'un cycle est égale alors à : $1/40.10^6$ s = $0.025.10^{-6}$ s = 25 ns.
Nous avons fait le choix de simuler 4.10^8 cycles d'horloge dans une seule exécution du modèle, donc 0.1 ms du fonctionnement du système par exécution, afin de ne pas avoir des fichiers traces trop volumineux, difficiles à ouvrir. Ainsi, afin de simuler une heure il faut effectuer 36.10^6 simulations successives.

```
int sc_main(int argc, char *argv[])
{
	sc_report_handler::set_actions("/IEEE_Std_1666/deprecated",SC_DO_NOTH
ING);

	int exit;

	sc_clock clock("clock",25,SC_NS);
	sc_signal<sc_uint<40>> CAN_bus;

	ECU1 ECU1_block("input_block");
	ECU2 ECU2_block("sdk_block");

	ECU1_block.CAN(CAN_bus);
	ECU2_block.CAN(CAN_bus);
```

```
        ECU1_block.CLK(clock.signal());
        ECU2_block.CLK(clock.signal());


        sc_start(100000,SC_NS); );// 0.1 ms
        scanf("%d", &exit);
        return 0;
}
```

Le délai le plus long de l'exécution de la fonction « *SDK* » (ou WCET) affiché est égal à 16. Ainsi 16 cycles d'horloges sont nécéssaires pour une seule éxécution dans l'environnement Simulink/Matlab. Avec l'horloge utilisée de 40MHz, 16 cycles d'horloges correspondent à 16*25ns = 400ns.

c. Co-simulation du co-modèle et analyse des traces

COSITA lance la co-simulation. Ensuite, nous visualisons les résultats graphiquement via « *Matlab Command Window* »[4] la relation entre les variations des vitesses des deux véhicules (Figure 46) et la relation entre les variations de la distance de sécurité et de la distance relative entre les deux véhicules (Figure 47).

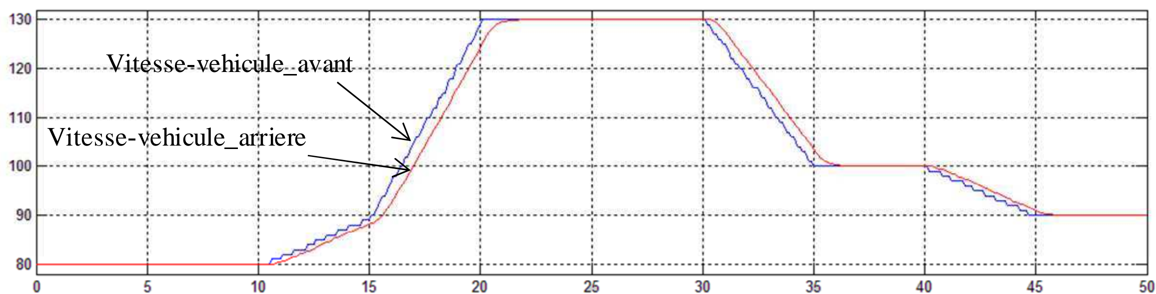

Figure 46. Vitesse du véhicule avant et vitesse du véhicule arrière

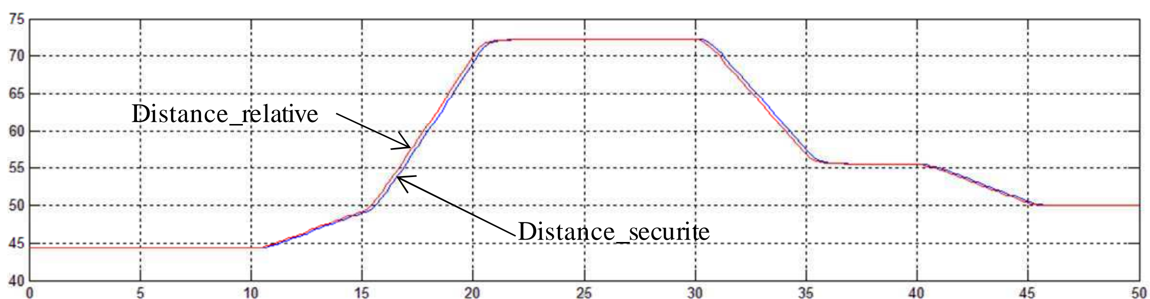

Figure 47. Distance de sécurité et distance relative

[4] Dans « Matlab Command Window », nous créons un vecteur de temps de durée 50 secondes avec un pas de 0.1 seconde *t=0:0.025:50;* puis nous demandons de tracer la vitesse du véhicule avant et celle du véhicule arrière ainsi que la distance relative et la distance de sécurité à l'aide des deux commandes suivantes: *plot(t(1:length(vva)), vva,'b', t(1:length(vvar)), vvar,'r');* et *plot(t(1:length(ds)), ds,'b', t(1:length(dr)), dr,'r');* qui nous permettent d'obtenir respectivement la relation entre les variations des vitesses des deux véhicules et la relation entre les variations de la distance de sécurité et de la distance relative entre les deux véhicules.

Un fichier trace est généré à la fin de la co-simulation, il permet d'observer et d'analyser plus profondément le comportement du système durant la simulation.

Les deux fichiers traces générés suite à la co-simulation «fichier_trace_ECU1» et «fichier_trace_ECU2», gardent la trace des instants qui marquent le début ou la fin de chaque variation de la variable « vitesse-véhicule_arrière » dans les deux calculateurs. Nous présentons ci-après, un extrait du fichier trace «fichier_trace_ECU1» :

```
$date
     Jul 26, 2010       16:51:01
$end

$version
              SystemC 2.2.0 --- Nov  9 2009 12:42:59
$end

$timescale
     1 ps
$end

$scope module SystemC $end
$var real    1  aaa  VVA       $end
$var real    1  aab  VVAR       $end
$var real    1  aac  DS       $end
$var real    1  aad  DR       $end
$var wire    8  aae  Data [7:0]  $end
$var wire    1  aaf  Clock       $end
$upscope $end
$enddefinitions  $end

$comment
All initial values are dumped below at time 0 sec = 0 timescale units.
$end

$dumpvars
r80 aaa
r80 aab
r44.44444444444444 aac
r44.44 aad
b1010000 aae
1aaf
$end

#12500
0aaf

#25000
1aaf

#37500
0aaf

#50000
1aaf

#62500
0aaf
```

Nous visualisons la trace de la co-simulation sous forme de chronogramme à l'aide des outils qui permettent la visualisation des signaux à partir d'un fichier avec l'extension .vcd. Ainsi, nous avons choisi l'outil « *vsim* » de « *Modelsim* » pour tracer les chronogrammes correspondants aux traces (Figure 48**).**

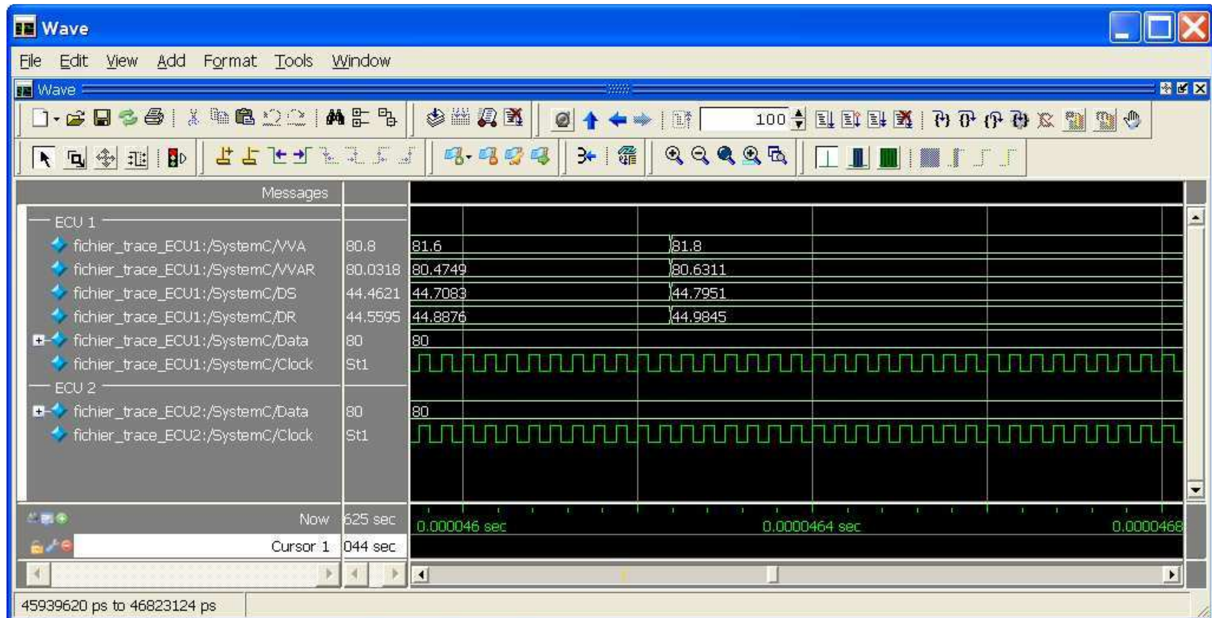

Figure 48. Chronogrammes des signaux des deux calculateurs

L'exécution de la méthode « *SDK ()*» est événementielle, elle est en effet liée aux signaux reçus du radar, les calculs ne se lancent pas cycliquement, ils se lancent après chaque variation de la valeur du signal reçu du radar (après chaque variation de la vitesse du véhicule avant : « *VVA* »).
Ensuite, suite aux variations de la valeur du signal « *VVA* » les variables « *VVAR* », « *DS* », « *DR* » et « *Data* » (égale à « *VVAR* », envoyée par le bus CAN) changent aussi de valeurs afin de réguler la puissance moteur du véhicule dans les meilleurs délais.

5. Résultat de l'analyse de la diagnosticabilité fonctionnelle-architecturale

a. Analyse de la disponibilité temporelle

<u>Vérification de la Propriété 5</u>: Disponibilité temporelle

Cette propriété prend la valeur « VRAI » lorsque le temps disponible est suffisant pour un diagnostiqueur (ou sous-fonction du diagnostiqueur dans le cas du diagnostic décentralisé).
Le diagnostiquer proposé pour la fonction SDK est en effet un diagnostiquer général pour tout le véhicule. L'objectif est de s'assurer que la fonction SDK restera toujours diagnosticable même lorsqu'elle est implantée sur le même calculateur que le diagnotiqueur du système.

Après une co-simulation du fonctionnement du système d'une minute, le module « *OBSAN* » de l'outil « *COSITA* », affiche une interface qui permet d'ouvrir le(s) fichier(s) trace de la co-simulation afin de vérifier la disponibilité temporelle pour un diagnostiqueur sur le calculateur « *ECU1*» et ceci en analysant les fichiers trace. Sachant que :

- la fréquence souhaitée du diagnostic est 1 fois tous les 4200ns = 4200/25= 168 cycles d'horloge,
- la durée du processus de diagnostic est estimée à 160 cycles = 160*25ns= 4000ns
- et le temps d'accès au calculateur pour lecture ou écriture est estimé à 10 ns.

Notre module OBSAN, permet d'ouvrir les fichiers traces, saisir les informations concernant le processus diagnostiqueur ainsi que la(les) fonction(s) implantée(s) sur le composant à analyser et effectuer une analyse de la disponibilité temporelle du composant par rapport aux paramètres saisis et souhaités, basée sur l'analyse des fichiers trace de la co-simulation.

Avec les paramètres que nous disposons, le module OBSAN déduit que le composant analysé (ECU1) n'est pas disponible pour implanter un processus de diagnostic avec les critères souhaités. Le résultat est négatif pour toute la durée de l'exécution, car la fonction analysée est périodique, le processus de diagnostic prévu est périodique également. Ainsi, si le résultat est négatif pour une période il l'est aussi pour toutes les autres pendant toute la simulation (Figure 49).

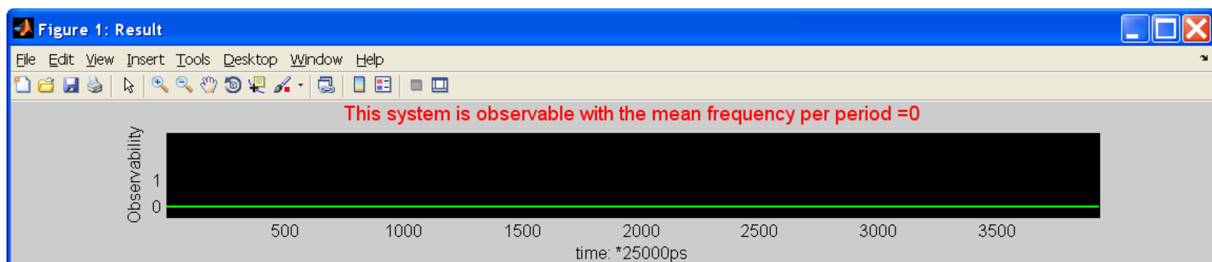

Figure 49. Résultat de l'analyse de la disponibilité temporelle

Ainsi, avec les caractéristiques proposées pour l'architecture, la propriété 5 n'est pas satisfaite.

b. Analyse de l'observabilité

<u>Vérification de la Propriété 6</u>: Observabilité

Pour satisfaire cette propriété, il faut :

- vérifier qu'il existe une disponibilité temporelle au niveau du système pour implémenter un diagnostiqueur,
- vérifier l'accessibilité de l'observateur aux variables portant des informations issues des capteurs par le diagnostiqueur, selon le modèle de diagnosticabilité fonctionnelle,
- et vérifier l'atteignabilité des calculateurs qui participent à l'événement en question.

Les deux derniers points de la propriété 6 sont satisfaits, alors que la disponibilité temporelle n'est pas satisfaite. Ainsi, la propriété 6 ne peut pas être satisfaite.
Comme les deux propriétés 5 et 6 ne sont pas satisfaites, la diagnosticabilité fonctionnelle-architecturale n'est pas satisfaite. Nous constatons que même si la fonction SDK est diagnosticable fonctionnellement, son architecture matérielle-logicielle proposée l'empêche d'être diagnostiquée.
Notre approche propose dans ce cas d'effectuer des modifications sur la description et le modèle de l'architecture matérielle-logicielle afin de la rendre diagnosticable de point de vue matériel-logiciel.

V. Conclusion

Dans ce chapitre, nous avons présenté une analyse de la diagnosticabilité fonctionnelle-architecturale de la fonction SDK. L'analyse a commencé par la construction du modèle du système global sous formes d'automates, ensuite par l'analyse de sa diagnosticabilité « fonctionnelle » dont le résultat est positif et nécessite une vérification au niveau fonctionnel-architectural. Pour ceci, nous avons établi une description de l'architecture matérielle en AADL ainsi qu'un co-modèle SystemC-Simulink de la fonction SDK et de son implantation sur les différents calculateurs. La description AADL a servi pour la première étape de la vérification des propriétés (à savoir la vérification de la description de l'architecture) et le co-modèle SystemC-Simulink a servi pour la deuxième étape de la vérification des propriétés (à savoir la vérification de la description de l'interaction architecturale-fonctionnelle, par l'analyse des traces de co-simulation).

Notre approche n'est pas limitée au niveau de grain que nous avons choisi pour la modélisation matérielle ou logicielle. En effet elle est adaptable au niveau du grain choisi pour la conception du système [79]. La méthode d'analyse des traces de co-simulation que nous avons définie dans notre approche est adaptable à des analyses plus poussées telles que les analyses statistiques ou probabilistes.

Enfin, comme notre méthode s'appuie sur l'itération de la conception et de l'analyse de la diagnosticabilité fonctionnelle-architecturale, nous proposerons dans le chapitre suivant un retour à l'étape de la conception dans un but de modification de la conception matérielle-logicielle du système.

De l'analyse de la diagnosticabilité vers la conception

I. Introduction

Dans ce chapitre, nous allons valider notre approche en deux étapes ; D'abord, par l'obtention du même résultat d'analyse de diagnosticabilité suite à l'émulation de la fonction SDK (avec la même configuration d'architecture proposée dans le chapitre « étude de cas »), sur une plateforme réelle. Ensuite, par le retour à la conception en reprenant aussi le même exemple que nous avons utilisé pour « l'étude de cas », nous montrerons l'intérêt d'un tel processus itératif de conception et de vérification de la diagnosticabilité dans la réalisation d'une architecture diagnosticable.

II. Validation par l'émulation

1. La plateforme DIAFORE

La plateforme DIAFORE est une plateforme physique composée de plusieurs calculateurs, de certains éléments de connexion et des logiciels de programmation. Elle a été délivrée par FAAR-Industry en mars 2008 et est actuellement dans les locaux de l'UTC. Cette plateforme constitue une partie d'une architecture électronique embarquée en automobile. Nous exploitons cette plateforme pour réaliser des tests qui nous rapprochent du fonctionnement concret et de la manipulation des architectures électroniques embarquées en automobile.

La plateforme DIAFORE est composée de trois calculateurs (ECUs) fabriqués par « Continental ». Deux calculateurs sont de type GCM-0563-048-0802 et le troisième est de type ECM-0555-080-0703 (Figure 50).

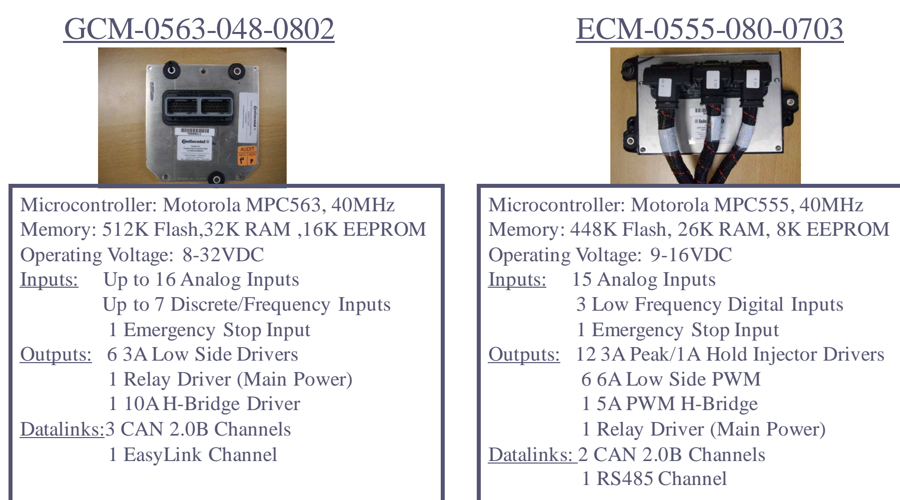

Figure 50. Les deux différents types de calculateurs utilisés pour la plateforme DIAFORE

La plateforme est constituée aussi d'un PC, une étoile CAN et une interconnexion CAN-USB de type Mototron [80]. Pour établir un bus CAN entre les calculateurs et le PC, il faut

connecter chacun d'eux à l'étoile CAN. Le PC ne peut pas être connecté directement pour cela nous disposons de l'interconnexion CAN-USB (Figure 51).

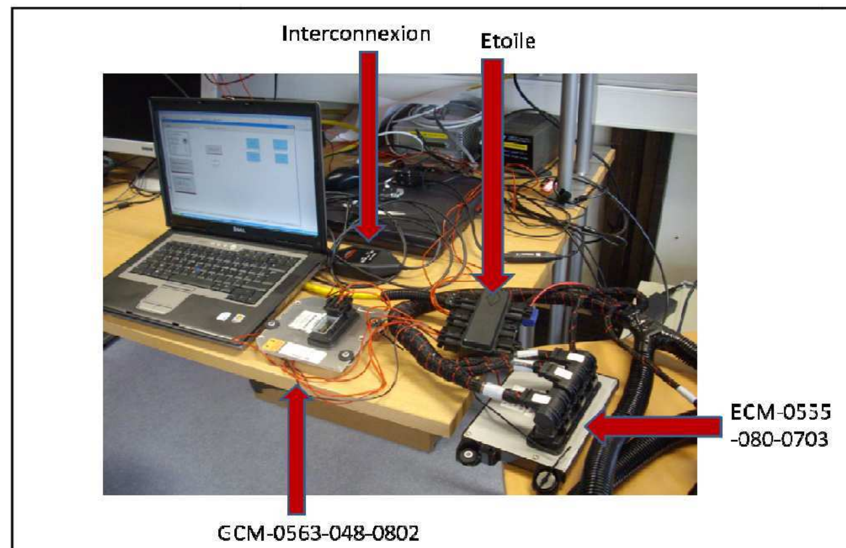

Figure 51. La plateforme DIAFORE

La plateforme dispose d'un ensemble de logiciels qui permettent sa manipulation et son contrôle, à savoir :

- Matlab / Simulink 7.0 : Ce logiciel de programmation permet de modéliser et tester les fonctions embarquées. Il permet aussi, grâce à la librairie Motohawk (Annexe E) [81] qui lui est ajoutée, de configurer et programmer la plateforme DIAFORE. De plus il contient le RTW (*Real TimeWorkshop*) nécessaire pour la génération le code C équivalent aux blocs de Simulink.
- GREENHILLS-MULT I: Ce logiciel permet de générer du code C équivalent aux blocs de Motohawk [82].
- Mototune : Ce logiciel permet d'effectuer, en temps réel, des mesures, des calibrations, des enregistrements et des opérations de gestion sur les calculateurs (Annexe F) [83].

2. Programmation de la plateforme

Pour programmer les ECUs de la plateforme, il faut y implanter du code C. Mais dans l'industrie automobile moderne, le développement en C est assez lourd et la plupart des concepteurs adoptent Simulink/Matlab pour la conception et la modélisation des fonctions embarquées. Il permet une modélisation simple, rapide et sûre d'une fonction dynamique grâce à ses librairies standards sous formes de blocs. De plus, grâce à des nouvelles librairies ajoutées telle que la librairie Motohawk de Mototune, Simulink a plus la capacité de s'adresser aux ECUs et de gérer le bus CAN et ses trames, Ensuite, les blocs sont transformés en code C. Mototune permet la programmation de chaque ECU. Une fois tous les programmes sont bien installés sur les ECUs, ils commencent à s'exécuter continuellement suivant leur code. Mototune permet de commander et d'observer les différentes valeurs d'entrées et de sorties configurées par le modèle Simulink.

3. Emulation de SDK sur la plateforme Diafore

L'exemple de la fonction SDK a été testé sur cette plateforme en considérant les mêmes paramètres d'entrée utilisés pour la co-simulation [84]. Au début de la co-simulation le véhicule équipé par SDK et le véhicule en avant roulent à une vitesse égale à 80 Km/h avec une distance de sécurité de 45 m. Les variations des résultats s'affichent sur un graphe distinguant les variations de la distance entre véhicules, la puissance du moteur et la vitesse du véhicule SDK (Figure 52).

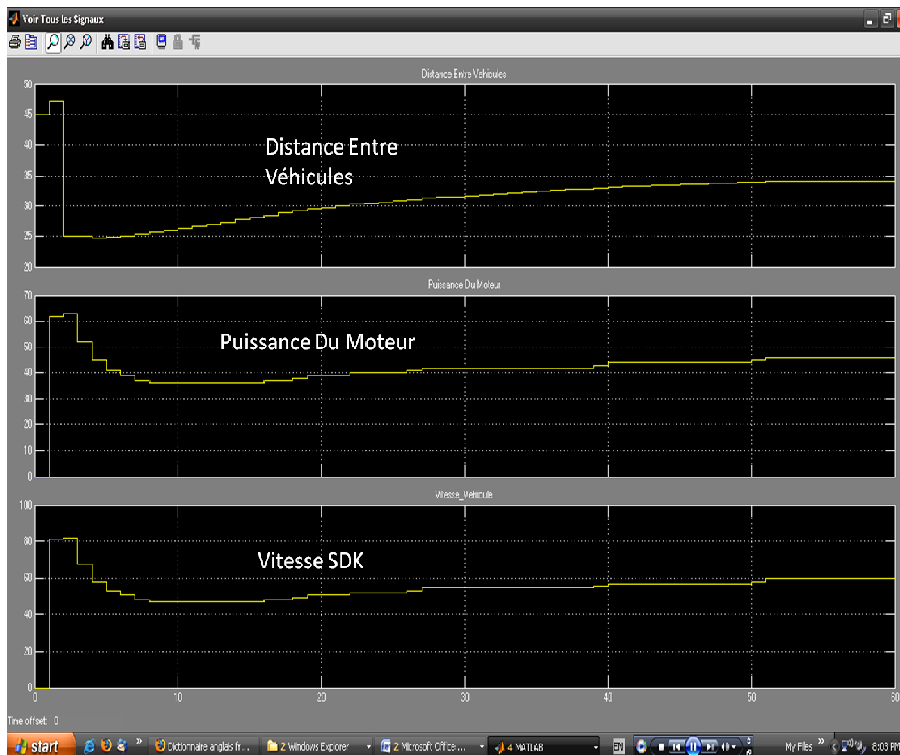

Figure 52. Variation des signaux du SDK

Mototune génère un fichier trace d'émulation, avec un pas d'acquisition des données limité à 63ms, Pour l'exemple de SDK nous ne pouvons pas obtenir un fichier trace précis, car les variations des signaux qui existent dans une période de temps plus petite que 63ms ne sont pas détectables. Afin d'avoir plus de précision dans le fichier trace, nous avons utilisé le logiciel CANalyzer [85], installé sur un PC relié grâce au bus CAN à l'architecture. Il permet de générer des fichiers de traces reproduisant le trafic sur le bus CAN. Pour ceci, nous avons développé un programme interne à CANalyzer, en utilisant son propre langage de programmation CAPL (*Communication Access Programming Language*) (Annexe G). Ce programme, détecte les moments d'activation et les moments de repos de la fonction SDK. Dans les moments où la fonction SDK fonctionne, un signal oscillant entre 0 et 1 s'enregistre dans un fichier au format .log (Figure 53). Dans le programme écrit en CAPL, nous avons choisi une fréquence d'échantillonnage égale à 25ns afin qu'elle soit la même avec celle utilisée par SystemC pour la trace de la co-simulation :

settimer(timer1,0.025); //fréquence d'échantillonnage =0.025ms =25ns.

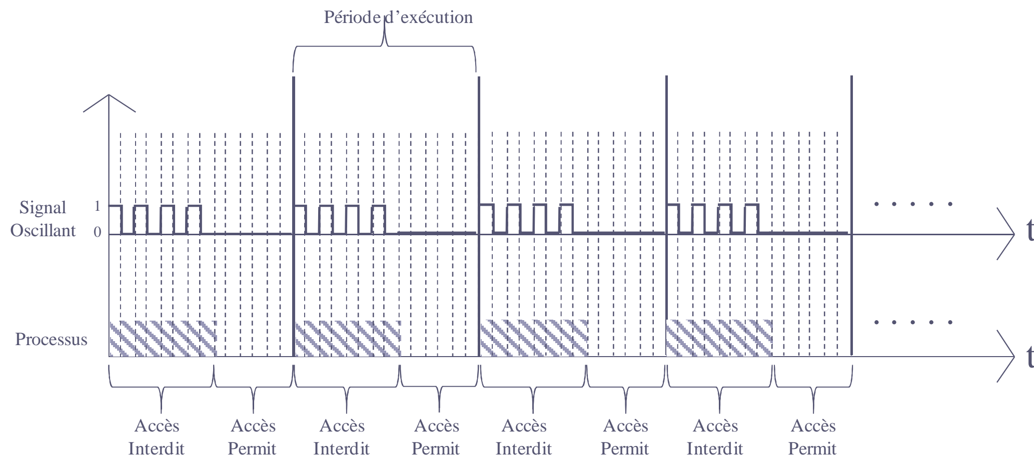

Figure 53. Signal Oscillant

4. Analyse de la diagnosticabilité

a. Vérification des propriétés de l'architecture

La plateforme Diafore possède la même description d'architecture que nous avons utilisée dans le chapitre « étude de cas ». Ainsi le résultat de la vérification des propriétés de l'architecture de la diagnosticabilité reste le même. Par la suite, la vérification de la diagnosticabilité de ce système s'arrête au résultat de la vérification des propriétés de l'interaction fonctions-architecture et cette fois-ci à partir des traces de l'émulation.

b. Vérification des propriétés de l'interaction fonctions-architecture

Suite à l'émulation, un fichier .log est généré par CANalyzer. Ensuite, nous utilisons un autre module de « COSITA », que nous avons développé en C++ et appelé « log2vcd » (Annexe H), qui convertit les fichiers du format .log au format .vcd, afin de créer un fichier interprétable par le module « OBSAN » (Figure 54).

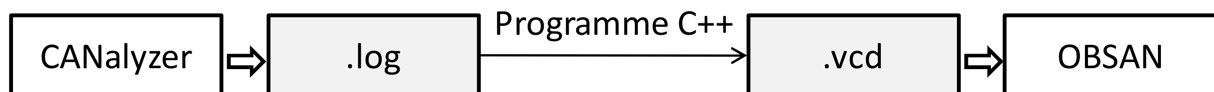

Figure 54. La conversion du .log au .vcd

Les paramètres saisis au niveau de l'interface de « OBSAN » sont les mêmes utilisés précédemment dans le chapitre « étude de cas » :

- la fréquence souhaitée du diagnostic est 1 fois tous les 4200ns = $\frac{4200}{25}$ = 168 cycles d'horloge (durée du cycle 25ns),
- la durée du processus de diagnostic est estimée à 160 cycles = 160*25ns= 4000ns
- et le temps d'accès au calculateur pour lecture ou écriture est estimé à 10 ns.

Le module OBSAN déduit que le composant analysé (ECU1) n'est pas disponible pour implanter un processus de diagnostic avec les critères souhaités ; Il s'agit du même résultat obtenu suite à la co-simulation.

c. Comparaison des résultats de l'émulation avec les résultats de la co-simulation (module SIMECO)

A priori, il s'agit du même résultat car il s'agit du même fichier trace généré. Afin d'être sûrs que les fichiers trace de co-simulation et d'émulation sont similaires, nous avons développé un autre module de « COSITA » que nous avons appelé « SIMECO » pour (*SIMulation Emulation COmparison*). SIMECO pointe toutes les divergences et laisse à l'utilisateur de rajouter des filtres sur les divergences qu'ils considèrent comme critiques :

- Définir une variable et la tolérance dans sa valeur,
- Définir un instant ou une période pendant laquelle l'écart ne doit pas dépasser un certain seuil, etc.

Pour l'exemple de SDK la comparaison a conduit à des divergences légères dues la plupart du temps à des écarts de fraction des valeurs de certaines variables, provenant probablement des différences de précision des calculs mathématiques (Annexe C).
Nous avons effectué plusieurs co-simulations et émulations de scénarios différents de SDK afin de s'assurer que l'analyse de la diagnosticabilité fonctionnelle-architecturale est toujours la même dans les deux cas. Ainsi, grâce à cette comparaison, nous avons prouvé que notre approche de vérification de la diagnosticabilité fonctionnelle-architecturale est réalisée au niveau de la conception d'une architecture mais reproduit des résultats proches (ou les mêmes) de ceux qu'on peut obtenir à la phase de l'implémentation.

III. Validation par retour à la conception

Dans le chapitre précédent nous avons constaté que même si la fonction SDK est diagnosticable fonctionnellement, le choix de l'architecture matérielle-logicielle de son implémentation l'empêche d'être diagnostiquée. Notre approche propose dans ce cas, de réaliser des modifications sur la description et sur le modèle de conception de l'architecture matérielle-logicielle du système afin de la rendre diagnosticable de point de vue matériel-logiciel. Ces modifications sont suggérées au concepteur qui les intégrera selon son expertise et savoir faire. Ainsi, nous proposerons dans ce chapitre des modifications à l'architecture matérielle-logicielle de SDK afin qu'elle devienne diagnosticable.

A l'étape de la vérification des propriétés, toutes les propriétés sont satisfaites à part la propriété 5 « la disponibilité temporelle », ce qui a engendré la non satisfaction de la propriété 6 « l'observabilité ». Ainsi, il faudrait trouver une solution au niveau de la conception de l'architecture matérielle-logicielle pour que la « disponibilité temporelle » soit satisfaite. Nous proposons dans ce cas, deux solutions différentes :

- La première consiste à utiliser un calculateur plus puissant au lieu du calculateur ECU1 (Calculateur_radar) de fréquence 40 MHz prévu pour l'implantation de la fonction SDK et du diagnostiqueur.
- La deuxième solution consiste à ajouter à l'architecture un calculateur de la même fréquence que l'ECU1 (Calculateur_radar) et ECU2 (Calculateur_moteur) afin d'y implanter le diagnostiqueur.

Ci-après, nous testerons les deux solutions proposées en estimant à chaque cas leur coût.

1. Première proposition de modification de l'architecture

En prenant un calculateur plus puissant à 80MHz par exemple (disponible sur étagère), la durée du cycle d'horloge est de $\frac{1}{80*10^6}$s = 12.5.10^{-9} s = 12.5 ns.
Comme la méthode « SDK () » est exécutable en 16 cycles d'horloge, elle est alors exécutable sur ce calculateur en 16*12.5= 200ns. De même, le diagnostiqueur est exécutable sur 160 cycles d'horloge, il est alors exécutable sur ce nouveau calculateur en 160*12.5= 2000ns.
Nous présumons que la périodicité souhaitée du diagnostic est la même que dans la configuration précédente de l'architecture, une fois toutes les 4200ns.
Ainsi, pour la première solution à tester la description ainsi que le modèle SystemC de l'architecture reste le même à part l'horloge du calculateur ECU1 qui doit passer à 80MHz :

```
sc_clock clock("clock",12.5,SC_NS);
```

Au niveau de la vérification des propriétés de l'architecture, les résultats restent les mêmes au niveau de la propriété 1.1, la propriété 3 et la propriété 4, car la description de l'architecture n'a changé qu'au niveau de l'attribut « microprocesseur » du Calculateur_radar :

```
system implementation Calculateur_radar.imp
  subcomponents
    ……;
    Proc : processor MPC563.Motorola {Clock_Period => 12.5 ns ;};
    …..;
end Calculateur_radar.imp;
```

Ainsi la propriété 2 doit être re-vérifiée :

<u>Vérification de la Propriété 2</u> : Exécutabilité diagnostiqueur

Nous avons :

F_Diag= p_diag,
Size_Mem (D_principal) = Size_Mem (ECU1) =10Mo,
Size_F (p_diag) =600Ko,
ΔT (p_Diag) = 4200ns
WCET (p_Diag, ECU1) = 2000ns

Ainsi :

$\forall d \in D, \forall f \in F_Diag,$

((Implemented (f,d)==1) ^ (Size_Mem (d)$\geq$ Size_F (f)) ^ (ΔT (f) >= WCET (f,d))) $\rightarrow$ Exécutabilité (f,d)=1

Au niveau de la vérification de l'interaction focntions-architecture, nous co-simulons cette configuration d'architecture matérielle-logicielle avec le même modèle Simulink de la fonction SDK et les mêmes valeurs d'entrée que la configuration précédente, ce qui reproduit le même résultat au niveau de l'exécution de la fonction SDK (Figure 55).

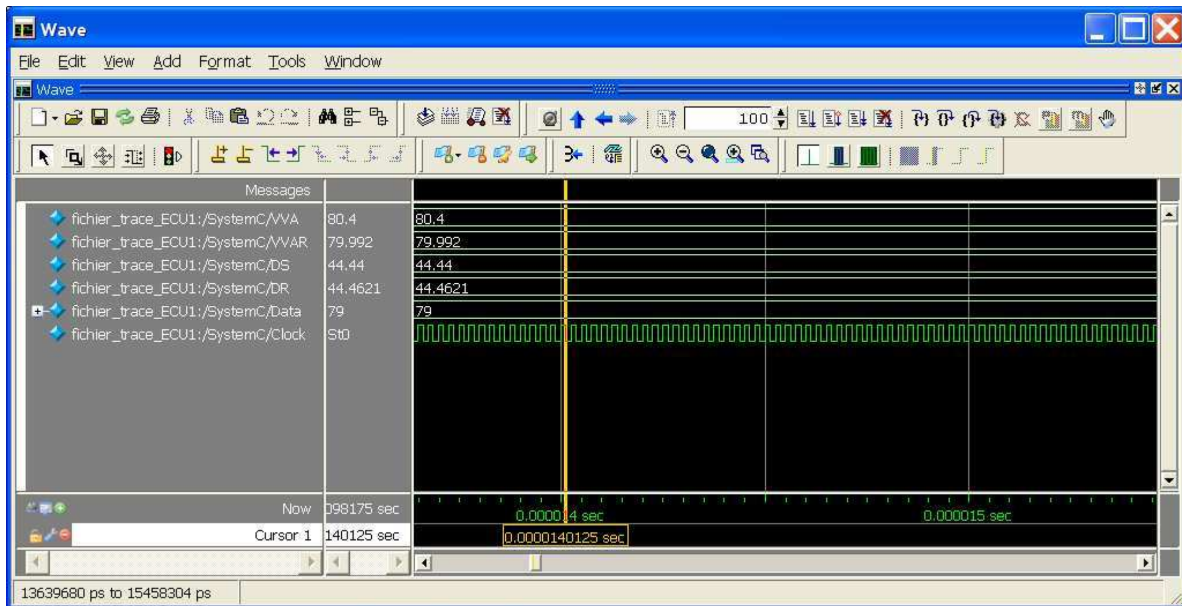

Figure 55. Co-simulation avec ECU1 à une fréquence de 80MHz

Vérification de la Propriété 5: Disponibilité temporelle

Le temps d'accès à ce calculateur pour une lecture/écriture est estimé à 10 ns. La périodicité souhaitée du diagnostic est la même que dans la configuration précédente de l'architecture, une fois toutes les 4200 ns, équivalente ainsi à $\frac{4200}{12.5}$ = 336 cycles d'horloge et la durée du processus diagnostiqueur est la même 160 cycles égale à 12.5*160= 2000ns.

Avec les nouveaux paramètres ainsi saisis, l'analyse de la disponibilité temporelle du calculateur à 80MHz devient favorable.

Vérification de la Propriété 6: Observabilité

La propriété « disponibilité temporelle » est satisfaite, la propriété « observabilité » est par la suite satisfaite également, car ses deux autres points « l'atteignabilité » et « l'accessibilité » sont déjà satisfaites.

Avec les modifications proposées, l'architecture devient diagnosticable de point de vue fonctionnel-architectural.

2. Deuxième proposition de modification de l'architecture

La deuxième solution proposée est l'ajout d'un calculateur à l'architecture de la même puissance que les calculateurs « Calculateur_radar » et « Calculateur_moteur » (puissance égale à 40MHz) afin d'y implanter seul le diagnostiqueur (Figure 56).

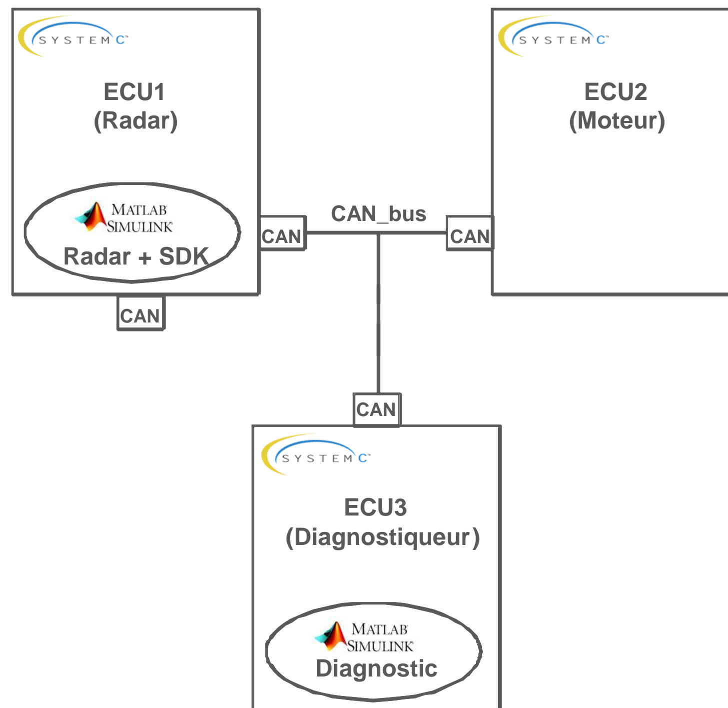

Figure 56. Configuration de l'architecture matérielle-logicielle avec un calculateur supplémentaire

Le calculateur à 40MHz a une durée de cycle d'horloge de $\frac{1}{40*10^6}$ s = 25.10^{-9} s = 25 ns.
La description de ce calculateur en AADL est comme suit :

```
processor implementation MPC563.Motorola3
properties
Clock_Period => 25 ns;
end MPC563.Motorola3;

memory implementation RAM1.Motorola_RAM3
end RAM1.Motorola_RAM3;


bus implementation membus.impl3
end membus.impl3;


-------Le diagnostiqueur à 40 MHz------

system Calculateur_diagnostiqueur
  features
  interface : requires bus access CAN.impl;
end Calculateur_diagnostiqueur;

system implementation Calculateur_diagnostiqueur.imp
  subcomponents
    HSRAM3: memory RAM1.Motorola_RAM3;
    Proc : processor MPC563.Motorola3 {Clock_Period => 25 ns;};
      high_speed_bus : bus membus.impl3;
  connections
  bus access high_speed_bus -> proc.bus01;
  bus access interface -> proc.bus_can;
  bus access interface -> HSRAM3.bus_can;
end Calculateur_diagnostiqueur.imp;
```

Les connexions de type « Data » changent au niveau du système complet :

```
system implementation systeme_complet.impl
  subcomponents
    ….
    ECU3: system Calculateur_diagnostiqueur.imp;
     ….

      connections
      ….
      -- vers le processus diagnostiqueur
      cnx5: data port p_radar.output_RV_diag -> p_diag.input_RV_diag
      {Actual_Connection_Binding => reference reseau_CAN;};
      cnx6: data port p_sdk.output_VV_diag -> p_diag.input_VV_diag
      {Actual_Connection_Binding => reference reseau_CAN;};
      cnx7: data port p_radar.output_detection_diag ->
      p_diag.input_detection_diag
      {Actual_Connection_Binding => reference reseau_CAN;};
      cnx8: data port p_sdk.output_vva_and_vvar_diag ->
      p_diag.input_vva_and_vvar_diag
      {Actual_Connection_Binding => reference reseau_CAN;};


      -- Connexions au bus CAN
      bus access reseau_CAN -> ECU3.interface;

      -- Liaisons
      properties
      …….
      Actual_processor_Binding => reference ECU3.proc applies to p_diag; --
      mapping du processus p_diag sur le processeur
      Actual_memory_Binding => reference ECU3.HSRAM3 applies to p_diag; --
      mapping du processus p_diag sur la mémoire

end systeme_complet.impl;
```

La destination des messages change suivant la nouvelle description de l'architecture :

- Les deux messages contenant les valeurs des variables d'état,

Message_1 :
Source : ECU1
Destination : ECU3
Contenu : Valeur(RV)

Message_2 :
Source : ECU1
Destination : ECU3
Contenu : Valeur (VV)

- Ainsi que quatre messages pour notifier les événements « détection », « VVa^VVar », « Décision_de_régulation » et « contrôle_Moteur »:

Message_3 :
Source : ECU1
Destination : ECU3

```
Contenu : Valeur (Détection)

Message_4 :
Source : ECU1
Destination : ECU3
Contenu : Valeur (VVa ^ VVar)

Message_5 :
Source : ECU2
Destination : ECU3
Contenu : Valeur (Décision de régulation)

Message_6 :
Source : ECU2
Destination : ECU3
Contenu : Valeur(Contrôle_Moteur)
```

Au niveau de la vérification des propriétés de l'architecture, de la diagnosticabilité fonctionnelle-architecturale, nous vérifions ci-après que la description du système répond aux propriétés de l'architecture exigées.

Soient les variables :
A : L'ensemble des calculateurs (ou composants) contenants une ou plusieurs fonctions à analyser -> A= {ECU1, ECU2}
B : L'ensemble des composants sources de variables d'état -> B = {ECU1}
C : L'ensemble de tous les composants calculateurs **->** C= {ECU1, ECU2}
D_principal : Le calculateur (ou composant) diagnostiqueur -> D_principal= ECU3
E : L'ensemble des évènements à diagnostiquer -> D = {Détection, (Vva^Vvar), Décision_Régulation, Contrôle_Moteur}
F : L'ensemble de toutes les fonctions
F_Diag : L'ensemble des fonctions de diagnostic -> F_Diag= {p_Diag}
M : L'ensemble des messages envoyés contenant la valeur des variables d'état ou la notification des événements de l'automate représentant le système-> M= {Message_1, Message_2, Message_3, Message_4, Message_5, Message_6}
O : Le calculateur contenant le processus observateur
V : L'ensemble des variables d'états issues de capteurs ou autres composants du système ->V= {Rv, Vv}

Soit le type :
void : Type de retour non défini différent de null

Soient les fonctions :
F_Data : $(x,y)\rightarrow\{0,1\} \backslash (x,y) \in C^2$ retourne 1 si x et y sont connectés par une connexion de type données, retourne 0 sinon
Size_F : $x\rightarrow y \backslash x \in F_Diag, y \in \mathbb{R}$ retourne la taille de x
Size_Mem : $x\rightarrow y \backslash x \in C, y \in \mathbb{R}$ retourne la taille mémoire de x
ΔT : $x\rightarrow y \backslash x \in F_Diag, y \in \mathbb{R}$ Fonction qui retourne la périodicité souhaitée pour l'exécution de la fonction x
WCET : $(x,y)\rightarrow z \backslash x \in F_Diag, y \in C, z \in \mathbb{R}$ retourne le temps maximal d'exécution que x peut prendre sur la plateforme y, correspond au WCET (W*orst-Case Execution Time*) de x sur y

Conn_I/O : (x,y)→z\ x∈ C, y ∈ B, z ∈{0,1} retourne 1 si x et y sont connectés
Implemented : (x,y)→z\x∈ F, y ∈ C, z ∈ {0,1}retourne 1 si x est implémentée sur y
Source : x→y\ x∈ M, y∈ C retourne le nom (ou adresse) du calculateur émetteur du message x.
Destination : x→y\ x∈ M, y∈ C retourne le nom (ou adresse) du calculateur récepteur du message x.
Valeur : x→y\ x∈ M, y∈ void retourne la valeur du message M.

Vérification de la Règle1 **:**

Comme la structure de diagnostic employée est centralisée, il faut vérifier les propriétés 1.1, 2, 3, 4, 5, 6 et 7

Vérification de la Propriété 1.1 : Connectivité-Diagnostiqueur

Nous avons :
ECU1 ∈ A, F_Data(ECU1, D_principal)=1

Ainsi :
Connectivité_Diagnostiqueur (ECU1)=1

-------------------***************-------------------*****************--------------------
Nous avons :

ECU2∈ A, F_Data(ECU2, D_principal)=1,

Ainsi :
Connectivité_Diagnostiqueur (ECU2)=1

-------------------***************-------------------*****************--------------------

Donc :
∀a ∈ A, Connectivité_Diagnostiqueur (a)=1

Vérification de la Propriété 2 : Exécutabilité diagnostiqueur

Nous avons :

F_Diag= p_diag,
Size_Mem (D_principal) = Size_Mem (ECU3) =10Mo,
Size_F (p_diag) =600Ko,
ΔT (p_Diag) = 4200ns
WCET (p_Diag, ECU3) = 4000ns

Ainsi :

$\forall d \in D, \forall f \in F_Diag,$

((Implemented (f,d)==1) ^ (Size_Mem (d)≥ Size_F (f)) ^ (ΔT (f) >= WCET (f, d))) → Exécutabilité (f,d)=1

Vérification de la Propriété 3: Atteignabilité

Nous avons :
RV ∈ V,
Origine (RV)=ECU1 ∈ B,
Message_1 ∈ M,
Source(Message_1)=ECU1,
Destination(Message_1)=ECU3,
Contenu (Message_1)=Valeur(RV),
O= ECU3,

Alors :

(Conn_I/O (Origine(RV), O) ∨ F_Data (Origine(RV), O)) ^
Source(Message_1)=Origine(RV) ^ Destination(Message_1)=O ^
Contenu(Message_1) =Valeur(RV)

Ainsi :

Atteignabilité (RV)=1

-------------------********--------------*******--------------******----------------*********---

Nous avons :
VV ∈ V,
Origine (VV)=ECU1 ∈ B,
Message_2 ∈ M,
Source(Message_2)=ECU1,
Destination(Message_2)=ECU3,
Contenu (Message_2)=Valeur(VV),
O= ECU3,

Alors :
(Conn_I/O (Origine(VV), O) ∨ F_Data (Origine(VV), O)) ^
Source(Message_2)=Origine(VV) ^ Destination(Message_2)=O ^ Contenu (Message_2) =Valeur(VV)

Ainsi :

Atteignabilité (VV)=1

<u>Vérification de la Propriété 4</u> : Accessibilité

Nous avons :

detection ∈ E,
Origine (detection)=ECU1 ∈ B,
Message_3 ∈ M,
Source(Message_3)=ECU1,
Destination(Message_3)=ECU3,
Contenu (Message_3)=Valeur (detection),
O= ECU3,

Alors :

(Conn_I/O (Origine (detection), O) ∨ F_Data (Origine (detection), O)) ^
Source(Message_3)=Origine (detection) ^ Destination(Message_3)=O ^
Contenu(Message_3)=Valeur (detection)

Ainsi :
Accessibilité (detection)=1

------------------********--------------*******--------------******----------------*********---

Nous avons :

(VVa ^ VVar) ∈ E,
Origine ((VVa ^ VVar))=ECU1 ∈ B,
Message_4 ∈ M,
Source(Message_4)=ECU1,
Destination(Message_4)=ECU3,
Contenu (Message_4)=Valeur ((VVa ^ VVar)),
O= ECU3,

Alors :

(Conn_I/O (Origine ((VVa ^ VVar)), O) ∨ F_Data (Origine ((VVa ^ VVar)), O)) ^
Source(Message_4)=Origine ((VVa ^ VVar)) ^ Destination(Message_4)=O ^
Contenu(Message_4)=Valeur ((VVa ^ VVar))

Ainsi :
Accessibilité ((VVa ^ VVar))=1

------------------********--------------*******--------------******----------------*********---
Nous avons :

Décision_de_régulation ∈ E,
Origine (Décision_de_régulation)=ECU1 ∈ B,
Message_5 ∈ M,
Source(Message_5)=ECU1,
Destination(Message_5)=ECU3,

Contenu (Message_5)=Valeur(Décision_de_régulation),
O= ECU3,

Alors :

(Conn_I/O (Origine(Décision_de_régulation), O) ∨
F_Data(Origine(Décision_de_régulation),O)) ^
Source(Message_5)=Origine(Décision_de_régulation) ^ Destination(Message_5)=O ^
Contenu(Message_5)=Valeur(Décision_de_régulation)

Ainsi :
Accessibilité (Décision_de_régulation)=1

------------------********-------------*******-------------******----------------*********---

Nous avons :

Contrôle_Moteur ∈ E,
Origine (Contrôle_Moteur)=ECU1 ∈ B,
Message_6 ∈ M,
Source(Message_6)=ECU1,
Destination(Message_6)=ECU3,
Contenu (Message_6)=Valeur(Contrôle_Moteur),
O= ECU3,

Alors :

(Conn_I/O (Origine(Contrôle_Moteur), O) ∨ F_Data(Origine(Contrôle_Moteur),O)) ^
Source(Message_6)=Origine(Contrôle_Moteur) ^ Destination(Message_6)=O ^
Contenu(Message_6)=Valeur(Contrôle_Moteur)

Ainsi :
Accessibilité (Contrôle_Moteur)=1

Ensuite, pour la vérification des propriétés 5 et 6, nous avons d'abord intégré dans le modèle SystemC de l'architecture le modèle du nouveau calculateur (et sa liaison à l'aide du bus CAN à l'ECU1 et l'ECU2).

```
SC_MODULE(ECU3)
{
	sc_inout<sc_uint<40>> CAN2;
	sc_inout<sc_uint<40>> CAN3;
	sc_in_clk CLK;
	sc_uint<5> id;
	sc_uint<3> length;
	sc_uint<8> data[2];
	Engine *ep;
	double vvar, memory_vvar;
```

```
	void ReadCAN(sc_uint<5> *id, sc_uint<3> *length, sc_uint<8> *data)
	{
		sc_uint<40> trame;
		sc_uint<32> data_block;
		trame = CAN2.read();
		if (*id == trame.range(39,35))
		{
			*length = trame.range(34,32);
			data_block = trame.range(31,0);
			for(int I = 0; i<*length; i++)
			{
				data[i] = data_block.range( 31 - 8*I, 31 - (8*I +
7));
			}
		}
	}



	SC_CTOR(ECU3)
	{
		SC_METHOD(diagnostic);
		dont_initialize() ;
		sensitive<<CLK.pos() ;
		sc_trace_file *tf=
sc_create_vcd_trace_file("fichier_trace_ECU3");
		sc_trace(tf,CLK,"Clock");
		sc_trace(tf,data[0],"Data");

	}
```

Comme notre objectif dans cette thèse ne s'intéresse pas à la réalisation d'un diagnostiqueur, nous avons intégré un processus de diagnostic simplifié dans le modèle, qui lit toutes les valeurs envoyées par le Calculateur_radar au Calculateur_moteur.

```
	Void diagnostic()
	{
	id = 17;
	ReadCAN(&id,&length,data);
	vvar= double (data[0]);

	if (vvar != memory_vvar)
		{

			next_trigger(40,SC_US) ;// delai d'acheminement de la
trame
	}

		memory_vvar=vvar;



	}
};
```

Nous co-simulons cette configuration d'architecture matérielle-logicielle avec le même modèle Simulink de la fonction SDK et les mêmes valeurs d'entrée que la configuration

précédente. Nous obtenons le même résultat au niveau de l'éxécution de la fonction SDK sur ECU1 et ECU2 et donne une idée sur le fonctionnement du calculateur_diagnostiqueur ECU3 quand il recoit les signaux de ECU1.

<u>Vérification de la Propriété 5</u>: Disponibilité temporelle

Le temps d'accès à ce calculateur pour une lecture/écriture est estimé à 10ns. La périodicité souhaitée du diagnostic est la même que dans la configuration précédente de l'architecture, une fois toutes les 4200ns équivalente ainsi à $\frac{4200}{25}$ =168 cycles d'horloge et la durée du processus diagnostiqueur est la même 160 cycles égale à 25*160= 4000 ns. Avec les nouveaux paramètres ainsi saisi, l'analyse de la disponibilité temporelle du calculateur supplémentaire à 40MHz devient favorable.

<u>Vérification de la Propriété 6</u>: Observabilité

La propriété « disponibilité temporelle » est satisfaite, la propriété « observabilité » est par la suite satisfaite aussi, car ses deux autres points nécessaires « l'atteignabilité » et « l'accessibilité » sont déjà satisfaites.

Avec les modifications proposées, l'architecture devient diagnosticable de point de vue fonctionnel-architectural.

3. Comparaison des deux solutions

Les deux solutions que nous avons proposées, mènent à deux configurations différentes d'architecture diagnosticable. La différence majeure entre les deux solutions est le coût, un écart estimé approximativement à 1800 euros (car il y a des coûts cachés de l'ingénierie et de la validation technique) qui offre plus de rapidité d'exécution (Tableau 4). Dans le cas du système que nous avons analysé, nous n'avons pas besoin d'une exécution plus rapide de la fonction SDK ou de la fonction de diagnostic, toute notre attente se focalise sur une architecture matérielle-logicielle diagnosticable. Ainsi, la première solution est, selon la comparaison que nous avons faite, plus raisonnable.

Tableau 4. Récapitulation des avantages et inconvénients des deux solutions proposées

Solution	Calculateurs	Caractéristiques	Diagnosticabilité	Coût
1	ECU1 : 80MHz	Exécution SDK : 200ns Exécution diagnostic : 2000ns	oui	500 euros supplémentaire par rapport à un calculateur à 40MHz
	ECU2 : 40MHz			
2	ECU1 : 40MHz	Exécution SDK : 400ns	oui	2000 euros (prix d'un calculateur) + 300 euros (prix des câbles de connexion)
	ECU2 : 40MHz			
	ECU3 : 40MHz	Exécution diagnostic : 4000ns		

IV. Conclusion

Dans ce chapitre, nous avons validé notre approche d'abord par l'obtention du même résultat d'analyse de diagnosticabilité suite à la co-simulation et à l'émulation. Grâce à cette étape, nous avons prouvé que notre approche de vérification de la diagnosticabilité fonctionnelle-architecturale est réalisée au niveau de la conception d'une architecture mais obtient des résultats proches de ceux qu'on peut obtenir à la phase de l'implémentation. En seconde étape, nous avons validé notre approche par le retour à la conception, en montrant l'intérêt d'un tel processus itératif de conception et de vérification de la diagnosticabilité dans la réalisation d'une architecture diagnosticable.

Conclusion et perspectives

Dans cette thèse, une méthode d'analyse de diagnosticabilité fonctionnelle-architecturale basée sur la co-modélisation logicielle-matérielle a été réalisée. D'après la littérature, la prise en compte des caractéristiques de l'architecture matérielle - considérée souvent comme des « contraintes » d'implémentation – pendant l'analyse de la diagnosticabilité est un point très original. La méthode que nous avons développée pour confronter le modèle de l'architecture (exprimé par exemple en AADL) avec le modèle de diagnosticabilité classique (portant sur la fonction) est particulièrement inédite.

Des propriétés de l'architecture et de l'interaction fonctions-architecture ont été définies pour analyser la diagnosticabilité fonctionnelle-architecturale. Durant la construction de notre approche, celles-ci ont été considérées comme indicateurs pour la vérification de la diagnosticabilité d'un système embarqué.

Néanmoins, ces propriétés sont progressivement devenues des exigences pour le processus itératif de conception conjointe de l'architecture matérielle-logicielle, y inscrivant ainsi la vérification de la diagnosticabilité comme étape essentielle.

Un prototype d'outil (COSITA) a été développé afin d'analyser les propriétés de l'interaction fonctions-architecture à partir des traces de co-simulation, bénéficiant ainsi de la puissance des moteurs de simulation des outils reconnus sur le marché (Matlab/Simulink, Modelsim, SystemC, etc.). Pour la validation de l'approche, nous avons effectué des tests sur une architecture réelle ayant les mêmes caractéristiques que le modèle de l'architecture réalisé en SystemC et nous avons obtenus les mêmes conclusions de l'analyse de la diagnosticabilité fonctionnelle-architecturale. Bien entendu, les prototypes d'outils développés sont encore embryonnaires.

Malgré les avantages que présente notre approche, elle souffre de plusieurs faiblesses. En effet, le module OBSAN de COSITA développé pour analyser la disponibilité temporelle, considère les exécutions du diagnostiqueur comme périodiques avec un temps d'exécution fixe. Nous nous sommes limités au seul cas de la périodicité de l'exécution de la fonction embarquée du diagnostic, car dans les architectures embarquées en automobile, la plupart des fonctions implantées sur les calculateurs sont exécutées périodiquement.

Comme perspectives, nous pensons d'abord à rendre le processus de vérification de la disponibilité temporelle « presqu'instantané » pendant la co-simulation ou de l'émulation du système. Ceci permettrait d'avoir les mêmes résultats d'analyse plus rapidement, sans avoir recours à analyser des fichiers de traces.

Nous devrons tenir compte également de la possibilité d'avoir des temps d'exécution non périodiques pour le diagnostiqueur. Peut-être faudrait-il penser à des plannings prévisionnels de l'exécution du diagnostiqueur adaptés au comportement dynamique du système.

Nous comptons également reprendre le développement de nos prototypes d'outils (COSITA) afin d'affiner et de finaliser la génération automatique de scénarios de simulation. Nous devons également développer plus de passerelles vers les outils existants, à la fois ceux

utilisés pour la capture d’exigences et pour la vérification de contraintes temporelles. Nous suivrons ainsi les recommandations d’un partenaire industriel.

En échangeant avec d’autres équipes de recherches, il nous a été recommandé d’explorer l’exploitation « statistique » ou « probabiliste » des traces de simulation. Toutefois, cette orientation n’est peut-être pas adaptée au niveau de criticité auquel sont soumises les fonctions habituellement rencontrées dans le domaine des transports.

Références

[1] J-F. Boland, et al, Using Matlab and Simulink in a SystemC verification Environment. *2nd North American SystemC User's Group*. Santa Clara, CA, USA. 2004.
[2] C. Warwick. SystemC calls matlab. 2003.
[3] T. Grötker et al, System Design with SystemC. Springer, ISBN 978-1-4020-7072-3, chapter 8, page 131, 2002.
[4] C. De Paula Silva, et al, The development of the Diagnosis Fault System in Electronic Architectures for Commercial Vehicles. *International Truck and bus meeting and exposition, SAE 2000*, 2000.
[5] P. Paulin, et al, Trends in Embedded Systems Technology : An Industrial Perspective. *In Hardware/Software Co-Design*, Kluwer Academic Publishers, 1996.
[6] K. E. Wiegers, Software Requirements. *Redmond: Microsoft Press*. ISBN 978-0735618794, 2003.
[7] T. A. Henzinger, J. Sifakis, The Embedded Systems Design Challenge. *FM 2006*: 1-15, 2006.
[8] T.A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design, *Computer,* pp. 32-40, 2007.
[9] L. Breslau and S. Shenker, Best-effort versus reservations: A simple comparative analysis. *ACM Transactions on Networking*, 1998.
[10] L. Szirmay-Kalos, G. Márton, Worst-Case Versus Average Case Complexity of Ray-Shooting. *Computing*, Vol 61, Issue 2, pp. 103-131, 1998.
[11] X. Crégut, M. Pantel, Aide à la conception des systèmes critiques. *Équipe ACADIE*, IRIT-ENSEEIHT.2010.
[12] R. Wilhelm. The Worst-Case Execution Time Problem— Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, Vol 7, Issue 3, 2008.
[13] N. Halbwachs, Synchronous Programming of Reactive Systems. *Kluwer Academic Publishers*, 1993.
[14] H. Kopetz, G. Bauer, The time-triggered architecture. *Proceedings of the IEEE,*pp.112-126, 2003
[15] P.R. Panda, SystemC: A modeling platform supporting multiple design abstractions. *In Proceedings of the International Symposium on Systems Synthesis (ISSS)*, pp. 75–80. ACM, 2001.
[16] National Instruments tutorial, Shortening the Embedded Design Cycle with Model-Based Design, 2009.
[17] J. Rumbaugh, I. Jacobson, and G. Booch, The Unified Modeling Language Reference Manual. *Addison-Wesley*, 2004.
[18] P.H. Feiler, B. Lewis, and S. Vestal, The SAE Architecture Analysis and Design Language (AADL) Standard: A basis for model-based architecture-driven embedded systems engineering. *In Proceedings of the RTAS Workshop on Model-driven Embedded Systems*, pp. 1–10, 2003.
[19] Y. Sorel, Massively Parallel Systems with Real Time Constraints, the Algorithm Architecture Adequation Methodology. *In Proceedings of Conference on Massively Parallel Computing Systems, MPCS'94*, Ischia, Italy, 1994.
[20] A. Dias, C. Lavarenne, M. Akil, Y. Sorel. Adéquation Algorithme Architecture appliquée aux circuits reconfigurables. *In Actes des Quatrièmes Journées AAA en Traitement du Signal et des Images*, Saclay, France, 1998.
[21] A. L. Sangiovanni-Vincentelli. Defining Platform-Based Design. *In EEDesign*, Fevrier 2002.
[22] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications. *Kluwer Academic Publishers*, 1997.
[23] M. Shawky et al. Architecture exploration and validation methodology for RTP V1: Consideration of functional and extra-functional criteria. *Cesar project Deliverable*, 2010.
[24] J.-C. Laprie, et al, Guide de la sûreté de fonctionnement, *Edition Cépaduès*, ISBN

2.85428.382.1, 1996.
[25] IMdR, Fiches méthodes, 2009.
[26] Analyse par arbre de pannes (AAP), Norme internationale CEI IEC 1025, Octobre 1990.
[27] Norme CEI 812, Technique d'analyse de la fiabilité des systèmes –Procédure d'analyse des modes de défaillance et de leurs effets (AMDE), 1985.
[28] Desroches A., Baudrin D., Dadoun M., *L'Analyse Préliminaire des risques- principes et pratiques, Ed Hermes science,* 2009.
[29] J-F. Monin, Understanding Formal Methods. *Springer*, ISBN 1-85233-247-6, 2003.
[30] M. Edmund, et al, Model Checking. *MIT Press*, ISBN 0-262-03270-8, 1999.
[31] K. Schneider, T. Tuerk and M. Gordon, Model Checking PSL Using HOL and SMV. *Lecture Notes In Computer Science Proceedings of the 2nd international Haifa verification conference on Hardware and software, verification and testing*, pp. 1-15, 2006.
[32] L. Console and O. Dressler, Model-based diagnosis in the real world: lessons learned and challenges remaining. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99).* Stockholm, pp. 1393–1400, Sweden, 1999.
[33] W. Hamscher, et al, Readings in model-based diagnosis. *Morgan Kaufmann Publishers Inc.*, ISBN: 1-55860-249-6, 1992.
[34] J.N. Chatain, Diagnostic par système experts. *Hermes Sciences Publications.* ISBN: 978-2866013769, 1993.
[35] R. Reiter, A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57-95, 1987.
[36] M. Blanke, M. Kinnaert, J. Lunze, M. Staroswiecki, Diagnosis and fault tolerant control, *Springer-Verlag,* 2003.
[37] J.J. Gerlter, Fault Detection and diagnosis in engineering Systems. *CRC Press*, ISBN: 978-0824794279, 1998.
[38] R.J. Patton, P.M. Frank, R.N.Clark, Issues of fault diagnosis for dynamics systems. *Springer Verlag*, 2000.
[39] V. Puig, J. Quevedo, T. Escobet and B. Pulido, on the integration of fault detection and isolation in model based fault diagnosis. *In: Proceedings of DX'05*, pp. 227-232, 2005.
[40] M. Cordier, et al, A comparative analysis of AI and control theory approaches to model-based diagnosis. *14th European Conference on Artificial Intelligence (ECAI 2000)*, Berlin, Germany, 2000.
[41] X. Pucel, L. Travé-Massuyès, A unified point of view on diagnosability. *INSA de Toulouse*, 2008.
[42] F. Nouioua, P. Dague, Diagnosticabilité des systèmes à évènements discrets, *Université Paris Sud –LRI*, Rapport de Recherche N° 1517, 2009.
[43] F. Lin, Diagnosability of discrete-events systems and its applications. *Discrete Event Dynamic Systems,* vol. 4, pp. 197-212, 1994.
[44] M.O. Cordier, L. Travé-Massuyès, and X. Pucel. Comparing diagnosability in continuous and discrete-event systems. *In Proceedings of the 17th International Workshop on Principles of Diagnosis, DX'06*, pp. 55-60, 2006.
[45] M. Nyberg, Criterions for detectability and strong detectability of faults in linear systems. *International Journal of Control*, 75(7):490-501, 2002.
[46] M. M. Polycarpou, A. T. Vemuri, A. R. Ciric, Nonlinear fault diagnosis of differential/algebraic system. *In Proceedings of IFAC Symposium on Fault Detection, Supervision and Safety for Technical Processes*, pp. 510-515, 1997.
[47] O. Dressler and P. Struss, A toolbox integrating model-based diagnosability analysis and automated generation of diagnostics. *In Proceedings of the 14th International Workshop on Principles of Diagnosis, DX'03*, 2003.
[48] M. Sampath, et al, Diagnosability of discrete-event systems. *IEEE transactions On Automatic Control 9(40)*, p-p: 1555-1575, 1995.
[49] S. Jiang, Z. Hiang, V. Chandra and R. Kumar, A polynomial Algorithm for Testing Diagnosability of Discrete Event Systems. *IEEE Transactions on Automatic Control*, 46(8):1318 – 1321, 2001.
[50] T. Yoo et S. Lafortune: Polynomial-Time Verification of Diagnosability of Partially-Observed Discrete-Event Systems. *IEEE Transactions on Automatic Control*, 47(9): 1491 –

1495, 2002.
[51] O.Contant, S. Lafortune and D. Teneketzis, Diagnosability of Discrete Event Systems with modular structure. *Discrete Event Dynamic Systems*, 16(1)-9-37, 2006.
[52] Y. Pencolé, Diagnosability analysis of distributed discrete event systems. *Proceedings of the16th European Conference on Artificial Intelligence (ECAI-04)*, pp. 43-47, 2004.
[53] A. Schumann and Y. Pencolé, Scalable Diagnosability Checking of Event-Driven Systems. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 575-580, 2007.
[54] S. Tripakis, Fault Diagnosis for Timed Automata. *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, LNCS*. vol. 2469, pp. 205-224, 2002.
[55] Y.L. Wen, M. Jeng, Diagnosability of Petri nets. *IEEE International Conference on Systems, Man and Cybernetics*, 5(10-13): 4891-4896, 2004.
[56] S. Haar, A. Benveniste, E. Fabre, C. Jard, Partial Order Diagnosability Of Discrete Event Systems Using Petri Net Unfoldings. *Proceedings of the 42th IEEE Conference on Decision and Control*, Vol 4(9-12), pp. 3748-3753, 2003.
[57] S. Haar. “Diagnosability of Asynchronous Discrete Event Systems in Partial Order Semantics”. *Research Report INRIA*, 2005.
[58] J. Esparza, S. Römer, W. Vogler, An improvement of McMillan’s Unfolding Algorithm. *Formal Methods in System Design*, 20(3): 285-310, 2002.
[59] A. Cimatti, C. Pecheur and R. Cavada, Formal Verification of Diagnosability via Symbolic Model Checking. *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pp. 363-369, 2003.
[60] L.Console, C. Picardi and M. Ribaudo, Diagnosing and diagnosability analysis using pepa. *Proceedings of the 14th European Conference on Artificial Intelligence, (ECAI-02)*, pp. 131-135, 2000.
[61] J. Rintanen and A. Grasien, Diagnosability Testing with Satisfiability Algorithms. *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pp. 532-537, 2007.
[62] G.K. Fourlas, K.J. Kyriakopoulos and N.J. Krikelis, Diagnosability of hybrid systems. *Proceedings of the 10th IEEE Mediteranean Conference on Control and Automation (MED2002)*, 2002.
[63] M. Khlif, M. Shawky, Enhancing Diagnosis Ability for Embedded Electronic Systems Using Co-Modeling. *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering (CIS2E 07)*, 2007.
[64] M. Khlif, M. Shawky, Observability Checking to Enhance Diagnosis of Real Time Electronic Systems. *DS-RT 2008. The 12-th IEEE International Symposium on Distributed Simulation and Real Time Applications*. Vancouver, British Columbia, Canada, 2008.
[65] http://www.aadl.info/aadl/currentsite/
[66] J. Rouillard, Lire & Comprendre VHDL & AMS. *Lulu éditeur*, ISBN 978-1-4092-2787-8, 2008.
[67] M. Bombana and F. Bruschi. Systemc-vhdl co-simulation and synthesis in the hw domain. *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, Washington, DC, USA, 2003.
[68] F. Czerner, and J. Zellmann, Modeling Cycle-Accurate Hardware with Matlab/Simulink using SystemC. *6th European SystemC Users Group Meeting (ESCUG)*. 2002.
[69] P. Kass Hanna, Modélisation et simulation de modèles HW/SW pour améliorer la diagnosticabilité des systèmes électroniques embarqués. *Université de Technolgie de Compiègne*, rapport de projet de fin d’études (sous le co-encadrement de Mohamed Shawky et Manel Khlif), 2009.
[70] O. Tahan, Modélisation et simulation hétérogène pour système embarqué. *Rapport de stage de master de recherche. Université de Technologie de Compiègne*, Juillet 2009.
[71] M. Khlif, O. Tahan, M. Shawky, CO-SImulation Trace Analysis (COSITA) Tool for Vehicle Electronic Architecture Diagnosability Analysis. University of California, San Diego, CA, USA, 2010.

[72] P. Louvel, Systèmes électroniques embarqués et transports. *Dunod*, ISBN 2 10 049510 0, Paris, 2006.
[73] G. Castelli, The Seemingly Unlimited Market for Microcontroller-based Embedded Systems. *IEEE Micro*, pages 6-8, 1995.
[74] D. Paret, Multiplexed Networks for Embedded Systems: CAN, LIN, FlexRay, Safe-by-Wire. *Wiley*, ISBN 978-0470034163, 2007.
[75] D. Paret, Le bus CAN description : de la théorie à la pratique. *Dunod*, ISBN 978-2100047642, 2009.
[76] BOSCH, CAN specification version 2.0, *Robert BOSCH GmbH*, 1991.
[77] H. Shraim et al. Smart Distance Keeping: Modeling and Perspectives for Embedded Diagnosis. *ISMS 2010: 1st International Conference on Intelligent Systems, Modelling and Simulation*, Liverpool, United Kingdom, pp. 27-29, 2010.
[78] X. Claeys, et al, Chauffeur Assistant Functions. *Report restricted to RENAULT TRUCKS,* European Contract number IST-1999-10048, Lyon, FRANCE, 2003
[79] M. Khlif, M. Shawky, Co-modelling and simulation with multilevel of granularity for real time electronic systems supervision. *EUROSIM/UKSIM 2008. 10th International conference on computer Modelling and Simulation*.Cambridge, England, 2008.
[80] http://www.mototron.com/support/, *Site officiel de Mototron.*
[81] http://www.mototron.com/products/MotoHawkSuite/MotoHawk/, *Site officiel de Mototron*.
[82] http://www.ghs.com/products/MULTI_IDE.html, *Site officiel de GreenHills.*
[83] http://www.mototron.com/products/MotoHawkSuite/MotoTune/, *Site officiel de Mototron*.
[84] O. Tahan, Simulation et parsing de modèles HW/SW Pour améliorer l'Observabilité. *Université de Technolgie de Compiègne*, rapport de projet de fin d'études (sous l'encadrement de Mohamed Shawky), 2008.
[85] http://www.vector-worldwide.com/vi_canalyzer_en.html, *Site officiel de Vector*.

Livrables de projets

[1] M. Khlif, M. Shawky, Bilan sur l'analyse de l'observabilité du diagnostic embarqué. Livrable D2.3.1. *Projet Diafore*. Rapport de Contrat ADEME. Mai 2008**.**

[2] M. Khlif, M. Shawky, Plateforme d'émulation d'ECU. Livrable D2.3.1. *Projet Diafore*. Rapport de Contrat ADEME. Janvier 2009**.**

[3] M. Khlif, M. Shawky, Démonstrateur de l'outil d'analyse. Livrable D2.3.1. *Projet Diafore*. Rapport de Contrat ADEME. Septembre 2009**.**

[4] M. Khlif, M. Shawky, Survey of state-of-the-practice and state-of-the-art in safety and diagnosability. Delivrable D_SP1_R5.8_M2. *Projet CESAR*. Décembre 2009

[5] M. Khlif, M. Shawky, Safety-Diagnosability requirements specification V2. Delivrable D_SP1_R5.3_M2. *Projet CESAR*. Janvier 2010.

Annexes

Annexe A. Co modèle SystemC-Simulink des deux calculateurs ECU1 et ECU2

Nous présentons ci-après le code SystemC du co-modèle de chacun des deux ECUs adoptant la technique d'interfaçage SystemC–Simulink :

```
SC_MODULE(ECU1)
{
	sc_inout<sc_uint<40>> CAN; );// 40 bits : longeur de la trame
	sc_in_clk CLK;
	sc_uint<5> id;
	sc_uint<3> length;
	sc_uint<8> data[4];
	Engine *ep;
	mxArray *result;
	double *resPr;
	mxArray *input;
	mxArray *VVAR,*DS,*DR;
	double *VVAR_Pr,*DS_Pr,*DR_Pr;
	double vva,vvar,ds,dr, memory_vva;
	sc_event e1;

	clock_t tbegin, tend, delai ;


	void SendCAN(sc_uint<5> id, sc_uint<3> length, sc_uint<8> *data)
	{
		sc_uint<40> trame, trame2;
		sc_uint<32> data_block;
		for(int i = 0; i<length; i++)
		{
			for(int j = 0; j<8; j++)
			{
				data_block[31-(8*i+j)] = data[i][7-j];
			}
		}
		trame = (id,length,data_block);
		trame2=trame;
		CAN.write(trame);
	};
```

À l'intérieur du constructeur du module ECU1, un processus Matlab se lance à l'aide de la fonction « *engOpen* » ; cette fonction retourne un pointeur vers « *Matlab Engine* ». Ensuite, la simulation du modèle Simulink se réalise en combinant les deux instructions « *engEvalString* » de C++ et « *set_param* » de Matlab avec les paramètres convenables. Nous créons quatre variables de type « *mxArray* » tels que « *Input* », « *VVAR* », « *DS* » et « *DR* » sont des pointeurs vers ces variables et qui désignent respectivement la vitesse du véhicule d'avant, la vitesse du véhicule d'arrière, la distance de sécurité et la distance relative. Enfin, nous précisons, à l'aide de « *SC_METHOD* » et « sensitive », qu'il faut lancer les méthodes « *radar()* » et « *sdk()* » à chaque fois que le port d'entrée subit une variation.

```
	SC_CTOR(ECU1)
	{
		id = 17;
		length =1;
		memory_vva=0;
		ep = engOpen("\0");
		engEvalString(ep, "sim('SDK_input')");
		engEvalString(ep,
"set_param('SDK_input','SimulationCommand','start');set_param('SDK_input','
SimulationCommand','pause');");

		engEvalString(ep, "sim('SDK')");
		engEvalString(ep,
"set_param('SDK','SimulationCommand','start');set_param('SDK','SimulationCo
mmand','pause');");
		input = mxCreateDoubleMatrix(1,1,mxREAL);

		sc_trace_file *tf=
sc_create_vcd_trace_file("fichier_trace_ECU1");
		sc_trace(tf,CLK,"Clock");
		sc_trace(tf,vva,"VVA");
		sc_trace(tf,vvar,"VVAR");

		sc_trace(tf,ds,"DS");
		sc_trace(tf,dr,"DR");
		sc_trace(tf,data[0],"Data");



		SC_METHOD(radar);
		sensitive<<CLK; // la méthode radar() est sensible aux tops
d'horloge


		SC_METHOD(sdk);
		sensitive<<e1; // la méthode sdk() est sensible à l'événement
e1 (fin de l'exécution de la méthode radar())
		dont_initialize();


	}
```

A chaque exécution de la méthode « *radar()*», la valeur de la variable « *result*» est pointé, qui désigne la vitesse du véhicule en avant, par le pointeur « *resPr* » car la fonction « *mxGetPr (result)* » retourne l'adresse du premier élément du champ « *result* » relatif au « *mxArray* » pointé par « *result*». Ensuite, la variable « *vva* » prend la valeur de la vitesse du véhicule avant, elle ne change de valeur que si la valeur en cours de la vitesse est différente de la valeur qui l'a précédé « *memory_vva* » et notifie l'événement « *e1*» afin de pouvoir déclencher « *SDK()*» (« *e1.notify()* »). Ainsi, la méthode « *SDK()*», ne se déclenche pas successivement pour la même valeur, elle ne se déclenche que lorsqu'il y a un changement de valeur de « *vva* ». La méthode « radar () » simule approximativement le temps que prend l'onde pour revenir au radar après avoir été réfléchie par le véhicule en avant (1000ns ou 5000ns).

```
	void radar()
	{
```

```
            engEvalString(ep,
"set_param('SDK_input','SimulationCommand','step');");
            engEvalString(ep,
"set_param('SDK_input','SimulationCommand','continue');set_param('SDK_input
','SimulationCommand','pause');");
            engEvalString(ep,"R = x(length(x))");
            result = engGetVariable(ep,"R");
            resPr = mxGetPr(result);
            vva= (double) *resPr; // vva : Vitesse Véhicule Avant

            if (vva != memory_vva)
            {
            if (vva < memory_vva)
            {next_trigger(1000,SC_NS);}
            else
            {next_trigger(5000,SC_NS);}
            e1.notify();
            }
            memory_vva=vva;


      }
```

Au moment de l'exécution de la méthode « *SDK ()*» et en paramétrant convenablement la fonction « *memcpy* », une valeur correpondante est affectée au « *mxArray* ».
La fonction « *engPutVariable (ep, "in", input)* » permet de déclarer dans l'environnement Matlab, pointé par « *ep* », une variable nommée « in » dont le bloc mémoire correspondant est pointé par « input».
Les valeurs des ports ne sont pas encore transmises vers Simulink. Pour ceci, Matlab doit exécuter la commande « *set_param ('SDK/input', 'value', 'in')* ». Cette commande affecte le bloc « *Vitesse_Vehicule_Arriere* » du modèle Simulink par une valeur égale à celle de la variables « *in* » de Matlab.
Depuis l'environnement SystemC, la simulation du modèle Simulink avance d'un seul pas via la commande « *set_param ('SDK_input','SimulationCommand', 'step')* ». La génération d'une valeur d'input étant exécutée, le bloc « *ToWorkspace* » du modèle Simulink envoie le résultat vers Matlab. Il le sauvegarde à la fin d'un vecteur « *vvar* ». La commande « *x1 = vvar (length (vvar))* » permet de récupérer le résultat dans une variable de type « *Matlab Array* » (scalaire) et la fonction « *engGetVariable* » retourne un pointeur vers cette variable, ce qui permet d'accéder à la variable « vvar » de Matlab depuis SystemC. La fonction « *mxGetPr* » retourne l'adresse du premier élément du champ data de la variable « *vvar* », sauvegardée dans la variable « *VVAR_Pr* ». Finalement, il suffit d'écrire la valeur pointée par « *VVAR_Pr* ».

```
void sdk()
      {

            memcpy((void *) mxGetPr(input), &vva, sizeof(double));
            tbegin=clock();// correspond à l'instant zéro du temps
d'éxécution de sdk dans l'environnement Simulink/Matlab

            engPutVariable(ep,"in",input);
            engEvalString(ep,"set_param('SDK/input','value','in');");
            engEvalString(ep,
"set_param('SDK','SimulationCommand','step');");
```

```
            engEvalString(ep,
"set_param('SDK','SimulationCommand','continue');set_param('SDK','Simulatio
nCommand','pause');");

            tend=clock();// correspond à la fin de l'exécution de sdk

            engEvalString(ep,"x1 = vvar(length(vvar));");
            VVAR = engGetVariable(ep,"x1");
            // VVAR : Vitesse Vehicule ARriere
            VVAR_Pr = mxGetPr(VVAR);
            vvar = *VVAR_Pr;

            engEvalString(ep,"x2 = ds(length(ds));");
            DS = engGetVariable(ep,"x2"); // DS : Distance de Sécurité
            DS_Pr = mxGetPr(DS);
            ds = *DS_Pr;

            engEvalString(ep,"x3 = dr(length(dr));");
            DR = engGetVariable(ep,"x3"); // DR : Distance Relative
            DR_Pr = mxGetPr(DR);
            dr = *DR_Pr;

            data[0] = (int) *VVAR_Pr;
            SendCAN(id,length,data);
            printf("%3.2lf--%3.2lf--%3.2lf--%3.2lf--
%3.2lf\n",vva,vvar,ds,dr);

            delai=(tend-tbegin);  // « delai » correspond au nombre de
cycles (ou tops) d'horloge nécéssaires pour l'exécution de la fonction SDK
dans l'environnement Simulink/Matlab
            cout<<delai<<endl; //affiche le nombre de cycles (ou tops)
d'horloge nécéssaires pour l'exécution de la fonction SDK dans
l'environnement Simulink/Matlab
}


};

----- *** ----- *** ----- *** ----- *** ----- *** ----- *** ----- *** -----

SC_MODULE(ECU2)
{
      sc_inout<sc_uint<40>> CAN;
      sc_in_clk CLK;
      sc_uint<5> id;
      sc_uint<3> length;
      sc_uint<8> data[2];
      Engine *ep;
      double pvar, vvar;


      SC_CTOR(ECU2)
      {
            SC_METHOD(moteur);
            dont_initialize();
            sensitive<<CLK.pos();
            sc_trace_file *tf=
            sc_create_vcd_trace_file("fichier_trace_ECU2");
            sc_trace(tf,CLK,"Clock");
            sc_trace(tf,data[0],"Data");
```

```
	}

	void ReadCAN(sc_uint<5> *id, sc_uint<3> *length, sc_uint<8> *data)
	{
		sc_uint<40> trame;
		sc_uint<32> data_block;
		trame = CAN.read();
		if (*id == trame.range(39,35))
		{
			*length = trame.range(34,32);
			data_block = trame.range(31,0);
			for(int i = 0; i<*length; i++)
			{
				data[i] = data_block.range( 31 - 8*i, 31 - (8*i +
7));
			}
		}
	}

	void moteur()
	{
	id = 17;
	ReadCAN(&id,&length,data);
	vvar= double (data[0]);
	next_trigger(40,SC_US);// delai d'acheminement de la trame

	}
};
```

Annexe B. OBSAN

OBSAN permet d'effectuer une analyse sur le fichier de traces généré par la simulation. Il permet de vérifier si le système est suffisamment prêt à implanter le diagnostiqueur dans les cycles libres (pour un diagnostic en temps réel).

Dans son interface graphique, nous devons saisir les valeurs de :

- certaines caractéristiques du diagnostiqueur tel que le temps de calcul nécessaire par cycle (cycle de diagnostic),
- les caractéristiques du calculateur sur lequel devra être implanté le diagnostiqueur (temps de calcul nécessaire pour les entrées /sorties, etc.)
- et la durée de la périodicité du diagnostic en termes de nombre de cycles d'horloge (Figure 57).

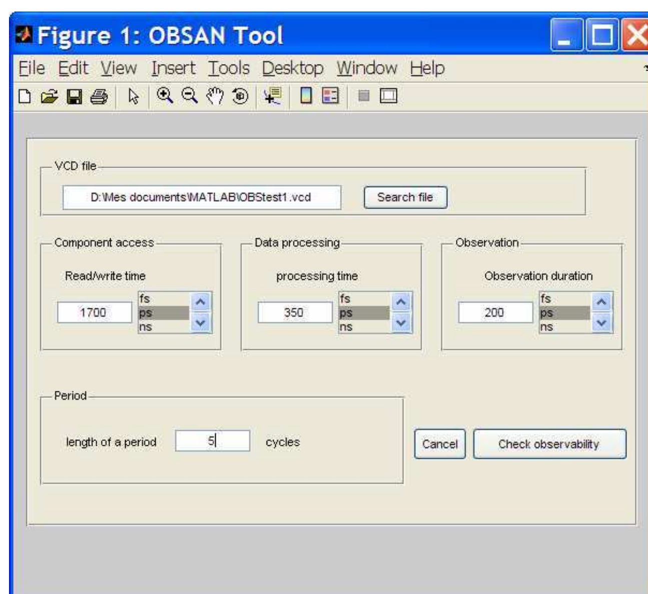

Figure 57. Interface graphique de la saisie des entrées du module OBSAN

Le résultat de l'analyse est un chronogramme qui indique les fenêtres temporelles disponibles pour l'implantation du processus de diagnostic souhaité en attribuant la valeur « 1 » à Observability (Figure 58).

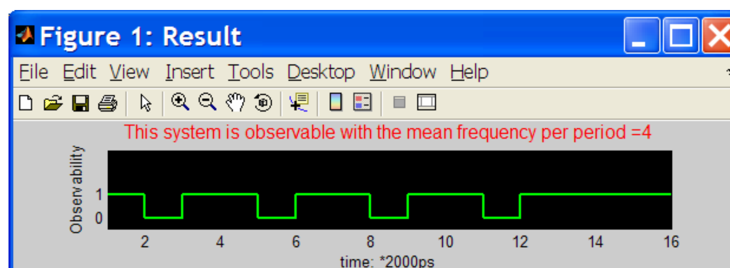

Figure 58. Interface graphique du résultat (OBSAN)
Sur l'axe vertical, l'observabilité (ou disponibilité temporelle) prend une valeur booléenne : observabilité="1" si le cycle correspondant à l'axe horizontal est suffisant pour l'observation, autrement, l'observabilité="0".

Annexe C. La différence entre deux fichiers traces

Pour l'exemple de la SDK, la comparaison a conduit à des divergences légères dues la plupart du temps à des écarts de fraction des valeurs de certaines variables, provenant probablement des différences de précision des calculs mathématiques. Deux différences entre les deux fichiers sont surlignées en gris dans la colonne droite du Tableau 5.

Tableau 5. Différences entre deux fichiers trace

Fichier trace issu de la co-simulation de SDK	Fichier trace issu de l'émulation de SDK
#10618750 r80.40000000000001 aaa r79.99199999999999 aab r44.44 aac r44.46209995 aad 0aaf #10625000 1aaf #10631250 0aaf #10637500 1aaf #10643750 0aaf	#10618750 r80.40000000000000 aaa r79.99199999999999 aab r44.44 aac r44.46209999 aad 0aaf #10625000 1aaf #10631250 0aaf #10637500 1aaf #10643750 0aaf

Annexe D. COSITA

L'interface graphique d'accueil de COSITA permet l'accès aux différents modules de l'outil :

- Lancement de la co-simulation à partir d'un co-modèle (SystemC-Simulink).
- OBSAN, pour l'analyse de la disponibilité temporelle et l'observabilité.
- Log2vcd, pour convertir les fichiers du format log au format vcd.
- SIMECO, pour la comparaison de deux fichiers traces au format vcd (Figure 59).

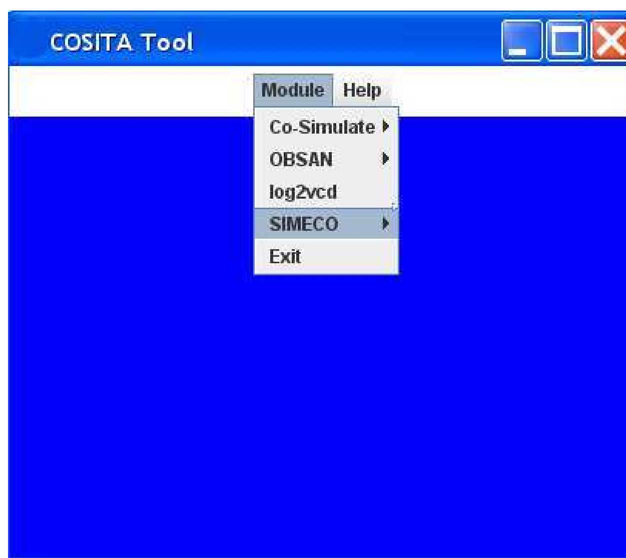

Figure 59. Interface graphique d'accueil de COSITA

Annexe E. La Librairie MOTOHAWK

Motohawk est une librairie de Simulink contenant 165 blocs différents. Elle est utile dans le domaine de l'électronique embarqué de l'automobile. Dans ce paragraphe nous décrivons brièvement comment l'utiliser pour programmer une architecture électronique formée de plusieurs ECUs connectés par un bus CAN.

Afin de programmer un ECU, il faut lui associer un modèle complètement développé sous Simulink à l'aide de la librairie Motohawk (Figure 60).

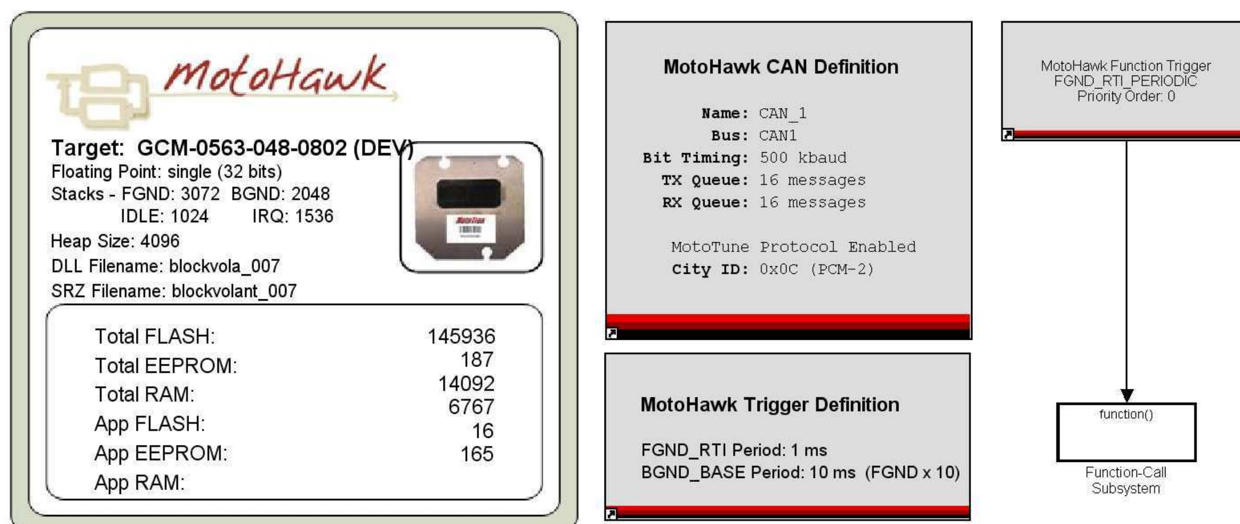

Figure 60.Modèle Simulink avec des blocs Motohawk pour la programmation d'un ECU

Dans ce modèle de base, il ya les blocs suivants :

- *motohawk_target_def :* permet de définir le type de l'ECU sur lequel le programme doit être installé
- *CAN definition :* permet de définir le bus CAN auquel l'ECU doit être connecté, ainsi que ses principales caractéristiques comme le « bit timing »
- *motohawk_trigger_def :* initialise les horloges de base du modèle
- *motohawk_trigger :* génère comme sortie une horloge dépendante des horloges de base, cette horloge déclenche périodiquement une tâche ou une fonction
- *function-call subsystem :* à l'intérieur de ce bloc un modèle Simulink décrivant la fonctionnalité de l'ECU doit être implémenté. Plusieurs blocs standards ou Motohawk y sont généralement utilisés.

Annexe F. MOTOTUNE

MOTOTUNE est l'outil utilisé pour la programmation de l'ECU, pour visualiser les E /S et pour faire la calibration afin de contrôler des paramètres internes, les modifier et pour créer des fichiers texte de traces.

Pour la programmation il faut cliquer sur le bouton **Program**, choisir le fichier de type.srz qui contient le code C généré, choisir le port de programmation qui peut être PCM1, PCM2,…, PCMn suivant le nombre d'ECU reliées et suivant le numéro de port programmé avant dans le CAN définition bloc dans le schéma de Simulink. Une fois programmée, nous pouvons visualiser les E/S de l'ECU tout en créant un nouveau DISPLAY et en choisissant le type de grille désiré. Pour enregistrer un fichier de trace il faut cliquer d'abord sur le bouton **Chart,** ensuite sur le bouton **start_logging** (Figure 61).

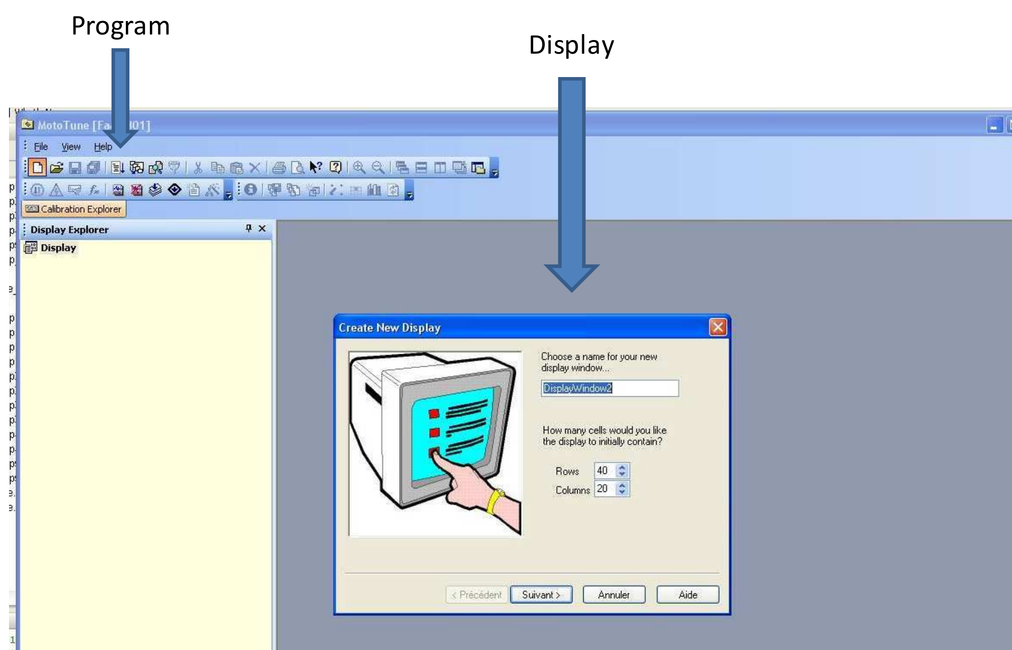

Figure 61. Interface graphique de Mototune

Annexe G. Programme CAPL pour la génération du fichier .LOG

```
{    message 0x01 msg1={dlc=8};
     message 0x02 msg2={dlc=8};
     mstimer timer1; //definir un timer : unité par défaut « ms »
     dword x,y,clk,timestamp;
     dword handle;
     char s[20] ="";
}
on start
{ x=1;
     y=0;
     clk=0;
     timestamp=0;
     settimer(timer1, 0.025); //fréquence de l'horloge =25 ns
     setfilepath("c:\\",1);
     handle = openfilewrite("testcan.txt",0);
     ltoa(45,s,10);
     write(s);
     if ( handle!=0)
fileputstring("date 28/05/2008 \n",200,handle);
 fileputstring("this is our first test\n\n",200,handle);
 fileputstring("TimeStamp (mS)\tSignal\t\n",200,handle);
//fileclose(handle);   }

on key 's'
{//writetolog("this is the begining");}
on key 'a'
{
     //writetolog(" this is the end");
}
on message 0x01 //msg1
{x=1;
  write("x eguale %d",x);}
on message 0x02 // msg2
{  x=0;
     write("x eguale %d",x);}
on timer timer1
{  timestamp=timestamp+1;//pour calculer l'instant...
     if (x==1)  {y=!y;}
     ltoa(timestamp,s,10);
     fileputstring(" ",200,handle);
     fileputstring(s,200,handle);
     fileputstring("\t",200,handle);
     ltoa(y,s,10);
     fileputstring(s,200,handle);
     fileputstring("\t\n",200,handle);
     settimer(timer1,0.025);}
on key 't'
{fileclose(handle);}
```

Annexe H. log2vcd

```
#include <iostream>
#include <stdio.h>
#include <string>
#include <sstream>
#include <vector>
#include <fstream>
using namespace std;

void dec2bin(int decimal, char *binary);// Decimal to binary

// Fonction pour trouver la longeur d'une chaine de caractere
int StringLength(std::string inputString)
{
   int length = 0;
   for (int i = 0; inputString[i]!= '\0'; i++)
      length++;
   return length;
}

//fin de la fonction StringLength
int main()
{
// Declaration des Variables
  char nomlog[42];
 char nomvcd[42];
    char c;
    std::cout<<"Les fichiers doivent etre sous la forme c:/Sheet7.log ou c:/test.vcd\n\n";
    std::cout<<"Veuillez donner le nom du fichier LOG de mototune : ";
    scanf("%s",&nomlog);
    std::cout<<"\n\nVeuillez donner le nom du fichier VCD: ";
    scanf("%s",&nomvcd);
   std::ifstream filemoto(nomlog);
   std::ofstream filevcd(nomvcd);
   if (!filevcd)
   {
      std::cout<<"Le fichier .VCD ne peut etre ouvert...";
   }
   if(!filemoto.good())
     {
      std::cout<<"L'adresse du fichier .LOG est fausse...";
   }
 if ((filevcd)&&(filemoto.good()))
 {
   std::string s;
   std::string date;
   std::string specs;
   std::string nbsignaux;
   std::string ligne;
```

```
 bool clk=0; char lu;
char caractere;

   int j=0;
   int i=1;
   int k=0;
   int compteur=0;
   int decimal;
   char binary[80];
   int alpha ;
   bool m=false;
   bool n=false;
   int time=0;
   std::istringstream s1;
   std::istringstream s2;
   std::stringstream s3;
   int numero;

//Lecture des 3 premières lignes du fichier log.
//-------------------------------------------------
     std::getline(filemoto,date);
     std::getline(filemoto,specs);
     std::getline(filemoto,nbsignaux);//on fait 2 fois car la 3eme ligne est seulement un enter
     std::getline(filemoto,nbsignaux);

          for (int k=0; k<nbsignaux.length() ; k++)
   {   caractere=nbsignaux[k];
    if (caractere=='\t') compteur=compteur+1;}

 //Le nombre des signaux est égal à compteur connu donc il faut créer le tableau des
 stringpour les noms des signaux
      std::string tableau[compteur];

     // Initialisation du tableau avec les valeurs
      i=0;
      for (int k=0; k<nbsignaux.length() ; k++)
    {
       caractere=nbsignaux[k];
       tableau[i]+=caractere;
       if (caractere=='\t') i++;
    }

  //Elimination des espaces et des tab dans les noms des signaux:
    for (k=0;k<compteur;k++)
    {
     s="";
     for (j=0;j<StringLength(tableau[k]);j++)
     {
         if ((tableau[k][j]!=' ')&&(tableau[k][j]!='\t')) s+=tableau[k][j];
     }
```

```
    tableau[k]=s;
    }
    // Fin de l'élimination des espaces...
    //--------------------------------------------
    // Ecriture de l'entête du fichier .vcd
    //---------------------------------
    filevcd<<"$date\n";
    filevcd<<date<<"\n";
   filevcd<<"$end\n\n";
  filevcd<<"$timescale\n";
    filevcd<<"1 ms\n";
    filevcd<<"$end\n\n";
    filevcd<<"$scope module LOGfile $end\n";
    for (j=1;j<compteur;j++)
    {
       filevcd<<"$var port    32  "<<tableau[j]<<"    "<<tableau[j]<<"    "<<"$end\n";
    }
   filevcd<<"$upscope  $end\n";
    filevcd<<"$enddefinitions    $end\n";
    filevcd<<"\n";

    //Fin de la premiere partie du fichier .VCD
    // Lecture des autres données et la manipulation
    std::string valeursanc[compteur];
    std::string valeursnouv[compteur];
    int numberanc[compteur];
    int numbernouv[compteur];

    //Ecriture des conditions initiales...
    filevcd<<"$dumpvars\n";
    //filevcd<<clk<<"clk\n";
    for (j=1;j<compteur;j++)
    {  filevcd<<"b0"<<"    "<<tableau[j]<<"\n";
    }
filevcd<<"$end\n";

 //Fin de l'écriture des conditions initiales
    for (k=0;k<compteur;k++)
    {
       valeursanc[k]="0";
       valeursnouv[k]="0";
    }
    while ( !filemoto.eof() )
    {
       std::getline( filemoto, ligne );
       i=0;
       for ( k=0; k<ligne.length() ; k++)
          {
             caractere=ligne[k];
             valeursnouv[i]+=caractere;
```

```
            if (caractere=='\t') {i++;}
          }
      //clk=!clk;
      m=false;
      //filevcd<<clk<<"clk\n";
      time=1;
      for (k=0;k<compteur;k++)
      {  s1.str(valeursanc[k]);
         s2.str(valeursnouv[k]);
         s1>>numberanc[k];
        s2>>numbernouv[k];
   if ((k==0)&& (n==true) ) {time=numbernouv[k];}
         n=true;
          if (m==false) {filevcd<<"\n#"<<time<<"\n";}
          m=true;
         if ((numberanc[k]!=numbernouv[k])&&(k!=0))
           { dec2bin(numbernouv[k],binary);
             for (alpha=0;alpha<StringLength(binary);alpha++)
             {
                if (alpha!=StringLength(binary)-1)
                  binary[alpha]=binary[alpha+1];
                if (alpha==StringLength(binary)-1)
                  binary[alpha]=' ';
             }
             filevcd<<"b"<<binary<<""<<tableau[k]<<"\n";
           }
         numberanc[k]=numbernouv[k];
         valeursnouv[k]="";
      }
      }
   filemoto.close();
   filevcd.close();
   std::cout<<"Conversion est terminee\n";
   std::cout<<"----------------------";
   }
 int x;

    do {
    cout << "\n \nPress a key to stop.\n";
    cin >> c;
    } while (!c);
     return 0;
    }

    // Fonction pour trouver la valeur binaire d'un integer
    void dec2bin(int decimal, char *binary)
    {
    int k = 0, n = 0;
    int neg_flag = 0;
    int remain;
```

```
int old_decimal; // for test
char temp[80];

// take care of negative input
if (decimal < 0)
{
decimal = -decimal;
neg_flag = 1;
}
do{old_decimal = decimal; // for test
remain = decimal % 2;

// whittle down the decimal number
decimal = decimal / 2;

// this is a test to show the action
printf("%d/2 = %d remainder = %d\n", old_decimal, decimal, remain);

// converts digit 0 or 1 to character '0' or '1'
temp[k++] = remain + '0';
} while (decimal > 0);
if (neg_flag)
temp[k++] = '-'; // add - sign
else
temp[k++] = ' '; // space

// reverse the spelling
while (k >= 0)
binary[n++] = temp[--k];
binary[n-1] = 0; // end with NULL
}

//fin de la fonction binary
```

Annexe I. Installation de la librairie « engine.h »

Pour installer la librairie « engine.h », il faut :

1. Lancer « Visual Studio » et créer une nouvelle « Console Application »
2. Cliquer « Tools _ Options…_Projects and Solutions _VC++ directories »

- Dans **Show directories for : Library files**, cliquer l'icone New et ajouter la librairie de Matlab (Figure 62)

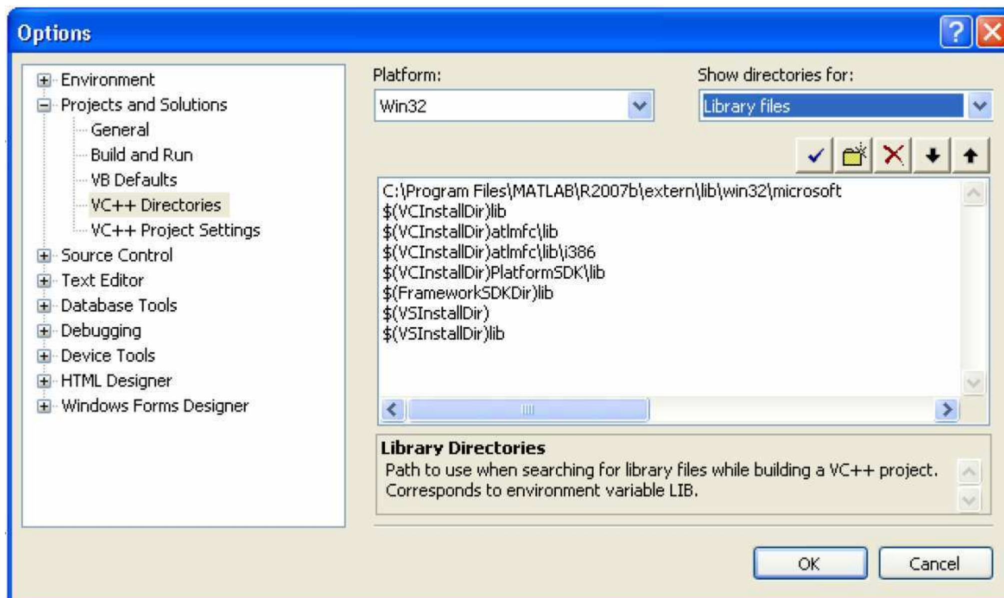

Figure 62. Fenêtre "Library files"

- Dans **Show directories for : Include files**, cliquer l'icone **New** et écrire votre chemin Matlab (Figure 63).

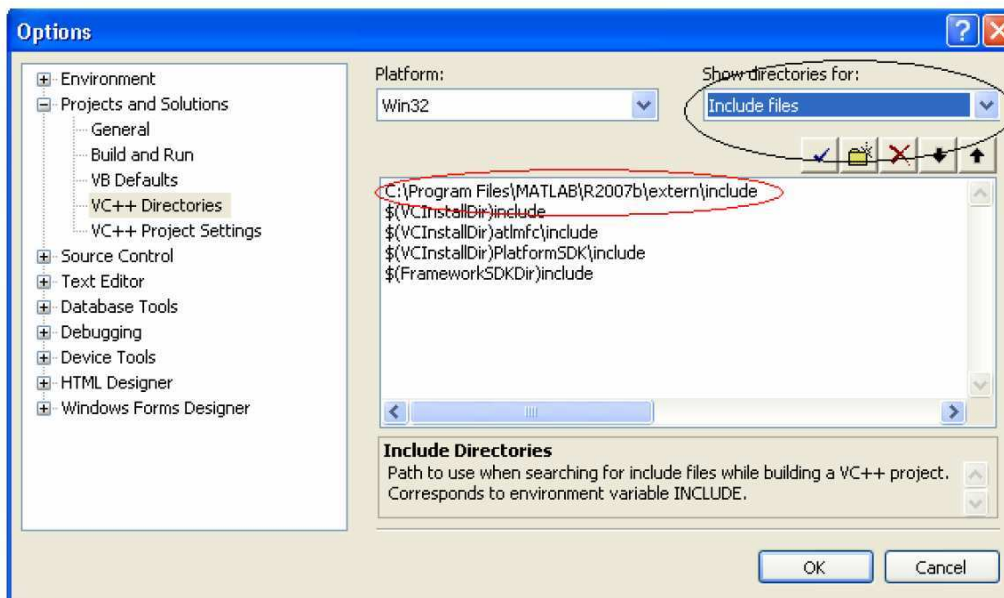

Figure 63. Fenêtre "Include Files"

Cliquer sur **Additional Dependecies** (Figure 64).

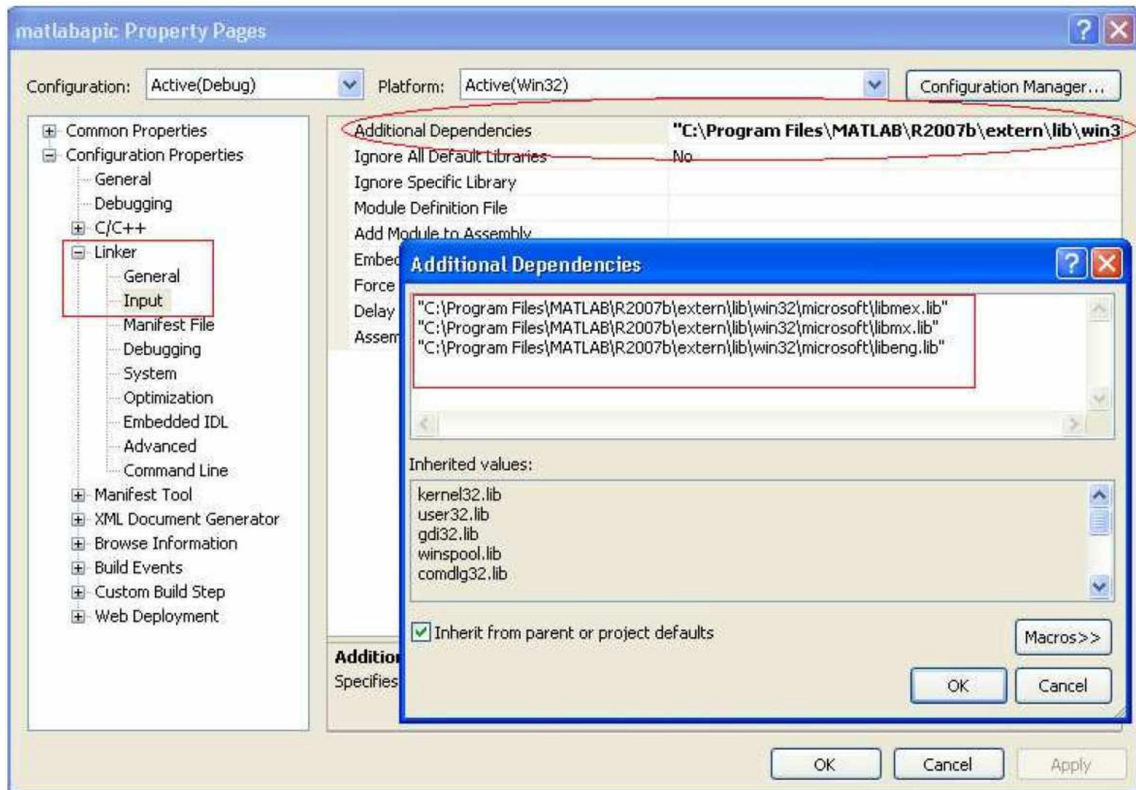

Figure 64. Fenêtre Additional Dependencies

Annexe J. Le protocole CAN

Le protocole CAN (*Controller Area Network*) de Bosch est un protocole de communication série assurant la diffusion fiable de messages sur un bus à faible coût. C'est un protocole mature (expérimentations depuis 1986) qui définit une surveillance très stricte du bus.
La gestion du bus est dirigée par les événements (*Event-Triggered*). Le protocole CAN a fait l'objet d'une normalisation ISO 11898[5].

Principales caractéristiques

Les principales caractéristiques du bus CAN sont :

- Bus multi-maîtres asynchrone. Il n'y a pas de mémoire commune, pas de contrôleur de trafic, pas d'horloge globale et une station peut émettre dès que le bus est libre,
- la notion de priorité pour les messages. Résolution des collisions sans destruction du message le plus prioritaire,
- diffusion fiable de messages. Il y a un acquittement par toutes les stations reliées au bus. La détection d'une erreur par une station est diffusée à toutes les autres,
- distinction entre les erreurs passagères et les erreurs permanentes,
- les messages sont de taille limitée (au plus 8 octets),
- le débit maximal du bus est de 1Mbit/s, pour un bus limité à 40m. Le débit maximal diminue en fonction de la longueur du bus (jusqu'à 125Kbits/s pour 500m).

Le bus CAN peut être dans deux états différents, l'état récessif correspondant au « 1 » logique et l'état dominant correspondant au « 0 » logique. Les différentes stations peuvent émettre sur le bus et « écouter » le bus. S'il y a émission d'un bit dominant par une station et d'un bit récessif par une autre, le bus ne retiendra que le bit dominant (fonction ET câblée, en logique positive). Cette propriété se retrouve alors dans le Tableau 6 :

Tableau 6. Bus CAN : Émission-détection

émission	détection	Interprétation par le nœud
récessif (1)	récessif	Toutes les autres stations émettent un bit récessif
	dominant	Une ou plusieurs autres stations émettent un bit dominant
Dominant(0)	récessif	Il y a une erreur matérielle grave
	dominant	Les autres stations émettent des bits récessifs ou dominants

Quand le bus est au repos, toutes les stations émettent un état récessif et reçoivent un état récessif.

Types de messages

Une trame de données contient un identifiant et des données (jusqu'à 8 octets). C'est l'identifiant qui permet de renseigner sur la nature des données. Par exemple dans un contexte automobile, il est possible d'avoir une trame ou l'identifiant 0x34 correspond à la vitesse du véhicule. Les données seront interprétées alors en conséquence. Dans cette approche, tous les nœuds CAN sur le bus reçoivent le message. Il existe alors des systèmes de filtrage (spécifiques à chaque composant) pour ne pas solliciter les processeurs à chaque nouveau message. Tous les nœuds qui ont besoin de cette information l'obtiennent avec la même trame. Ainsi, la communication ne s'effectue pas d'un émetteur vers un récepteur comme c'est le cas avec l'Ethernet par exemple ou chaque nœud possède une adresse (IP).

[5] ISO 11898 : Road Vehicles – Controller area network (CAN)

Il existe deux types de messages (Figure 65):

- ***Une trame de données (DATA FRAME) :*** elle contient des données de 0 à 8 octets. A chaque message est associé un identifiant de 11 bits (trame standard) ou de 29 bits (trame étendue). L'utilisation de trames de 29 bits permet d'utiliser des modes de filtrages plus efficaces (basés sur des masquages de bits), mais en augmentant la taille de la trame (et donc le temps de transmission) ;
- ***Une trame de requête (REMOTE FRAME) :*** elle ne contient aucune donnée. Lorsqu'un nœud envoie une trame de requête, il attend du nœud qui peut envoyer la donnée demandée (celle avec le même identifiant) qu'il fasse une transmission. Cette méthode peut être utilisée pour des trames non périodiques, comme du diagnostic en ligne par exemple.

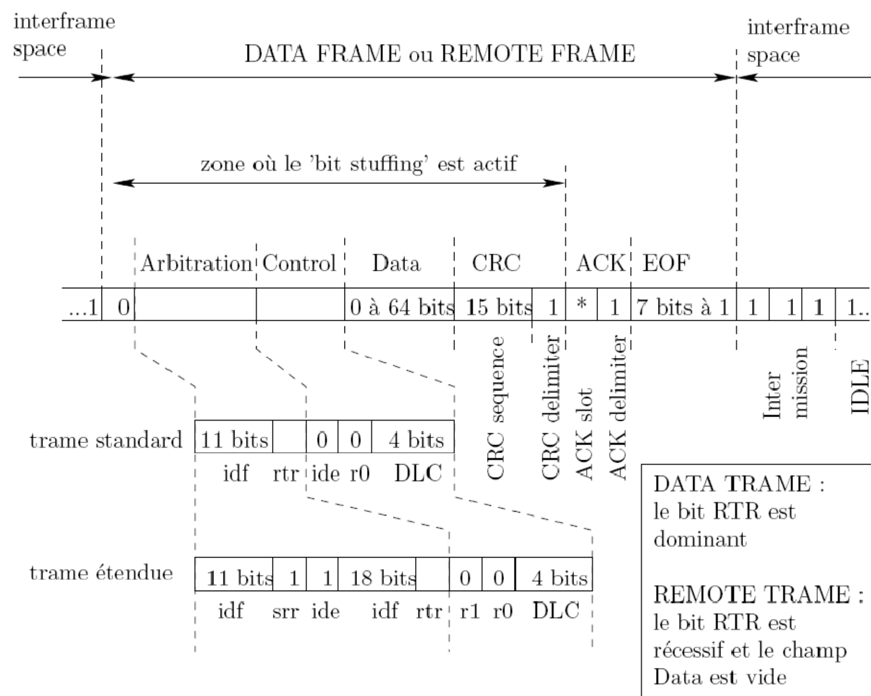

Figure 65. La trame CAN

Envoi des messages

Dès que le bus est libre, un nœud peut commencer une émission, car il n'y a pas de contrôleur de bus. Cependant, il résulte que plusieurs nœuds peuvent prendre simultanément la décision d'émettre. Il y a alors collision entre au moins deux stations. L'intérêt du bus CAN dans ce cas est que le message le plus prioritaire n'est pas détruit (contrairement au protocole Ethernet).

La résolution des collisions est basée sur le codage récessif / dominant des données : durant l'émission des premiers bits, si un nœud émet un bit récessif et reçoit un bit dominant, cela signifie qu'une autre station émet et qu'elle est plus prioritaire. Le nœud arrête alors d'émettre car il a perdu l'arbitrage.