Table des matières

Table des illustrations

Introduction

Introduction

L'introduction de nouvelles prestations d'aide à la conduite automatisée, de sécurité active, par exemple, fait croître le nombre de calculateurs dans les véhicules actuels. Aujourd'hui, l'architecture électronique est composée en moyenne d'une vingtaine de calculateurs partageant de nombreuses informations (ex : vitesse, état du moteur), via des réseaux locaux. Les applications embarquées automobiles sont encore majoritairement des solutions propriétaires indépendantes, mais le regroupement de plusieurs fonctions connexes sur un même calculateur se pratique de plus en plus souvent pour économiser des ressources. Cette tendance sera plus vraisemblablement favorisée dans l'avenir avec des calculateurs plus puissants. La Figure 1 compare de façon très synthétique un réseau de calculateurs représentatif des véhicules actuels et le type de réseaux étudiés, à venir dans un futur proche (calculateurs moins nombreux mais hébergeant plus d'applications). L'organisation du logiciel, à l'intérieur de chaque calculateur, est représentée par une superposition de bandes colorées, correspondant à des couches d'abstraction logicielles. Il apparaît clairement, que la complexité grandissante des systèmes de contrôle électronique se traduit par une structure du logiciel embarqué de plus en plus évoluée. En effet, le multiplexage de plusieurs applications sur un calculateur requiert un support d'exécution capable de gérer un système multitâches temps réel.

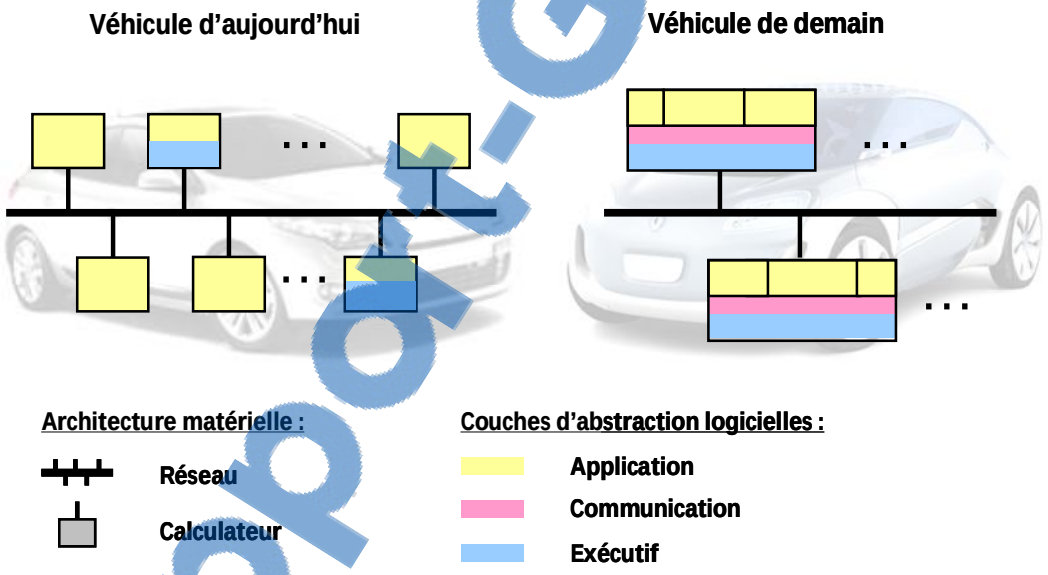


Figure 1 : Evolution des systèmes embarqués automobiles

Plus précisément, l'organisation du logiciel embarqué automobile évolue vers une architecture plus rigoureuse et standardisée. Outre la gestion de la complexité, les avantages recherchés d'une telle approche sont nombreux, en particulier : réutiliser du logiciel à porter sur une nouvelle plateforme matérielle, faciliter l'interopérabilité de l'intégration de composants logiciels sur étagère (*Components-Off-The-Shelf COTS*) économiquement avantageux, et tenir des délais de développement courts (quelques

mois). Le standard automobile émergent AUTOSAR met en œuvre ces idées, avec une architecture logicielle modulaire, stratifiée en différentes couches d'abstraction.

Toutefois, bien qu'il apporte des propriétés intéressantes sur le plan de la flexibilité des systèmes, ce type d'architecture peut aussi être source de nouveaux problèmes, en termes de robustesse par exemple. En effet, l'intégration de composants logiciels sur étagère de niveaux de criticité et de robustesse différents dans le même calculateur peut s'avérer dangereuse. Il s'agit de garantir, pour chaque fonction, les exigences de sécurité (-innocuité) qui lui sont associées indépendamment des autres fonctions, en dépit des phénomènes de propagation d'erreurs.

Le logiciel embarqué dans un calculateur fait partie du système mécatronique du véhicule. Ainsi, les fautes matérielles liées à l'électronique et l'environnement (perturbations électromagnétiques, variations de température, etc.) sont des sources d'erreurs pouvant provoquer la défaillance du logiciel. Elles peuvent se traduire, par exemple, par la corruption de données, de paramètres, voire même de segments de code. Par ailleurs, la complexité du logiciel est un facteur d'augmentation du nombre de fautes logicielles résiduelles. Ces fautes sont susceptibles d'apparaître tout au long du processus de développement du logiciel : au moment des spécifications, des évolutions de conception, de l'implémentation manuelle (erreurs d'interprétation possibles) ou automatique (si l'outil de génération de code n'est pas certifié), de l'intégration de modules de fournisseurs différents, etc.

Dans l'automobile, la notion de sûreté de fonctionnement est principalement caractérisée par la propriété de disponibilité, c'est-à-dire l'aptitude à l'emploi du véhicule. Elle concerne aussi d'autres aspects tels que la fiabilité (aptitude à assurer la continuité de service), la maintenabilité (aptitude au maintien en conditions de fonctionnement ou à la réparation en conditions d'après-vente définies) et la sécurité-innocuité (aptitude du système à ne pas connaître d'évènement catastrophique). Dans ce contexte, notre travail vise à améliorer la robustesse du logiciel ce qui contribue à améliorer la sûreté de fonctionnement du système. Nous définissons la robustesse comme l'aptitude à fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes.

Nous proposons une architecture tolérante aux fautes, pour des plateformes logicielles de type modulaire et multicouches (application/communication/exécutif), ainsi qu'une méthodologie pour la mettre en œuvre. D'un point de vue théorique, l'approche suivie repose sur le principe de « réflexivité » : il s'agit de rendre le système capable de s'auto-surveiller et de s'auto-corriger, en le dotant d'une connaissance sélective de lui-même. En pratique, notre approche consiste à concevoir un « logiciel de défense » externe au système fonctionnel qui est chargé de contrôler la détection et le recouvrement d'erreurs. Sa connaissance du système correspond à des propriétés de sûreté, implémentées par des assertions exécutables. L'instrumentation, qui permet au « logiciel de défense » de récupérer des informations du système et de le corriger, est choisie minutieusement. En effet, la gestion de la tolérance aux fautes ne doit pas détériorer les performances temps réel exigées du système (sélection de l'instrumentation strictement nécessaire) et le contexte économique automobile impose d'optimiser les coûts de développement logiciel (utilisation d'interfaces préexistantes du système si possible).

Ce mémoire présente notre approche suivant 6 chapitres. Le Chapitre I introduit le contexte industriel et la problématique détaillée autour de la robustesse logicielle. Il définit en particulier les types de fautes considérées et les particularités du monde automobile, apportées par les standards émergents AUTOSAR (architecture logicielle) et ISO26262 (sécurité fonctionnelle).

Le Chapitre II décrit d'abord un état de l'art des techniques de tolérance aux fautes. Il introduit ensuite le principe de la solution et de la méthodologie proposée, compatibles au standard ISO26262. L'architecture de tolérance aux fautes repose donc sur l'ajout d'un composant logiciel, appelé « logiciel de défense », qui est principalement conçu à partir de la vérification d'assertions exécutables. La méthodologie se divise en quatre phases classiques : analyse des propriétés de sûreté du système, puis conception, implémentation et validation du logiciel de défense.

Le Chapitre III montre comment obtenir des assertions exécutables à partir des exigences industrielles de sûreté. Ces dernières sont d'abord décomposées selon le type de défaillances qu'elles cherchent à éviter. Une classification de référence des types de défaillances de niveau applicatif est proposée suivant leur impact sur les flots critiques de données ou de contrôle d'une application. Ensuite, les informations d'implémentation liées aux défaillances considérées permettent d'exprimer les propriétés de sûreté sous forme d'assertion exécutables. Une matrice de traçabilité des exigences industrielles de sûreté est introduite pour suivre l'évolution du développement du logiciel de défense à travers les différentes étapes de la méthodologie.

Le Chapitre IV décrit les aspects génériques de l'architecture du logiciel de défense. Il propose des stratégies génériques de détection et de recouvrement d'erreurs, pour chaque type de défaillance de référence, tel un guide de conception du logiciel de défense. La stratégie de stockage d'informations est également précisée. La fiabilité des traces d'exécution est fondamentale, car elles servent de référence pour estimer l'état du système, et dans certains cas pour effectuer du recouvrement par reprise. Enfin, le chapitre termine par une description de l'instrumentation logicielle nécessaire pour intercepter les flots critiques de contrôle et de données à l'exécution, pour récupérer les informations et pour modifier le comportement de l'application défaillante.

Le Chapitre V présente une étude de cas, avec plusieurs applications automobiles, compatible au standard AUTOSAR, cohabitant dans un calculateur unique. Ces applications sont caractérisées par des propriétés de sûreté de types différents, pour illustrer la diversité et la flexibilité des mécanismes de tolérance aux fautes. Le développement du logiciel de défense est mis en œuvre depuis la phase d'analyse jusqu'à l'implémentation.

Le Chapitre VI aborde la validation expérimentale du « logiciel de défense ». Le principe de notre technique d'injection de fautes pour évaluer l'efficacité des mécanismes de tolérance aux fautes est présenté. Plusieurs chantiers en construction sur les outils d'injection de fautes, les possibilités d'améliorations de la technique de vérification proposée, et la mesure de robustesse de la cible protégée sont également discutés.

Chapitre I

Problématique

Problématique

L'objectif général de l'étude est l'amélioration de la robustesse et la fiabilité du logiciel embarqué, dans le contexte et les contraintes spécifiques au domaine automobile. Pour limiter ce domaine de recherche relativement vaste, le chapitre présent pose arbitrairement la problématique détaillée du travail.

La spécification du besoin industriel est une conjecture des exigences sur les systèmes embarqués à venir dans un futur proche. En particulier, nous imaginons vraisemblable le succès des standards émergents d'architecture logicielle [AUTOSAR] et de sécurité fonctionnelle [ISO26262].

La description des caractéristiques de l'environnement d'étude permet ensuite de définir notre problématique, en identifiant les contraintes méthodologiques, les exigences techniques générales, et les autres propriétés spécifiques aux applications automobiles.

I.1 Du système mécatronique au logiciel embarqué

Un système mécatronique automobile est composé de constituants de natures technologiques différentes en interaction entre eux et avec l'environnement extérieur :

- Le système de contrôle électronique, également appelé architecture Electrique/Electronique (cf. Fig.2) ;
- Les interfaces homme-machine (volant, pédalier, commandes au volant, boutons du tableau de bord, badges, etc.) ;
- Les composants mécaniques, hydrauliques, électriques (câblage) et les matières chimiques présentes dans le système (fluides, gaz, etc.).

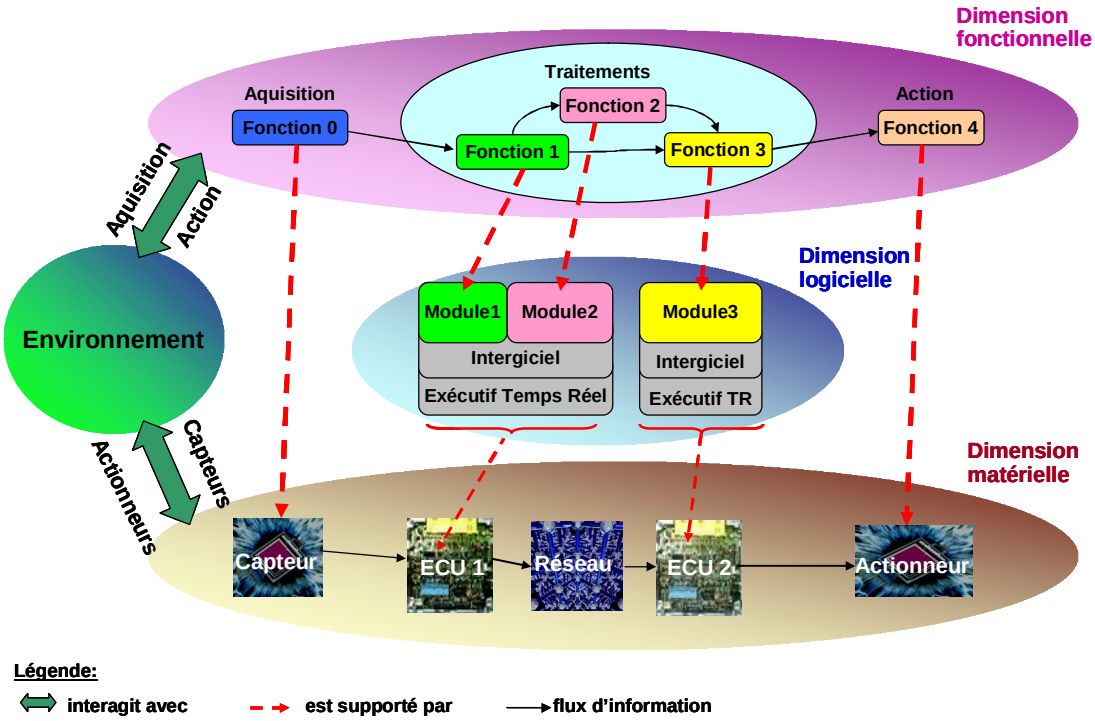


Figure 2 : Un système Electrique/Electronique

Un système électrique/électronique peut être décrit suivant trois angles : fonctionnel, logiciel et matériel (cf. Fig.2). Le système réalise des fonctions. Celles-ci échangent entre elles des flux d'information, qui peuvent représenter des grandeurs physiques à capturer, des signaux à produire pour piloter les actionneurs, ou encore des messages à communiquer entre les calculateurs via un réseau ou une liaison filaire. Les fonctions d'un système peuvent être, par exemple, l'acquisition des requêtes issues des utilisateurs et des autres systèmes ou l'état de l'environnement, le traitement de ces informations afin de calculer les commandes des actionneurs, la mémorisation de ces informations, l'envoi de ces informations vers les utilisateurs ou les IHM via des systèmes de communication. Certaines fonctions (acquisition, action) sont directement supportées par du matériel (capteurs, actionneurs).

Les fonctions de traitement de l'information (calcul, loi de commande, etc.) sont généralement implémentées par du logiciel, de plus en plus organisé en couches d'abstraction (couche applicative, intergiciel, exécutif temps réel).

Au niveau matériel, une fonction peut être répartie sur un ou plusieurs calculateurs (*Electronic Control Unit ECU*). Dans la suite du travail, nous considérons le logiciel embarqué dans un calculateur unique, qui communique éventuellement avec d'autres calculateurs.

Les problématiques de conception classiques de l'embarqué temps réel s'appliquent à l'automobile. Les systèmes doivent être déterministes et sont soumis aux contraintes temporelles de l'environnement physique. En conséquence, les temps de réponse et l'utilisation de la mémoire des calculateurs doivent être bornés statiquement. Pour gérer en même temps plusieurs fonctions véhicules (climatisation, airbag, tableau de bord, etc.), la concurrence d'exécution est un moyen naturel de composer les systèmes.

L'environnement physique peut désigner tantôt un phénomène physique continu (e.g. température, vitesse), tantôt un évènement discret : appui bouton, seuil atteint, etc. Le contrôle de ces informations peut alors être continu, échantillonné, à évènements discrets ou encore hybride, selon la fonction véhicule considérée. Pour concevoir de tels systèmes informatiques de contrôle, le modèle logiciel classique de Turing est trop expressif et complexe. Depuis les années 80, l'introduction de langages de programmation (Lustre, Esterel) spécifiques aux systèmes embarqués temps réel, a particulièrement favorisé le développement d'une culture de programmation graphique, de spécifications exécutables, c'est-à-dire de conception des systèmes à base de modèles ([Simulink] [Stateflow] pour l'automobile, Scade pour l'aéronautique), et enfin de génération automatique de code (TargetLink pour l'automobile, Scade pour l'aéronautique).

Les politiques de développement du logiciel dépendent du constructeur automobile ou du projet véhicule en question. Le constructeur peut choisir de concevoir et développer entièrement ses propres logiciels, pour en avoir la maîtrise complète. Au contraire, dans un modèle de sous-traitance important, le constructeur confie la réalisation du matériel et du logiciel de la plupart des calculateurs à des équipementiers. L'émergence du standard d'architecture logiciel AUTOSAR favorise en particulier la sous-traitance de la production et de l'intégration de systèmes à logiciel prépondérant.

I.2 Modèle de fautes

Il est nécessaire de préciser ce que nous appelons « **modèle de fautes** ». C'est pour nous la liste de fautes (c'est à dire source ou origine des erreurs vis à vis du logiciel embarqué), que nous prenons en compte pour nos travaux. Ce modèle justifie de manière qualitative la nécessité de mettre en œuvre une solution de tolérance aux fautes.

Le logiciel embarqué est indissociable de son contexte matériel et environnemental. Ainsi, un système embarqué automobile est susceptible de tomber en panne à l'exécution à cause de **fautes physiques** ou de **fautes logicielles** (bogues) résiduelles.

Les fautes physiques sont toujours une menace à cause de l'environnement agressif des applications automobiles (température, CEM, pièces mécaniques, humidité, etc.) et la nature même des composants électroniques (vieillissement des composants, défaut du matériel à la production, etc.). Le logiciel peut être altéré lorsqu'un composant électronique est intrinsèquement défectueux ou qu'il subit les agressions de l'environnement. Par ailleurs, le logiciel, au niveau de ses entrées/sorties, peut aussi mal percevoir ou agir sur l'environnement.

En dehors des fautes physiques, le logiciel embarqué peut contenir des bogues résiduels, qui ont pu apparaître pendant les phases de conception, de codage ou d'intégration du cycle de développement logiciel. Pour chercher les origines des fautes purement logicielles, il s'agit de considérer le processus, et examiner pour chacune des phases de développement, les sources d'erreurs logicielles.

La persistance des fautes peut être permanente ou temporaire. Une faute est permanente si sa présence n'est pas liée à des conditions ponctuelles internes (processus de traitement) ou externes (environnement). Une faute est temporaire si elle est liée à de telles conditions ponctuelles, et qui sont donc présentes pour une durée limitée.

La probabilité d'occurrence des fautes est un facteur important pour décider des mesures de protection à prendre (mécanismes d'élimination ou de tolérance aux fautes). Des métriques existent pour les fautes matérielles [ISO26262-5]. Elles sont cependant spécifiques à des systèmes particuliers, propres à un constructeur, et présente donc souvent un caractère confidentiel. La quantification des fautes logicielles [Chillarege *&al.* 1992] n'est pas répandue dans l'automobile. Ne disposant pas de toutes les informations statistiques, nous adoptons donc une démarche pessimiste, où nous tenons compte de l'ensemble des sources d'erreurs potentielles, négligeant dans un premier temps la notion de probabilité d'occurrence.

I.2.1 Fautes physiques

De nombreux modèles de fautes physiques existent chez les constructeurs et équipementiers automobiles. Nous retenons en particulier ceux des standards AUTOSAR, ISO26262 et du projet EASIS, qui considèrent les composants matériels d'un système embarqué de manière générique. Les modèles de fautes AUTOSAR et ISO26262 ne sont pas encore mis à disposition du public, tandis que celui d'EASIS est diffusé [EASIS FT]. *Electronic Architecture and System engineering for Integrated Safety systems* [EASIS] est un projet de recherche européen (2007) réunissant plusieurs industriels du monde automobiles. L'objectif était de développer les technologies électroniques et logicielles nécessaires à l'implémentation de fonctions embarquées sécuritaires. Pour EASIS, les types de fautes sont organisés selon les sous-systèmes matériels qu'ils impactent : capteurs, actionneurs, réseaux, alimentation, et microcontrôleur. Les sous-sections suivantes présentent une vue simplifiée de la classification des types de fautes EASIS.

Les capteurs. Les circuits de conversion des signaux transforment les signaux du capteur en signaux sur lesquels le CPU peut travailler. Un capteur est défaillant dans les cas suivants : le capteur ne délivre aucune valeur ou signal ; la valeur lue du capteur est fausse ; ou encore le capteur ne délivre pas l'information à temps.

Les actionneurs. Des pilotes de puissance amplifient les signaux du CPU pour commander les actionneurs. Un actionneur est défaillant dans les cas suivant : l'actionneur ne répond pas ; l'actionneur fonctionne de manière permanente sans commande de contrôle ; l'actionneur ne fonctionne pas au bon moment ; l'actionneur ne fonctionne pas avec les performances attendues.

Les réseaux. Au niveau d'un nœud du réseau, les fautes considérées sont : les valeurs d'un message reçu sont erronées ; le message est reçu en retard ou trop tôt ; le message ne peut pas être envoyé dans la plage temporelle impartie. Dans un environnement distribué, les fautes potentielles sont : tous les récepteurs du message détectent le même type de fautes ; tous les récepteurs du message détectent des types de fautes différents ; certains récepteurs du message récupèrent un bon message, tandis que les autres détectent le même type de fautes ; certains récepteurs du message récupèrent un bon message, tandis que les autres détectent des types de fautes différents.

L'alimentation. Les alimentations externe et interne sont décrites séparément en raison de la différence importante entre l'alimentation du véhicule et d'un calculateur. Une faute sur l'alimentation externe, fournie par la batterie et le générateur du véhicule, peut causer ou être provoquée par une faute sur l'alimentation interne, qui délivre de la puissance à tous les circuits et organise la distribution de puissance à l'intérieur du calculateur. L'alimentation interne est exposée aux surtensions, aux sous-tensions, aux courts-circuits, aux surintensités dues à des actionneurs ou composants actionnés à tort ou défectueux, aux fuites de courant trop élevé, aux baisses de tension, aux problèmes temporels de démarrage ou d'arrêt. L'alimentation externe est exposée aux surtensions (due à aux cycles de charge/décharge, à une panne du générateur, etc.), aux sous-tensions (due à une batterie trop faible, une surintensité, etc.), aux courts-circuits, et aux limitations de courant.

Le microcontrôleur. Le microcontrôleur représente l'intelligence centrale du calculateur. Il contient généralement un cœur, de la mémoire (RAM, ROM, EEPROM) et des interfaces d'entrées/sorties. Le cœur est exposé à des fautes de calculs, des fautes de flot de programme, des fautes des sources d'interruption, entraînant des erreurs de séquence, fréquence, délai d'exécution etc. Les mémoires peuvent être aussi bien internes, qu'externes au CPU. Par exemple, les données (valeurs de calibration, code) qui sont utilisées pour l'exécution du programme peuvent être localisées dans des mémoires externes flash, qui sont sur des puces physiquement séparées du microcontrôleur. Dans tous les cas, les mémoires sont exposées à des défauts de cellules, des accès à la mémoire corrompus, un débordement de la mémoire. Enfin des problèmes peuvent survenir aux interfaces à cause d'erreurs de conversion analogique/digitale d'entrées/sorties.

I.2.2 Fautes logicielles

L'approche automobile des fautes logicielles est encore relativement peu approfondie, ce qui peut s'expliquer par le développement récent de l'informatique embarquée. Chez Renault, par exemple, les analyses de défaillances pour les calculateurs sont guidées par un modèle de fautes physiques. L'équivalent au niveau du logiciel n'existe pas encore.

Dans le projet européen EASIS, les fautes logicielles se mélangent aux fautes matérielles [EASIS FT]. Ainsi, parmi les types de fautes impactant les capteurs, actionneurs, microcontrôleur, etc., des fautes logicielles sont décrites : les fautes d'ordonnancement, les fautes de communication entre composants logiciels, et les fautes fonctionnelles.

Dans le consortium AUTOSAR, chaque groupe de travail exprime plus ou moins ses hypothèses de fautes, pour le module logiciel dont il est en charge. Ainsi, trois points de vue principaux coexistent :

1. Les erreurs considérées au **niveau applicatif** sont les impacts sur : les valeurs dans le système, les temps d'exécution, les séquences ou les ordres d'exécution, les accès aux ressources du système tels que la mémoire. Le traitement des erreurs mis en œuvre suit le principe du FDIR (*Fault Detection, Isolation and Recovery)* du domaine spatial [Salkham 2005].

2. Concernant le **logiciel de base**, le groupe de travail « *Error Handling*» a proposé une harmonisation des notifications et traitement d'erreurs de chaque module du logiciel de base. Les fautes sont considérées au niveau de granularité du composant logiciel, dans la mesure ou chaque module est susceptible de détecter et notifier, ou encore produire des erreurs. Concrètement, une table de traçabilité a été mise en place, répertoriant toutes les interfaces standardisées concernant des erreurs. Ceci permet de vérifier la cohérence syntaxique, puis fonctionnelle des chaînes de traitement d'erreurs.

3. Deux types d'erreurs sont distingués selon le moment d'occurrence dans le **cycle de vie du produit**. Les **erreurs de « production »** peuvent être des défauts matériels ou des exceptions logicielles inévitables, impactant le code produit. Les **erreurs de « développement »** sont principalement des bogues qui apparaissent au cours du développement du produit et qui doivent être éliminés avant la mise en service du produit.

En fonction des interlocuteurs, le concept de « faute logicielle » est généralement interprété différemment. Le constructeur automobile s'intéresse principalement aux fautes de niveau applicatif et fonctionnel (1), qu'il peut traduire en comportement indésirable du véhicule. Les fautes, auxquelles un équipementier fait face, appartiennent davantage au niveau modulaire du logiciel de base (2). Dans le monde académique ou dans d'autres secteurs, aéronautiques, ferroviaires, etc., les fautes logicielles sont plutôt liées au processus de développement (conception, implémentation, intégration) du produit (3).

Notre premier travail a donc consisté à regrouper toutes ces informations, dans un tableau, organisant les types de fautes logicielles, en trois volets : niveau applicatif, niveau modulaire (logiciel de base), niveau processus. Pour cela, nous avons bénéficié du soutien et des retours d'expériences des partenaires du projet collaboratif SCARLET (*Systèmes Critiques pour l´Automobile : Robustesse des Logiciels Embarqués Temps-réels*), piloté par Renault dans le cadre du pôle de compétitivité SYSTEM@TIC.

Cette synthèse est importante, dans la mesure où elle présente les faiblesses potentielles du logiciel embarqué automobile, de manière générale. Elle justifie alors qualitativement l'intérêt de tolérer ces fautes, pour améliorer la fiabilité logicielle.

Les tables 1, 2, 3 et 4 représentent la classification des types de fautes, telle que nous l'avons proposée dans le projet SCARLET, en exposant seulement la partie purement logicielle. Les catégories de fautes sont différenciées selon l'endroit et/ou moment d'occurrence, que nous appelons « cible ». Au niveau applicatif (Tab.1) et modulaire (Tab.2 et Tab.3), les cibles sont des parties logicielles. Au niveau du processus (Tab.4), les cibles concernent le processus de développement. Une relation de cause à effet aurait pu être explicitement tracée entre ces 4 tableaux. Nous avons simplement signalé globalement la chaîne de propagation d'erreur, en intitulant « fautes » les évènements indésirables au niveau processus (Tab.4), « erreurs » les perturbations au niveau des modules du logiciel de base (Tab.2 et Tab.3), et « défaillances » les impacts de niveau applicatif (Tab.1). En effet, les fautes logicielles sont potentiellement à l'origine d'erreurs dans le logiciel, qui mènent aux défaillances des applications.

LIST OF EMBEDDED SOFTWARE FAILURE TYPES			
Software Target			**FAILURE**
Control flow	**Software component, Applicative function**	Transition	non expected
			invalid/ forbidden
		Functional sequence	non expected
			invalid/ forbidden
		Time of execution	too short
			too long
Data flow	**Variable, Signal, Parameter**	Value at runtime	invalid/ out of range / forbidden
			wrong value (within range)
		Time of exchange	too late
			too early

Table 1 : Modèle de fautes SCARLET – niveau application

Les défaillances applicatives distinguent globalement les flots de contrôle et de données. Cette décomposition est un pilier majeur de notre travail, nous l'aborderons plus en détail en section III.2.

Les erreurs modulaires dépendent de la présence des modules du logiciel de base, dans la plateforme considérée. Les tables 3 et 4 ne sont donc pas exhaustives. Elles regroupent les modules qui ont été analysés par le groupe de travail « *error handling* » d'AUTOSAR, à savoir la gestion de la mémoire et de la communication. Les autres composants du logiciel de base peuvent être présentés de la même manière. Nous avons donc ajouté les erreurs du chien de garde logiciel. Ce niveau de description des erreurs montre la responsabilité de chaque service du logiciel de base, vis-à-vis du reste de l'architecture. Si les relations entre ce niveau modulaire et le niveau applicatif peuvent être explicité, cela permet de diagnostiquer lors d'une panne de l'application, quelles sont les responsables potentiels, et comment réagir.

LIST OF BASIC SOFTWARE ERROR TYPES (1)			
Software Target			**ERROR**
Communication Management	**Communication Channel**	CAN	CAN High Speed: Bus Off on a CAN network
			Cannot transmit a new message because the transmission queue is full
		RTE	An internal RTE limit has been exceeded. Request could not be handled. OUT buffers are not modified
	Signals	CAN	Time out while sending a command to the CAN controller
			Frame accepted by the CAN HW, with an unexpected CAN Identifier
			Error while writing in the message box of the can controller
		CANTP	Time out during the transmission of a message with the CAN TP protocol
			Timeout during the reception of a CAN TP message
		CANIF	CAN frame received with an unexpected Data Length Counter, and DLC check enabled
			Request for the transmission of a PDU, with an invalid ID
		COM	An expected message was not received in time by COM
			Timeout while waiting for the confirmation of a transmission
			The PDU counter differes from the expected counter
			A replicated PDU is not identical in all copies
		RTE	Returned by AUTOSAR Services to indicate a generic application error
			An IPDU group was disabled while the application was waiting for the transmission acknowledgment. No value is available. *It is generally not a fault, since the IPDU group is switched off on purpose*
			Timeout in a client/server operation.
			A blocking API call returned due to expiry of a local timeout rather than the intended result. OUT buffers are not modified. The interpretation of this being an error depends on the application
			An explicit read API call returned no data. *(This is generally not an error.)*
			An API call for reading received data of isQueued = true indicates that some incoming data has been lost due to an overflow of the receive queue or due to an error of the underlying communication stack
			An API call for reading received data of isQueued = false indicates that the available data has exceeded the aliveTimeout limit. A COM signal outdated callback will result in this error

Table 2 : Modèle de fautes SCARLET – niveau modulaire (1)

LIST OF BASIC SOFTWARE ERROR TYPES (1)			
Software Target			**ERROR**
Memory Management	Driver	FLS	The flash write job failed due to a hardware error
			The flash erase job failed due to a hardware error
			The flash read job failed due to a hardware error
			The flash compare job failed due to a hardware error
			Expected hardware ID not matched during initialization of the driver
		EEP	The EEPROM write job failed due to a hardware error
			The EEPROM erase job failed due to a hardware error
			The EEPROM read job failed due to a hardware error
			The EEPROM compare job failed due to a hardware error
	Flash Emulation (FEE)		The Flash Eeprom Emulation detects a problem of consistency in the block to read
	EEPROM Abstraction (EA)		The Eeprom Abstraction emulation detects a problem of consistency in the block to read
	Gestion de NVRAM (NVM)		The CRC check on the RAM block failed
			Job Failure : The driver level reports job failed
			The NVRAM Block written to NVRAM is immediately read back and compared with the original content in RAM
			Static Block ID Check failed
			Redundant block invalid during reading or writing is reported to the NVRAM User as a loss of redundancy
Watchdog	Watchdog Manager (WDGM)		API service used in wrong context (without module initialization)
			API service called with a null pointer parameter
			API service called with wrong "mode" parameter
			API service called with wrong "supervised entity identifier" parameter
			Disabling of watchdog not allowed (e.g in safety relevant systems)
			Desactivation of alive-supervision for a supervised entity not allowed
			Alive-supervision has failed and a watchdog reset will occur
			Watchdog drivers' mode switch has failed
	Watchdog Driver (WDGIF)		service API appelé avec mauvais paramètre indexation du matériel

Table 3 : Modèle de fautes SCARLET – niveau modulaire (2)

En termes de propagation d'erreurs, les tables 3 et 4 ne suffisent cependant pas à expliquer les défaillances applicatives. Ces tables ne définissent que les notifications d'erreurs, lorsque les services du logiciel de base ne sont pas corrects. De la même manière qu'une valeur de donnée peut être fausse à l'intérieur de sa plage de valeurs autorisée, un service de l'infrastructure logicielle peut être non voulu, sans pour autant être erroné ou interdit. Par exemple, un problème classique d'exclusion mutuelle peut toucher l'utilisation d'une variable globale critique non protégée, à cause d'un défaut

de conception logicielle, tandis que les services d'écriture et de lecture de données sont opérationnels. Dans la suite de notre travail, l'analyse des modes de défaillances d'un système devra donc être plus approfondie que le modèle de fautes SCARLET, pour mieux maîtriser la propagation d'erreurs logicielle.

LIST OF SOFTWARE FAULT TYPES		
Process Target (except operation)		**FAULTS**
Design	Rules for Design	non respected rules (ex : MISRA) non identified/considered rules
	Design specification / model	bad temporal design (sizing, execution order…) bad resource sizing bad data usage (wrong choice of data for usage, wrong handling of a data…) non expected modes
Development	Manual Coding	misinterpretation of specifications coding errors compiler/linker's default
	Automated Code Generation	generator's default bad scaling bad adaptation to generator's constraints
	Integration	misinterpretation of specifications coding errors (glue code) bad module versions/configuration compiler/linker's defaut bad scaling (global data)
Validation	Fault model	non identified/considered faults implicit / not identified
	Testing	low coverage of fault model

Table 4 : Modèle de fautes SCARLET – niveau processus

Enfin, l'origine des erreurs ou des défaillances à l'exécution peut provenir de mauvaises pratiques au cours du processus de développement du logiciel. Par exemple, les fautes logicielles survenant pendant la phase de **conception** peuvent s'expliquer par le non respect des règles de conception ([MISRA 1998]), par un mauvais dimensionnement en termes de ressources, de durée ou d'ordre d'exécution, par une mauvaise sélection des données à traiter, ou encore par des modes d'exécution non prévus. Les bogues pendant la phase d'**implémentation** peuvent apparaître suite au codage manuel, à cause d'une mauvaise interprétation des spécifications, ou d'une erreur de codage, ou de défauts de la compilation ou de l'édition de liens. Si la génération automatique de code est employée, la configuration des outils peut introduire des erreurs. Ce phénomène est d'autant plus important que le logiciel est complexe. L'adaptation de la conception aux contraintes du générateur peut être incomplète (ex : la multiplication de valeurs booléennes n'est pas optimale et n'est pas supportée par tous les générateurs). Les défauts du générateur, en particulier s'il n'est pas certifié, peuvent entraîner des erreurs dans le code généré. La phase d'**intégration** prend une place importante dans le contexte des systèmes modulaires et de l'utilisation de composants sur étagère. Au niveau de l'intégrateur, de la même manière que pour le codage manuel, il peut se produire des erreurs d'interprétation de spécifications, de codage du code glue (application/logiciel de base), ou encore un

mauvais dimensionnement des variables globales, une utilisation de la mauvaise version ou configuration de module, ou enfin un défaut du compilateur ou éditeur de liens. La phase de **vérification et validation** du logiciel n'introduit pas d'erreurs *a priori*, bien que son efficacité conditionne le taux de bogues résiduels du logiciel.

I.3 Standards automobiles émergents

Comme nous allons le voir dans cette section, le choix des deux standards automobiles en cours de construction AUTOSAR et ISO26262, comme référence pour les années à venir, est justifié par rapport aux tendances technologiques de ces dix dernières années dans les domaines de l'informatique embarquée, du transport et de la sûreté de fonctionnement.

I.3.1 Standard d'architecture logicielle

Aujourd'hui, la plupart des fournisseurs du domaine automobile ont déjà adopté une architecture logicielle **structurée en couches**. L'avantage, pour un fournisseur, est de pouvoir réutiliser les couches de support d'exécution pour différents clients, en localisant principalement les modifications dans la couche contenant les applications. Ce type d'architecture répond aussi à la problématique de portabilité, pour permettre d'adapter les ressources aux besoins fonctionnels de chaque application, ou aux éventuelles variations du besoin au cours d'un projet. Elle permet de changer le matériel sans avoir à modifier la couche applicative.

L'approche **modulaire** pour le logiciel consiste à construire le logiciel à partir de l'assemblage de briques logicielles réutilisables : les composants. Cette approche n'est pas encore généralisée aujourd'hui dans le monde automobile. En effet, la culture logicielle est fortement caractérisée par les solutions propriétaires. La modularité reste cependant une tendance de l'évolution technologique. Par exemple, un projet Renault EMS2010, démarré en 2004, utilise ce concept dans le but de proposer une architecture logicielle standardisée pour le contrôle moteur. Les diverses fonctions du contrôle moteur se déclinent ainsi en sous-fonctions puis en modules. Le module est la plus petite entité que Renault souhaite pouvoir facilement transporter d'une plate-forme à une autre. C'est donc au niveau du module que sont décrites les spécifications et les règles de codage. Les modules applicatifs sont indépendants du matériel et des fournisseurs. Par ailleurs les modules doivent être facilement réutilisables et adaptables dans une bibliothèque de modules.

Le consortium **AUTomotive Open System ARchitecture (AUTOSAR)** a pour vocation de standardiser une architecture modulaire et multicouche, pour le logiciel embarqué automobile. Créé en 2003, AUTOSAR est un partenariat entre acteurs majeurs, constructeurs et équipementiers, du domaine automobile. Son objectif est de proposer une infrastructure commune à l'ensemble des domaines moteur, châssis, carrosserie, multimédia. L'idée principale est de favoriser l'interopérabilité des composants logiciels.

L'attractivité de ce standard est forte par la possibilité de réduire les coûts de développement logiciel à travers :

- La facilité de réutiliser le logiciel applicatif ;
- La portabilité des fonctions d'une plate-forme matérielle à une autre ;
- La réduction d'efforts de maintenance et la facilité de gestion de l'obsolescence du logiciel ;
- La possibilité d'utiliser des composants commerciaux COTS ;
- La simplification de l'architecture matérielle par le multiplexage des fonctions sur des calculateurs génériques.

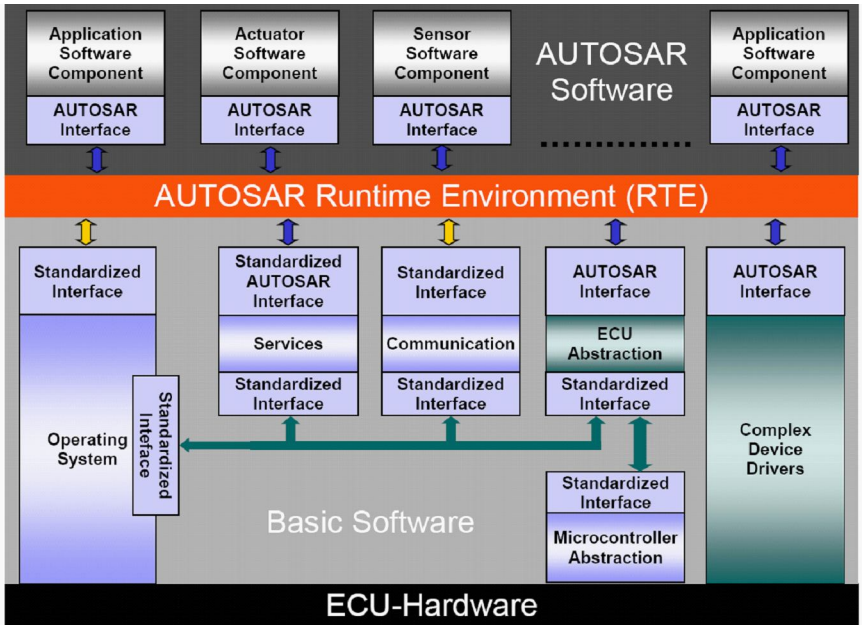


Figure 3 : Architecture logicielle modulaire multicouche AUTOSAR

L'architecture logicielle AUTOSAR est organisée en trois couches d'abstraction logicielles principales (cf. Fig.3) : couche applicative, intergiciel (*RTE Runtime Environment*), couches basses ou logiciel de base (*BSW Basic Software*). AUTOSAR se distingue des autres architectures par son intergiciel. Il suit un mouvement récent de recherche et développement sur les intergiciels orientés objet dans l'automobile (projet EAST-EEA 2004) et à plus grande ampleur dans le monde académique (CORBA, TAO, etc.). A l'image de l'ORB (Object Request Broker) du monde CORBA, " bus logiciel " par lequel les objets envoient et reçoivent des requêtes et des réponses de manière transparente, AUTOSAR introduit la notion de " bus virtuel " *(Virtual Fonctional Bus VFB*) (cf. Fig.4). Le rôle du *VFB* est de permettre la conception des fonctions véhicule, sans se préoccuper du support matériel, ou du partage de fonctionnalités entre les calculateurs. Ainsi, le VFB réalise l'abstraction de toutes les interconnexions entre les composants logiciels applicatifs dans le véhicule. La figure 4 montre des composants logiciels applicatifs comme des boîtes reliées entre elles dans le VFB. Les graphismes différents des liens représentent des types de communication différents (client-serveur, émetteur-récepteur). Une fois le code embarqué dans les calculateurs concernés, le *RTE Runtime Environment*, qui est

l'implémentation du *VFB* (qui n'est qu'un concept), fait l'interface entre le logiciel applicatif et le logiciel de base. Les applications peuvent ainsi utiliser de manière transparente les services des couches basses pour des fonctionnalités transversales (NVRAM, diagnostic, I/O,…).

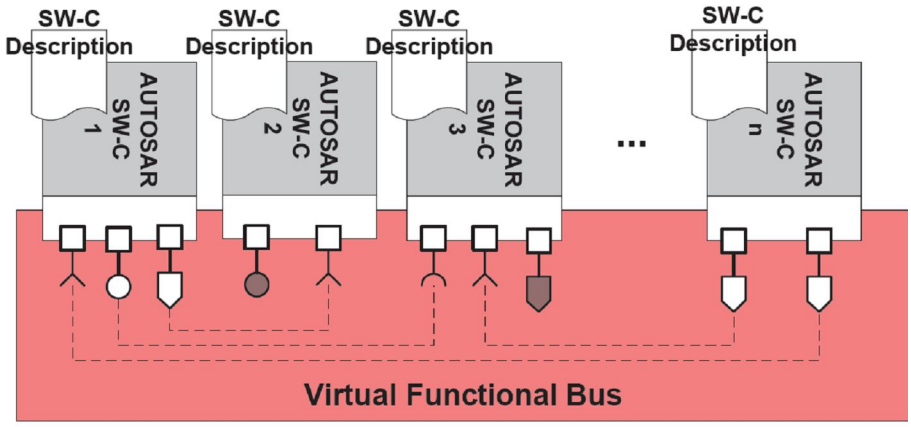


Figure 4: Virtual Functional Bus (VFB)

I.3.2 Standards pour la sûreté de fonctionnement logicielle automobile

Les principaux guides de sûreté de fonctionnement pour le logiciel automobile sont l'IEC 61508, qui est la norme de sécurité fonctionnelle appliquée actuellement, en attendant la norme ISO 26262 et le standard MISRA, qui est parfois utilisé lors du développement de logiciel.

- **IEC 61508** est un standard européen générique pour un nombre très large de secteurs d'activités [IEC 1998]. Il s'adresse particulièrement aux systèmes dans lesquels des dysfonctionnements peuvent avoir des conséquences sur des personnes ou sur l'environnement. Il définit un niveau de criticité (Safety Integrity Level SIL) des systèmes. Mais les méthodes sont difficilement applicables à des projets très contraints (délais courts, coûts de mise en œuvre).

- Le standard **MISRA guidelines** (Motor Industry Software Reliability Association), issu d'une collaboration de constructeurs automobiles, donne des règles pour guider le développement logiciel dans les véhicules, de manière à éviter certaines erreurs de conception [MISRA 1998]. Le MISRA a aussi introduit la notion de niveau de criticité qui a été reprise dans le standard IEC61508.

- La norme de sécurité ISO 26262, dédiée aux systèmes Electriques/Electroniques embarqués dans les véhicules routiers, est en cours de création (première version prévue pour 2011-2012). Ce sera la déclinaison automobile de l'IEC 61508. Elle s'inspire des standards de sécurité fonctionnelle d'autres domaines (ferroviaire ou aéronautique), mais vise à proposer un cycle de vie de sécurité en adéquation avec les phases de vie typiques d'un projet automobile.

ISO26262 marque une dépendance forte entre la sûreté de fonctionnement et le processus de développement d'un produit. En effet à chaque étape du processus, des erreurs de conception, de codage, etc., sont susceptibles d'intervenir et de se

transmettre à l'étape suivante et ainsi de nuire au bon fonctionnement du produit final. A chaque étape du processus, des exigences concernant la sûreté de fonctionnement sont donc proposées.

Méthodes		ASIL			
		A	B	C	D
1a	Organisation hiérarchique appropriée des composants logiciels	++	++	++	++
1b	Taille limitée des composants logiciels	++	++	++	++
1c	Taille limitée des interfaces	+	+	+	+
1d	Cohésion forte entre les composants logiciels	+	++	++	++
1e	Couplage réduit entre les composants logiciels	+	++	++	++
1f	Propriétés d'ordonnancement appropriées	++	++	++	++
1g	Utilisation restreinte des interruptions	+	+	+	++

Table 5 : ISO26262 – Principes de conception pour l'architecture logicielle

Les exigences peuvent être accompagnées de tableaux (cf. Tab.1), listant des méthodes ou techniques recommandées pour respecter l'exigence. En fonction de la criticité du produit, évalué en ASIL *(Automotive Safety Integrity Level)*, ces recommandations sont plus ou moins fortes. Les niveaux d'ASIL sont gradués de A à D, D étant le niveau le plus critique. Il y a également trois niveaux de recommandation :

++ : La technique est fortement recommandée pour l'ASIL considéré. Si la technique ou mesure n'est pas utilisée, la raison de ce choix doit être fournie ;

+ : La technique est recommandée pour l'ASIL considéré ;

o : Il n'y a pas de recommandation pour ou contre l'utilisation de la technique.

Par exemple, le tableau 5, extrait de l'ISO26262 partie 6 expose les principes généraux de conception de l'architecture du logiciel, qui doivent être appliqués pour réaliser les propriétés suivantes : « modularité, encapsulation, et complexité faible ».

Les lignes des tableaux de recommandation sont numérotées. La norme distingue deux types d'entrées : les entrées consécutives (1, 2, 3, etc.) et les entrées alternatives (1a, 1b, 1c, etc.). Pour les entrées consécutives, toutes les méthodes sont recommandées suivant le niveau d'ASIL. Si des méthodes autres que celles listées sont appliquées, une justification sera demandée pour être conforme à l'exigence correspondante. Pour les entrées alternatives, une combinaison appropriée des méthodes doit être appliquée suivant le niveau d'ASIL. Si les méthodes sont listées avec des degrés de recommandation différents pour un niveau d'ASIL donné, celle de plus haut degré devrait être préférée (e.g. « Propriétés d'ordonnancement appropriées » dans le tableau 2). La combinaison de méthodes sélectionnées doit être justifiée pour être conforme à l'exigence correspondante. Dans le cas où toutes les méthodes fortement recommandées pour un niveau d'ASIL particulier sont

appliquées, la justification n'est pas nécessaire (e.g. 1a, 1b, 1f, pour l'ASIL A dans le tableau 2).

L'objectif de l'exemple du tableau 5 était d'illustrer le formalisme des niveaux de recommandation de l'ISO26262. Notons tout de même que d'après ce tableau, la différence entre les principes de conception de l'architecture pour les applications de niveau d'ASIL B, C, et D est faible : il y a seulement l'utilisation restreinte des interruptions (ligne 1g) qui est fortement recommandée en ASIL D. Du point de vue du monde automobile, les principes de conception généraux du logiciel sont donc déjà des pratiques courantes, appliquées indifféremment des niveaux d'ASIL. La différence entre les niveaux ASIL se traduira d'avantages dans d'autres exigences et tableaux de méthodes de tolérance ou d'élimination de fautes, étudiés plus tard (cf. section II.1).

I.4 Problématique

L'objectif général de la thèse est d'« **améliorer la sûreté de fonctionnement du logiciel embarqué automobile**». Pour améliorer la sûreté de fonctionnement, plusieurs approches existent : empêcher l'occurrence de fautes (prévention), fournir un service en dépit de fautes (tolérance), et réduire la présence des fautes (élimination). L'axe d'étude choisi arbitrairement est de rendre le **système capable de tolérer des fautes**, c'est-à-dire détecter des erreurs et, dans la mesure du possible, les corriger. En effet, toute technique d'élimination et de prévention est susceptible de laisser des fautes. Il est alors nécessaire de tolérer les fautes résiduelles.

La problématique revient donc à construire une solution de tolérance aux fautes adaptée au contexte industriel décrit précédemment.

Nous appelons « complexe », un **système informatique multitâches géré par un exécutif temps réel**. L'augmentation du nombre de fonctions embarquées dans le véhicule, et le besoin de leur activité simultanée, nécessite une exécution concurrente des applications (e.g. climatisation et tableau de bord). L'utilisation des systèmes d'exploitation temps réel coopératifs ou préemptifs se généralise donc pour ordonnancer les tâches. Nous considérons des implantations sur une **plate-forme mono-processeur**. Le logiciel cible étudié est un **assemblage de composants logiciels**, pouvant provenir de fournisseurs différents. De plus, plusieurs applications sont susceptibles d'être multiplexées sur le même support d'exécution. Les propriétés des divers modules sont donc vraisemblablement **hétérogènes, en termes de robustesse, observabilité, commandabilité, de niveau de criticité, etc**. L'assemblage de composants sur étagère, sur un même support d'exécution, est donc une source de problèmes de robustesse.

Du système de contrôle d'airbag à celui du régulateur de vitesse en passant par la gestion des essuie-glaces, les applications automobiles sont nombreuses et très diverses. Caractérisées par des niveaux de criticité non-homogènes, les applications ne requièrent pas les mêmes mécanismes pour la robustesse. La solution de protection recherchée doit pouvoir **s'adapter à tout type d'application comportant des exigences sécuritaires**. De plus, la **réutilisabilité** de la solution proposée est requise,

pour favoriser la réduction de coûts de développement. Cette notion est associée aux propriétés de **portabilité**, de **maintenabilité** et de **faible complexité**.

Les standards automobiles émergents, **AUTOSAR et ISO26262**, concernant le logiciel, sont considérés comme des références pour nos travaux. Le type d'applications étudiées n'est cependant pas limité à ces standards. Ainsi, des systèmes propriétaires non-AUTOSAR peuvent également être traités. La solution de tolérance aux fautes recherchée cible plus généralement les architectures logicielles modulaires, organisées en couches d'abstraction superposées. La norme ISO26262, quant à elle, représente un guide d'exigences et de recommandations à suivre.

L'ensemble du modèle de fautes décrit en I.2 est considéré, c'est-à-dire les **fautes provenant du matériel, de l'environnement et du cycle de développement du logiciel** (conception, implémentation, intégration).

Dans l'aéronautique, l'exigence primordiale est la sécurité des passagers, ce qui n'autorise aucun compromis. Dans l'industrie automobile, même si l'objectif est d'optimiser à la fois **Qualité, Coûts et Délais**, les exigences les plus importantes restent le coût des réalisations. Ceci s'explique par un niveau global de criticité des applications automobile qui reste plus faible que dans l'aéronautique, et surtout des enjeux économiques très différents. Selon une étude de 2006 [EmbeddedTouch], le volume moyen annuel produit est autour de 2000 avions contre 64 millions de voitures, avec un prix à l'unité estimé à 100 millions d'euros pour un avion contre 25 000 euros pour un véhicule. Les moyens et ressources mis en œuvre pour la sûreté de fonctionnement dans l'automobile sont donc beaucoup plus faibles que dans l'avionique. Chaque ajout de ressources matérielles ou d'efforts de développement supplémentaires pour la sûreté de fonctionnement doit être optimisé et soigneusement justifié. Les solutions classiques de redondance matérielle, d'utilisation d'outils certifiés, etc. ont donc encore peu de succès dans l'automobile. Il s'agit donc de proposer une **solution à moindre coût**, permettant tout de même d'améliorer la fiabilité de la cible considérée. Compte tenu des contraintes économiques très fortes sur le matériel, il est préférable de s'orienter vers une **solution logicielle de tolérance aux fautes**. Une telle approche se marie parfaitement à la tendance actuelle au développement de systèmes à logiciel prépondérant. Le minimum pour obtenir un produit robuste est de **renforcer la phase de vérification et validation**, et démontrer la **conformité aux exigences attendues**.

Enfin, pour que le travail soit réutilisable, il doit suivre une **méthodologie générique rigoureuse** qui puisse s'inscrire dans le processus de développement industriel.

Conclusion du Chapitre I

Le logiciel embarqué automobile est à la fois une cible de fautes physiques et logicielles, car il ne peut être dissocié du contexte mécatronique auquel il appartient.

Par ailleurs, les architectures logicielles modulaires et multicouches préconisées par le standard AUTOSAR, pour leurs nombreux avantages de flexibilité, présentent un risque accru de propagation d'erreurs, liées à l'intégration de composants sur étagère de robustesse hétérogène.

Pour garantir la sécurité fonctionnelle des systèmes électriques/électroniques automobiles, le standard ISO26262 définit des exigences de protection du système à mettre en œuvre. Le degré de recommandations est graduel suivant le niveau de criticité de la fonction considérée.

L'objectif de la thèse est de proposer une solution de tolérance aux fautes, en tenant compte du contexte de ces deux standards.

Par la suite, nous justifions et présentons l'approche choisie, dite « réflexive ». Elle repose sur l'introduction d'un logiciel de défense, conçu au cas par cas selon les propriétés de sûreté des applications et leur niveau de criticité. Ceci permet à la solution proposée d'être compatible avec les exigences du standard ISO26262. Pour démontrer la faisabilité de notre solution, nous avons implémenté des applications à infrastructure AUTOSAR, protégées par notre logiciel de défense.

Chapitre II

Etat de l'art & Méthodologie proposée

Etat de l’art & Méthodologie proposée

L’état de l’art des techniques de tolérance aux fautes académiques et industrielles permet de dégager les principes de base à adapter au contexte automobile. Dans ce chapitre, nous décrivons d’abord le point de vue de la norme ISO26262, qui constitue en lui-même un état de l’art des techniques de tolérance aux fautes reconnu dans le monde automobile. Nous examinons ensuite des architectures organisées des mécanismes de détection et de recouvrement d’erreurs, tous domaines confondus (ferroviaire, avionique, etc.). Nous exposons alors les principes de l’architecture tolérante aux fautes que nous proposons. Elle vise à améliorer la robustesse du logiciel embarqué automobile, par une meilleure maîtrise des informations critiques à l’exécution. Enfin, nous planifions une méthodologie rigoureuse pour développer la solution présentée.

II.1 Techniques recommandées par l'ISO26262

Selon la norme ISO26262, la sûreté de fonctionnement est présente dans les diverses phases du cycle de développement du logiciel, sous forme de techniques d'élimination d'erreur (ex : vérification statique de code, tests de conformité aux exigences). L'étape de conception de l'architecture logicielle a une particularité supplémentaire : elle introduit les mécanismes tolérance aux fautes (détection et recouvrement d'erreur) à intégrer dans le logiciel.

II.1.1 Techniques de détection d'erreur

La norme ISO26262 indique plutôt des grandes familles de mécanismes que des techniques particulières (tables 6 et 7). Les intitulés des méthodes ne sont d'ailleurs pas définis explicitement dans le standard. Cela laisse alors une certaine marge de liberté et d'interprétation. Pour une approche concrète, nous essayons d'illustrer chaque famille de technique de détection recommandée dans le tableau 6, par des exemples de mécanismes. Les commentaires de la norme sur certaines méthodes sont intégrés à nos propres commentaires. L'explication des symboles (numérotation de ligne, degré de recommandation) apparaissant dans les tables ISO26262 a été donnée en détail dans la section I.3.2.

Exigence : Pour spécifier les mécanismes de sécurité logicielle nécessaires au niveau de l'architecture logicielle, sur la base des résultats des analyses de sûreté de fonctionnement exigées précédemment dans la norme, les mécanismes pour la détection d'erreurs listés dans la table [6] et les mécanismes pour la gestion d'erreurs listés dans la table [7] doivent être appliqués.					
Méthodes		**ASIL**			
		A	B	C	D
1a	Contrôle de vraisemblance	++	++	++	++
1b	Détection d'erreurs de données	+	+	+	+
1c	Unité de contrôle externe	o	+	+	++
1d	Gestion du flot de contrôle	o	+	++	++
1e	Conception logicielle diversifiée	o	o	+	++

Table 6 : ISO26262 – Détection d'erreur au niveau de l'architecture logicielle

(1a) Contrôle de vraisemblance. "Le contrôle de vraisemblance contient les vérifications d'assertion. Les contrôle de vraisemblance complexes peuvent être réalisés en utilisant un modèle de référence du comportement désiré". Par exemple, le modèle simplifié d'une loi de commande peut être utilisé pour vérifier que les variables de sorties d'une application appartiennent à une plage de valeur acceptable. La vérification d'assertions ou « programmation défensive » compare certaines relations d'invariance entre les variables du programme. Lorsqu'une erreur est détectée, un traitement d'exception est déclenché pour arrêter la tâche en cours et éviter la propagation d'erreurs. Ces types de mécanismes sont fortement

recommandés pour tous les niveaux d'ASIL. Dans l'automobile, les modules logiciels applicatifs intègrent déjà couramment ces mécanismes de protection.

(1b) Détection d'erreurs de données. "Les types de méthodes qui pourraient être utilisés pour détecter des erreurs de données incluent les codes détecteurs d'erreurs et le stockage de données multiple". Les techniques de codes détecteurs d'erreurs [Feldmeier 1995] utilisent une redondance dans la représentation de l'information, soit en ajoutant des bits de contrôle aux mots, soit par une nouvelle représentation des mots d'information. Le niveau de redondance dépend des hypothèses d'erreurs (simples, multiples, unidirectionnelles ou non). Ces types de mécanismes ont été imaginés et utilisés initialement pour le codage de l'information en vue de sa transmission en environnement bruité par exemple sur un bus CAN soumis à des perturbations électriques. Ils sont recommandés à un niveau relativement faible pour tous les niveaux d'ASIL.

(1c) Unité de contrôle externe. Le « chien de garde » logiciel ou matériel est un composant externe à la cible fonctionnelle. Il la surveille de manière périodique à l'aide d'un compteur. Le logiciel charge une valeur à l'initialisation de l'application, puis périodiquement, il force le rechargement du compteur avec cette valeur, avant que celui-ci n'arrive à zéro. Si le compteur atteint zéro avant que le logiciel n'ait eu le temps de réinitialiser le chien de garde, cela signifie que le logiciel ne fonctionne plus correctement sur le plan temporel. Cette technique est couramment utilisée pour détecter les retards temporels de tâches, les boucles infinies, et les suspensions d'interruptions. Le degré de recommandation de cette technique est croissant selon le niveau d'ASIL.

(1d) Gestion du flot de contrôle. L'idée de la gestion du flot de programme (Program Flow Monitoring [Yau & Chen 80]) est de partitionner le programme en blocs élémentaires associés à des signatures. La comparaison de la signature à l'exécution avec une signature pré-calculée permet de détecter des erreurs de flot de contrôle. Le degré de recommandation de cette technique est croissant selon le niveau d'ASIL.

(1e) Conception diversifiée du logiciel. "La conception logicielle diversifiée n'implique pas forcément la programmation en n-versions". La méthode « N-Version Programming (NVP) » ou « N-Self-Checking Programming (NSCP) » utilise N versions logicielles développées indépendamment, fonctionnellement équivalentes, à partir de la même spécification initiale. La même tâche de développement est confiée à N individus ou groupes qui n'interagissent pas, de manière à ce que les versions soient aussi diversifiées que possible. Les versions sont exécutées en parallèle avec des entrées identiques, les sorties sont soumises à un vote majoritaire. Cette technique implique un coût de développement élevé, mais la diversification est la seule méthode permettant de détecter en ligne des fautes de conception. Ainsi, elle est réservée aux niveaux d'ASIL les plus élevés. Si les contraintes économiques ne permettent pas de mettre en œuvre la programmation en n-versions dans l'automobile, une autre forme de diversification inspirée des techniques de validation est éventuellement moins coûteuse. Le minimum est d'implémenter deux versions, avec une des versions dérivant naturellement des spécifications fonctionnelles, et l'autre à partir des spécifications de sûreté, qui représentent deux aspects différents du même produit.

II.1.2 Techniques de recouvrement d'erreur

Les familles de recouvrement d'erreurs recommandées par la norme sont présentées dans le tableau 7 et commentées ci-après. Ce tableau fait partie de la même exigence que les mécanismes de détection d'erreurs (cf. Tab.6).

Méthodes		**ASIL**			
		A	B	C	D
1a	Mécanisme de recouvrement statique	+	+	+	+
1b	Dégradation acceptable	+	+	++	++
1c	Redondance parallèle et indépendante	o	o	+	++
1d	Codes correcteurs pour les données	+	+	+	+

Table 7 : ISO26262 – Recouvrement d'erreur au niveau de l'architecture logicielle

(1a) Mécanisme de recouvrement statique. "Les mécanismes de recouvrement statiques peuvent être réalisés par des blocs de recouvrement, du recouvrement par reprise, du recouvrement par poursuite, et du recouvrement par compensation". Le recouvrement par reprise consiste à ramener le système dans un état antérieur. Le recouvrement par poursuite recherche un nouvel état acceptable pour le système à partir duquel il pourra fonctionner (éventuellement dans un mode dégradé). Le recouvrement par compensation permet de transformer l'état erroné en état exempt d'erreur. Le terme « statique » signifie que les états que le système peut prendre lors du recouvrement sont prédéfinis. Cette recommandation du tableau est donc assez générale, de nombreuses méthodes sont possibles. En accord avec cette remarque, le degré de recommandation est moyen pour tous les niveaux d'ASIL.

(1b) Dégradation acceptable. "Une dégradation acceptable au niveau logiciel revient à affecter des priorités aux fonctions pour minimiser les impacts des défaillances potentielles sur la sécurité fonctionnelle". Prenons l'exemple du module de gestion centralisé des erreurs de l'architecture AUTOSAR, appelé « *Fault Inhibition Manager* ». Ce module est chargé de récupérer les notifications d'erreurs provenant des composants logiciels applicatifs ou du logiciel de base. Il interprète l'état du système, à partir de tables prédéfinies, qui définissent les fonctions applicatives à inhiber étant donné une erreur détectée. La priorité des fonctions et l'impact de l'inhibition de fonctions sont à étudier au cas par cas pour une application donnée. L'estimation de la minimisation de la dégradation est arbitraire, selon les exigences du constructeur automobile ou du concepteur de l'application. Plus le niveau d'ASIL est élevé, plus ce type de recouvrement est recommandé.

(1c) Redondance parallèle et indépendante. La redondance permet de réaliser du recouvrement par compensation. Le surcoût temporel d'une compensation d'erreur est beaucoup plus faible que celle d'une reprise ou d'une poursuite. Il existe deux approches. La première solution, reposant sur deux composants autotestables exécutant en redondance active le même traitement [Chérèque & al. 1992], consiste à déconnecter le composant défaillant, après une détection d'erreur. Ainsi, le traitement se poursuit sans interruption. La deuxième approche est appliquée systématiquement, même en l'absence d'erreur. Par exemple, le vote majoritaire [Avizienis & al. 1985]

consiste à évaluer systématiquement les résultats majoritaires de l'exécution de trois (ou plus) composants identiques. "Pour que la redondance parallèle soit indépendante, les logiciels doivent être différents dans chaque chemin parallèle". Cette technique implique un coût élevé en termes de ressources, mais la redondance est la seule méthode qui permet de corriger efficacement en-ligne les erreurs dues à des fautes matérielles importantes (ex : panne de calculateur). Ainsi, elle est réservée aux niveaux d'ASIL les plus élevés.

(1d) Codes correcteurs pour les données. Les techniques de codes correcteurs d'erreurs [Chen & Hsiao 1984] sont un cas particulier de redondance, pour corriger les erreurs de transmission de messages. Les codes correcteurs d'erreurs ont le même degré de recommandation que les codes détecteurs d'erreurs (cf. Tab.6).

II.1.3 Conclusion

La norme ISO26262 donne un état de l'art des méthodes de tolérance aux fautes actuelles. Ces principes sont vraisemblablement applicables aux logiciels embarqués automobiles, car la plupart des systèmes intègrent déjà ces mécanismes. Certaines techniques coûteuses mais efficaces, comme la diversification et la redondance, sont très recommandées pour les applications fortement critiques.

Les recommandations des tableaux 6 et 7 étant toutes alternatives (cf. section I.4.2.), c'est au concepteur du système de choisir l'association de mécanismes de tolérance aux fautes judicieux pour son système, et de justifier ces choix. Pour cela, nous présentons dans les sections suivantes des stratégies classiques, aussi bien dans l'automobile que dans d'autres domaines. Ces stratégies de détection d'erreurs ou de recouvrement d'erreur organisent des techniques de manière à former une architecture cohérente de détection et/ou de correction d'erreurs. Les architectures présentées combinent généralement plusieurs méthodes, c'est à dire plusieurs lignes des tableaux de recommandations de l'ISO26262.

II.2 Architectures classiques de tolérance aux fautes

Tolérer des fautes revient à créer des barrières pour prévenir la propagation d'erreur et la défaillance du système. Dans cet esprit, le système doit permettre le confinement d'erreur autant que possible.

Ainsi, l'approche classique de détection d'erreur est le confinement d'erreur à l'intérieur des couches logicielles, que nous appellerons « **confinement horizontal** ». Lorsque le confinement d'erreur cible l'intérieur des composants logiciels, nous parlerons de « **confinement modulaire** ». Enfin, nous étendrons le confinement de manière verticale (« **confinement vertical** »), c'est-à-dire transversal aux couches logicielles.

II.2.1 Confinement horizontal

II.2.1.1 E-GAS

E-GAS est un concept de sécurité standardisé [EGAS 2004], proposé par les constructeurs allemands BMW, Daimler Chrysler, VolksWagen, Porsche, Audi. Il donne des instructions spécifiques de conception pour le contrôle des moteurs essence et diesel. Ce concept est déjà largement adopté dans le monde automobile. E-GAS considère une structure à 3 niveaux logiciels de sécurité L1, L2, et L3, complètement indépendants. Ainsi, cette technique est un exemple de réalisation du confinement « horizontal » d'erreur à l'intérieur de chaque niveau. La description des différents niveaux donne des exemples de mécanismes de détection et de recouvrement d'erreur pour les types d'erreurs ciblés, différents selon les niveaux :

- L1 contient les fonctions applicatives et certains services de traitements de diagnostic sur les variables d'entrée/sortie, contenus dans les applications (cf. méthode 1a, Tab.6). Les capteurs, actionneurs ou réseaux défaillants sont détectés par ces tests logiciels de vraisemblance, au niveau de l'interface des composants logiciels applicatifs.
- L2 est dédié à la surveillance de l'ensemble des composants applicatif, dans le système. Par exemple, la comparaison du couple courant calculé avec le couple autorisé calculé permet de détecter une défaillance (cf. méthode 1a, Tab.6). En cas d'erreur, plusieurs comportements sont prévus : réinitialisation, arrêt de l'actionneur, mise en mode dégradé (cf. méthode 1a, Tab.7). Le périmètre de contrôle de ce niveau L2 englobe la couche applicative et, au-delà du logiciel, la structure fonctionnelle du système.
- L3 a pour rôle de s'assurer que le support matériel et électronique fonctionne, indépendamment des applications qu'ils supportent. On trouve donc des tests logiciels et matériels de RAM, d'ALU, du *program flow monitoring*, etc. (cf. méthodes 1c, 1d, Tab.6). La zone de contrôle de ce niveau contient essentiellement le calculateur.

Ici les niveaux de l'architecture de tolérance aux fautes différencient les erreurs fonctionnelles, ou dépendantes de l'application (L2), des erreurs non dépendantes de l'application (L3). L1 représente les mécanismes faisant partie de l'application.

II.2.1.2 SW FDIR

Dans le contexte spatial, le concept du FDIR (*Fault Detection, Isolation and Recovery*) est organisé, comme celui d'E-GAS, en niveaux hiérarchiques. Par exemple, pour un satellite particulier [Salkham 2005], les niveaux sont les suivants :

- L0 contient les mécanismes implémentés par le matériel, comme les chiens de garde matériels, les unités matérielles de contrôle de température, etc. (cf. méthodes 1c, 1d, Tab.6).
- L1 contient des fonctions logicielles intelligentes qui détectent par exemple des défaillances de boucles de contrôle. (c'est le L1 de E-GAS)
- L2 est dédié à la surveillance du niveau applicatif, il contient la surveillance des paramètres et des actions. (c'est le L2 de E-GAS)

- L3 contient des autotests et des chiens de garde logiciels qui surveillent l'exécution des tâches. (c'est le L3 de E-GAS)
- L4 est le module de recouvrement qui choisit les processeurs redondants à utiliser et qui déclenche les procédures de reconfiguration (cf. méthodes 1a, 1b, 1c, Tab.7).

Les concepts EGAS et FDIR se ressemblent beaucoup, certains niveaux sont équivalents (L1, L2 et L3). Le niveau L0 du FDIR contient ici des mécanismes de protection purement matériels. Le niveau L4 introduit l'idée de la séparation du recouvrement et de la détection d'erreur.

II.2.2 Confinement modulaire

II.2.2.1 Composants autotestables

Un composant autotestable [Anderson & al. 1990] est obtenu en ajoutant un module de contrôle au composant purement fonctionnel (cf. Fig.5). Si certaines propriétés évaluées par le contrôleur ne sont pas vérifiées, une erreur est signalée. Cette technique réalise donc un confinement modulaire d'erreur au niveau du composant logiciel. Le contrôleur peut être implémenté par n'importe quel mécanisme de détection d'erreur en fonction de l'application étudiée (cf. méthodes 1a, 1c, 1e, Tab.6).

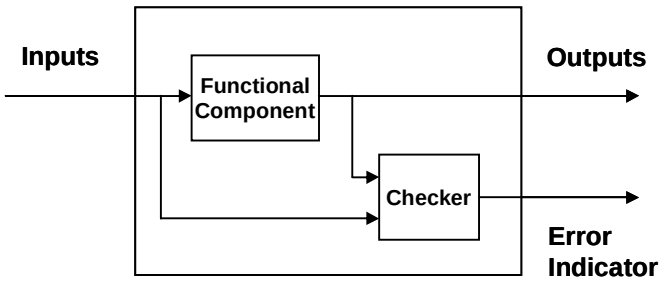


Figure 5 : Structure d'un composant autotestable

La structure du composant autotestable suit le principe de séparation des préoccupations. En génie logiciel, cette démarche de conception (SoC, *Separation of Concerns*) [Lopes & Hursch 1995] consiste à isoler, dans un système, des unités indépendantes ou faiblement couplées, et à traiter séparément chacune de ces unités.

L'intérêt de la « séparation des préoccupations » est de permettre au concepteur ou au développeur de se concentrer sur un problème à la fois, d'éliminer les interactions artificielles entre les aspects orthogonaux d'un système. Cette approche réduit ainsi la complexité de conception, de réalisation, mais aussi de maintenance d'un logiciel et en améliore la compréhension, la réutilisation et l'évolution. Réalisant la « séparation des préoccupations » à l'échelle du composant, le composant autotestable bénéficie alors des avantages de réutilisabilité, portabilité, maintenabilité, et diminution de complexité.

Le schéma « module fonctionnel – contrôleur – indicateur de validité des sorties » du composant auto-testable est déjà présent dans certains véhicules, pour la détection et le confinement d'erreurs des composants applicatifs. Les contrôleurs sont

généralement basés sur la vraisemblance des résultats par rapport à des plages de valeurs prédéfinies ou encore un modèle simplifié de fonctionnement.

II.2.2.2 EASIS Fault Management Framework

L'architecture EASIS est représentative du recouvrement dans le domaine automobile. Les erreurs détectées dans les composants logiciels (confinement modulaire), aux niveaux du logiciel applicatif ou du logiciel de base, sont centralisées et traitées par un mécanisme de sécurité appelé FMF (*Fault Management Framework*) [EASIS 2006]. Il existe des approches similaires à celle d'EASIS, avec le FIM (*Fault Inhibition Manager*) dans AUTOSAR [AUTOSAR 2007].

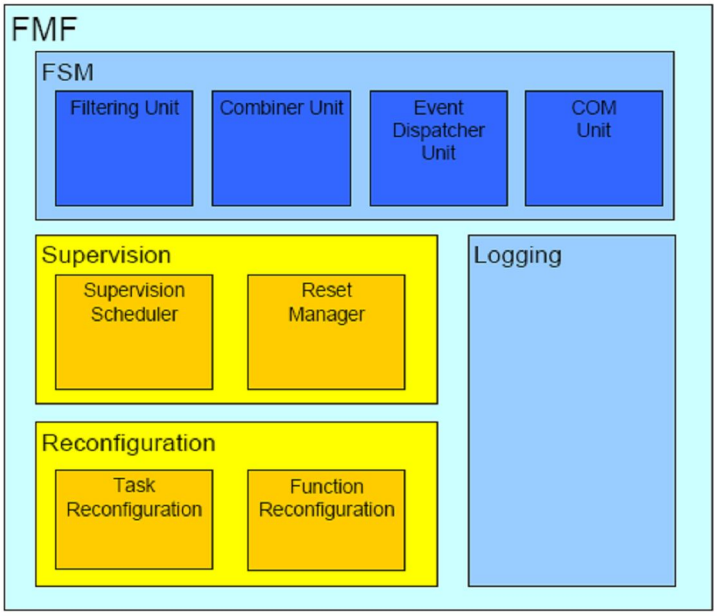


Figure 6 : Structure du Fault Management Framework

Le FMF détermine les actions de recouvrement à déclencher, à partir de conditions sur les erreurs définies statiquement. Il est organisé en quatre unités (cf. Fig.6) :

- *Fault State Manager* : il récupère les rapports d'erreur, les filtre, détecte des combinaisons d'erreurs, et distribue les informations aux autres unités.
- *Supervision* : il centralise les stratégies de gestion de fautes et commande la reconfiguration. Il est responsable de la synchronisation pour la reconfiguration distribuée.
- *Reconfiguration* : il fournit des mécanismes pour activer ou désactiver des fonctions ou encore relancer des tâches. (cf. méthodes 1a, 1b, Tab.7)
- *Logging* : il gère l'archivage des erreurs dans la mémoire non volatile.

II.2.2.3 Safety Bag

L'approche du « *Safety-Bag* » (cf. Fig.7) a été mise en œuvre pour les systèmes de signalisation ferroviaire Elektra [Kantz & Koza 1995], et commercialisée par *Alcatel Austria*. Pour chaque module de contrôle, les mécanismes de détection d'erreur

utilisés sont la diversification fonctionnelle (pour tolérer les fautes de conception) (cf. méthode 1e, Tab.6) et la redondance matérielle (pour tolérer les fautes matérielles) (cf. méthode 1c, Tab.7). Les composants fonctionnels forment une chaîne de traitement fonctionnel, tandis que les modules de contrôles forment une chaîne de traitement non-fonctionnel. Pour chaque module fonctionnel, il est possible d'associer un module de contrôle, pour rendre chaque composant autotestable, c'est pourquoi nous parlons encore de confinement modulaire d'erreur pour cette architecture. Les chaînes fonctionnelle et non-fonctionnelle sont réalisées par des équipes de programmation différentes. La première chaîne suit les spécifications fonctionnelles du contrôle de signalisation. La deuxième chaîne gère les conditions de sécurité : elle surveille les sorties de chaque fonction, pour vérifier qu'elles n'entraînent pas de situation critique. Si la chaîne non-fonctionnelle n'approuve pas les résultats, le système est mis dans un état sûr. Les spécifications pour la partie non-fonctionnelle proviennent des règlements d'exploitation *d'Austria Federal Railways*.

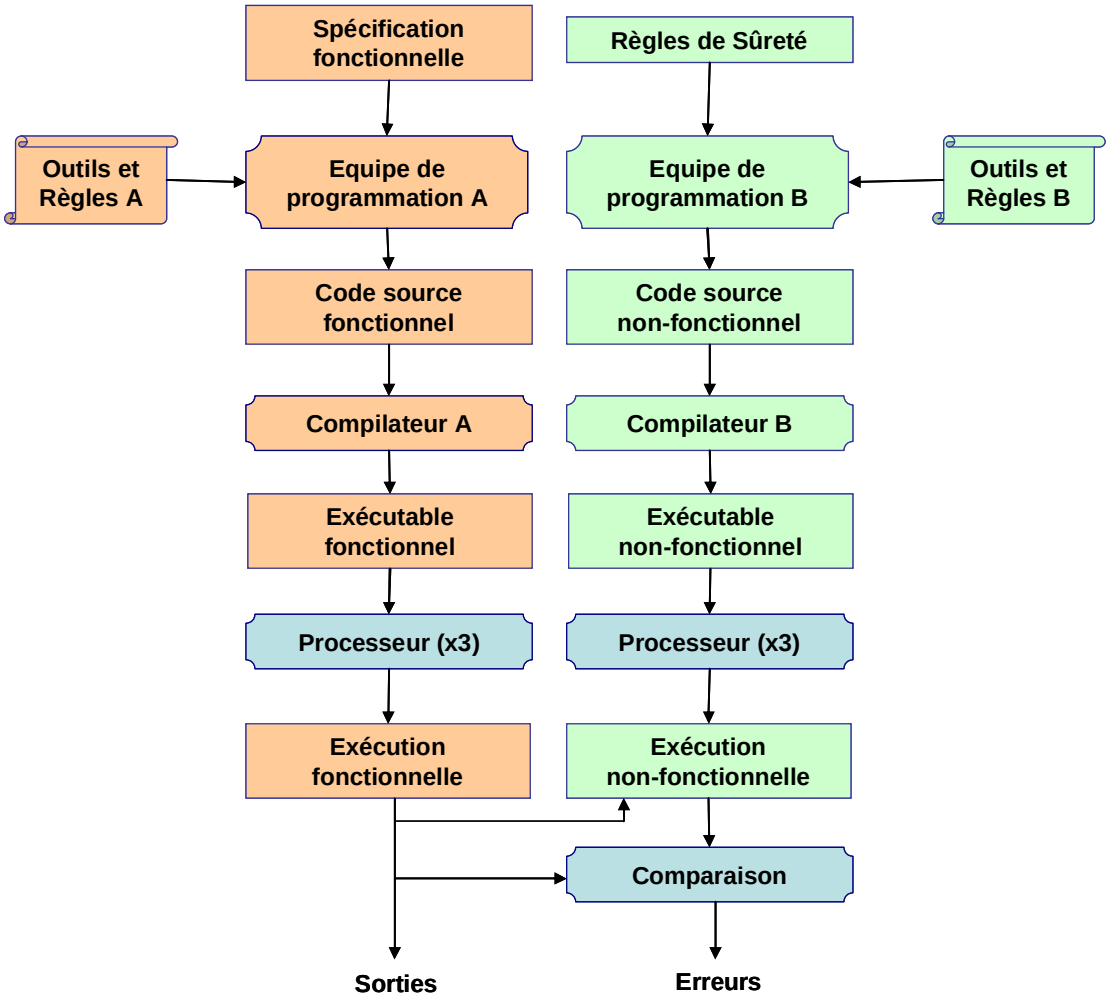


Figure 7: Safety-Bag Elektra

La redondance parallèle (cf. méthode 1c, Tab.7) et la conception diversifiée (cf. méthode 1e, Tab.6) sont des méthodes de tolérance aux fautes fortement recommandées par l'ISO26262, pour les applications à fort niveau de criticité (ASIL D).

II.2.3 Confinement vertical

La réflexivité [Smith 1983, Maes 1987, Kiczales & al. 1991] est la capacité d'un programme de raisonner sur son propre comportement et de pouvoir le modifier. Un système dit réflexif implémente un niveau d'abstraction, appelé **« méta-niveau »**, qui permet de surveiller et changer le comportement du système (**« niveau de base »**). La réflexivité a été étudiée au début plutôt dans le contexte des langages de programmation (Lisp, SmallTalk, Java…), et des systèmes d'exploitations commerciaux, tels que Aperios [Yokote 1992] développé chez Sony pour les systèmes de communication vidéo interactifs en 1995. Puis la réflexivité s'est étendue aux intergiciels comme DynamicTAO, University of Illinois [Kon & al. 2000]. Les intergiciels réflexifs sont généralement utilisés pour pouvoir adapter le comportement des applications distribuées, afin d'assurer des services non-fonctionnels : QoS, performance, sécurité, tolérance aux fautes, gestion d'énergie.

Dans le contexte de la sûreté de fonctionnement, le méta-niveau est l'implémentation des algorithmes de tolérance aux fautes, ou encore des assertions exécutables à vérifier (cf. méthode 1a, Tab.6). Le méta-niveau est externe au niveau de base (cf. méthode 1c, Tab.6). A l'interface entre le méta-niveau et le niveau de base, l'architecture proposée dans [Taïani 2004] introduit la notion **d'empreinte réflexive**, qui réunit les éléments suivants :

- Un ensemble de capacités d'observation qui décrivent de quelle manière le méta-niveau obtient des informations sur le système. Ces capacités d'observation peuvent soit prendre la forme de réification, ou d'introspection. Les informations sur l'activité du système sont réifiées lorsqu'elles sont transférées spontanément sous forme d'évènements au méta-niveau, l'initiative du transfert d'information résidant alors du côté du contexte d'exécution. Dans le cas de l'introspection, les informations sont fournies à l'algorithme à sa demande explicite.

- Un ensemble de capacités d'intercession permettant à l'algorithme de modifier le comportement du système, comme par exemple d'envoyer des messages particuliers, de forcer une reprise à partir d'un état déterminé, de rajouter des informations sur des requêtes distantes. Ces actionneurs logiciels permettent de placer le composant logiciel dans un état stable ou d'effectuer une opération de recouvrement.

On appelle « **méta-interface** » la traduction de l'empreinte réflexive en termes d'interface de programmation. En effet, l'empreinte réflexive définit de manière conceptuelle, ce dont le mécanisme de tolérance aux fautes a besoin en termes d'observabilité et de commandabilité, tandis que la méta-interface est l'interface des **sondes** et des **actionneurs logiciels** qu'il faudra implémenter pour mettre en œuvre de façon externe les mécanismes de tolérance aux fautes.

La notion de réflexivité multiniveaux [Taïani 2004] consiste à intégrer dans une même méta-interface des capacités réflexives à plusieurs niveaux de l'architecture logicielle (cf. Fig.8). Cette architecture réalise donc un confinement vertical d'erreur. De cette manière, les hauts niveaux d'abstraction peuvent puiser dans la richesse des informations des bas niveaux d'abstraction, auxquelles ils n'ont normalement pas accès. Inversement, les bas niveaux d'abstraction profitent de la connaissance fonctionnelle des hauts niveaux d'abstraction, qui leur fait généralement défaut. Cette

technique est particulièrement adaptée aux problèmes multicouches. Plutôt que d'agir localement, il faut avoir une perception globale du système. Pour un problème donné, il s'agit de mettre en place des moyens de détection et de recouvrement d'erreurs, à travers les différentes couches.

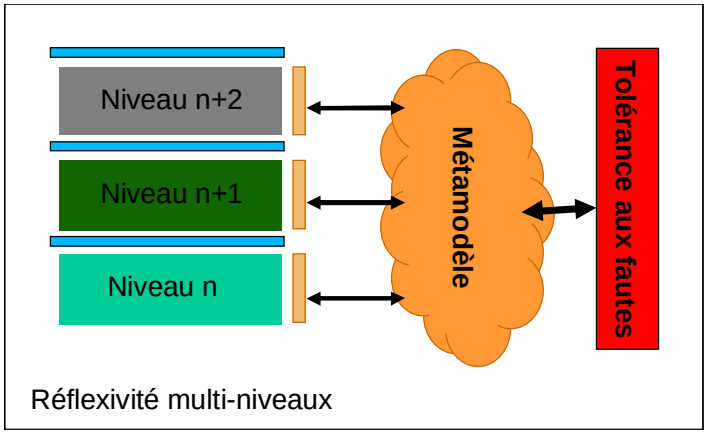


Figure 8 : Architecture réflexive multi-niveau

Cette approche permet de considérer le processus d'occurrence de fautes dans l'ensemble des couches d'un système complexe. Elle prend en compte le fait que le comportement d'une application dépend du bon fonctionnement des différents services sur lesquels elle repose. Toutes les applications ne nécessitant pas le même niveau de robustesse, elle est suffisamment flexible pour s'adapter au niveau de criticité d'une application particulière. Enfin, elle impose une démarche méthodique pour l'intégration de COTS dans la mesure où elle détermine par construction des pré-requis d'observabilité et de commandabilité, sous la forme d'une méta-interface et de capteur/actionneurs logiciels. Le compromis entre l'efficacité des mécanismes de tolérance aux fautes et les surcoûts temporels est alors ajustable.

II.2.4 Conclusion et position du problème

Dans les nouvelles architectures, le support d'exécution logiciel peut être partagé par de nombreux modules applicatifs de criticités différentes. Or, il constitue une source d'erreur au même titre que les modules applicatifs. Bien que le confinement modulaire protège à la fois les composants de l'infrastructure et ceux de la couche applicative, cette approche néglige les liens entre les éléments de l'architecture. Par exemple une préemption imprévue peut perturber la synchronisation des tâches et des données échangées, sans que le système d'exploitation et que le service de communication soient jugés défaillants. Le traitement de ce type d'erreurs nécessite d'observer et d'agir sur l'état de différentes couches logicielles à la fois. Ceci ne peut pas être résolu par le confinement horizontal. L'interaction entre les couches d'abstraction étant limitée, la surveillance des erreurs l'est aussi.

L'approche consistant à récupérer des informations et agir sur plusieurs niveaux d'abstraction est donc la plus séduisante. Ceci dit, il ne faut pas se perdre dans les détails de chaque couche d'abstraction et chercher les liens entre chaque couche. En effet, la tolérance aux fautes ne doit pas augmenter démesurément la complexité du logiciel, ni occuper trop de ressources ou dégrader les performances fonctionnelles.

II.3 Principe de la solution de tolérance aux fautes proposée

L'architecture de tolérance aux fautes proposée repose principalement sur le principe de réflexivité multi-niveau. L'idée est de rendre le système apte à **se connaître**, à **s'auto-surveiller** et **s'auto-corriger**, au niveau des différentes couches d'abstraction logicielle en fonction du besoin.

Conceptuellement, pour « se connaître », le système possède un **méta-modèle comportemental** et un **méta-modèle structurel** de lui-même (méta-niveau) :

Pour utiliser le principe de conception logicielle diversifiée (cf. méthode 1e, Tab.6) permettant de traiter les fautes de conception, notre méta-modèle comportemental du système est construit à partir des propriétés (non-fonctionnelles) de sûreté, au lieu des spécifications fonctionnelles. Un exemple de propriété de sûreté est énoncé comme suit :

> (**P**) : « Détection de pannes dangereuses (perte de l'ordre d'allumage des feux ou ordre d'extinction erroné) et maintien de la commande d'allumage le temps de la confirmation du défaut ».

Le méta-modèle est alors composé des différents types d'éléments qui composent les propriétés de sûreté. Pour l'exemple donné, le méta-modèle contient : une *séquence d'exécution* (pour l'allumage des feux), une *transition* (pour l'ordre d'extinction), une *action de recouvrement* (maintient de la commande d'allumage). Par la suite, l'intérêt de cette décomposition est d'associer à chacun des éléments du méta-modèle une stratégie générique de détection ou de recouvrement d'erreur : détecter une *séquence d'exécution* erronée, détecter une *transition* erronée, déclencher *une action de recouvrement*. Pour tenir compte de la diversité des applications embarquées automobile, nous proposerons, au chapitre III, une classification générique des exigences industrielles de sûreté, correspondant au méta-modèle comportemental.

Le méta-modèle structurel représente les différents types d'éléments de l'architecture logicielle du système (e.g. tâches, évènements, données, etc.), qu'il faut surveiller et/ou corriger. En l'occurrence, comme nous travaillons avec des cibles logicielles AUTOSAR, nous utilisons les méta-modèles AUTOSAR standardisés [AUTOSAR MM], décrits sous forme de diagrammes de classes UML. Pour la propriété (**P**) en particulier : la *séquence d'exécution* considérée est implémentée par une séquence de fonctions applicatives (*runnables*) contenues dans une même tâche ; la *transition* est réalisée lorsque le *runnable* A, ayant lu les valeurs des données a=1 et b=1 (*DataReadAccess*), écrit la donnée de commande c=1 (*DataWriteAccess*) ; *l'action de recouvrement* consiste à écrire la bonne valeur de la donnée (*DataWriteAccess*) lorsque le défaut a été confirmé. Les éléments soulignés de la phrase précédente sont des éléments du méta-modèle structurel.

En pratique, l'utilisation des méta-modèles comportementaux et structurels permet de traduire toute **propriété de sûreté du système** en **assertion exécutable**. La propriété (**P**) devient donc en pseudo-code :

Si (Tache courante = Tache Allumage des feux)

{**assertion1** (séquence correcte : *runnable* 1, *runnable* 2, *runnable* 3, *runnable* 4)}

Si (*DataWriteAccess*(c=1))

{**assertion2** (entrées correctes : *DataReadAccess*(a=1) et *DataReadAccess* (b=1))

Si erreur détectée {*DataWriteAccess*(c=0)}}

Les assertions exécutables contiennent la stratégie de tolérance aux fautes. Maintenant, pour que le système soit capable de s'auto-surveiller et s'auto-corriger, il s'agit d'instrumenter judicieusement le logiciel embarqué, c'est-à-dire placer des **sondes et des actionneurs logiciels**. Selon l'approche « **multi-niveau** », ils peuvent être distribués sur les différentes couches logicielles. Pour assertion1 par exemple, une sonde logicielle nécessaire est un service qui permet de récupérer les identifiants des fonctions applicatives *runnables* (information de la couche applicative). Mais une sonde est également nécessaire au niveau du logiciel de base pour récupérer l'identifiant de la tache courante. Pour assertion2, l'actionneur logiciel est le service fonctionnel d'écriture de donnée *DataWriteAccess*, de la couche de communication.

Le principe de séparation des préoccupations**,** sur lequel repose la réflexivité multi-niveau, garantit les propriétés de : portabilité, flexibilité, diminution de complexité, etc. Le stockage d'informations capturées, la vérification des assertions exécutables et la commande de recouvrement d'erreur constituent un « **logiciel de défense** » séparé du logiciel fonctionnel. Ces deux parties logicielles interagissent via une interface d'instrumentation (sonde et actionneurs logiciels), appelée « **empreinte réflexive** » (ou conceptuellement « méta-interface »). La figure 9 résume le principe de l'architecture réflexive proposée.

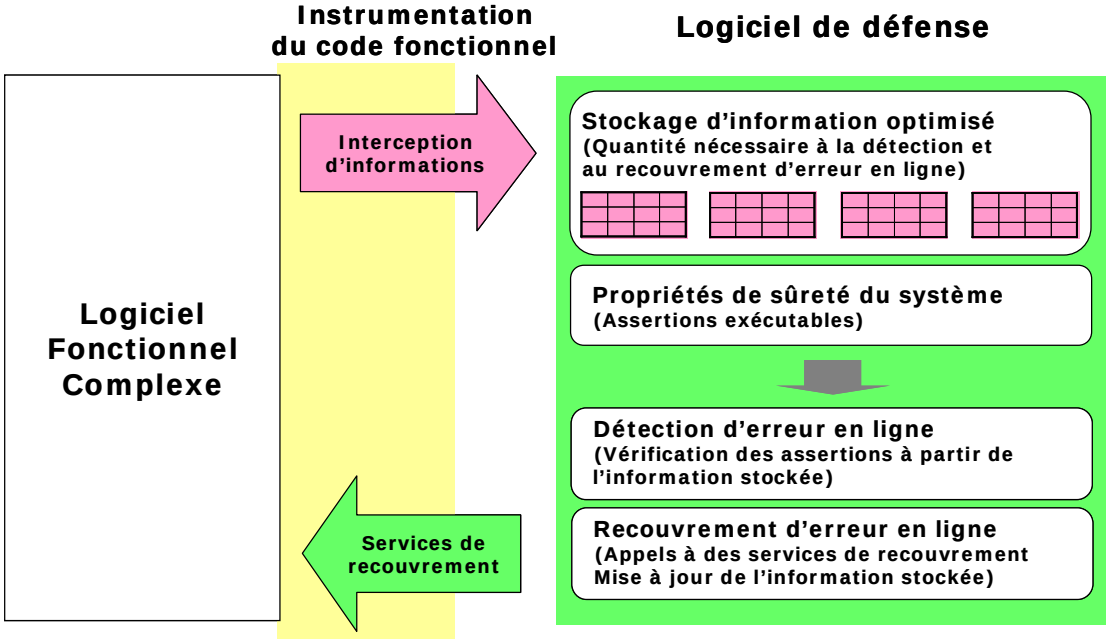


Figure 9 : Principe de réflexivité multi-niveau

Par rapport à la norme ISO26262, la vérification d'assertions à la base du logiciel de défense correspond à la méthode de contrôle de vraisemblance (cf. 1a, Tab.6) fortement recommandée pour tous les niveaux d'ASIL. Elle est classiquement utilisée pour détecter les fautes physiques et les fautes logicielles de codage. Par ailleurs, le logiciel de défense, externe au logiciel fonctionnel, et conçu à partir des propriétés de sûreté du système, est à la fois une unité de contrôle externe (cf. 1c, Tab.6) et une conception logicielle diversifié (cf. 1e, Tab.6). Ces types de techniques sont adaptés aux niveaux d'ASIL élevés C et D. Elles permettent en particulier de détecter des fautes logicielles de conception.

En termes de recouvrement, la capacité d'agir sur toutes les couches logicielles donne les moyens de mettre le système dans l'état souhaité, en corrigeant le flot d'exécution. Le stockage d'informations sur les états passés et/ou les états attendus permet de mettre en œuvre du recouvrement par reprise (état passé), par poursuite (état sûr), ou par compensation (état attendu) dans les limites du temps de détection et de recouvrement d'erreur. La stratégie choisie dépend de la propriété de sûreté à garantir, en tenant compte des contraintes temporelles de l'application critique. Si plusieurs fonctions de criticité différente cohabitent ensemble, et que des compromis sont nécessaires, le concepteur peut choisir de dégrader légèrement toutes les fonctions ou d'assurer la continuité de service de l'application la plus critique et de mettre les autres en mode dégradé. Ainsi, nous utilisons à la fois des mécanismes de recouvrement statique (cf. 1a, Tab.7) et de la dégradation acceptable (cf. 1b, Tab.7).

Méthodes de détection d'erreurs		**ASIL**			
		A	B	C	D
1a	Contrôle de vraisemblance	++	++	++	++
1b	Détection d'erreurs de données	+	+	+	+
1c	Unité de contrôle externe	o	+	+	++
1d	Gestion du flot de contrôle	o	+	++	++
1e	Conception logicielle diversifiée	o	O	+	++

Méthodes de recouvrement d'erreurs		**ASIL**			
		A	B	C	D
1a	Mécanisme de recouvrement statique	+	+	+	+
1b	Dégradation acceptable	+	+	++	++
1c	Redondance parallèle et indépendante	o	O	+	++
1d	Codes correcteurs pour les données	+	+	+	+

Table 8 : Architecture de tolérance aux fautes proposée et ISO26262

Pour résumer, dans la table 8, nous avons coloré les méthodes recommandées par l'ISO26262, qui forment la base de notre architecture de tolérance aux fautes. Les codes détecteurs (méthode de détection d'erreur 1b) et correcteurs d'erreurs (méthode de recouvrement d'erreur 1d), ainsi que les techniques génériques de gestion du flot

de contrôle comme le *program flow monitoring* (méthode de détection d'erreur 1d) sont des techniques complémentaires à notre solution. Par contre, la redondance parallèle et indépendante d'erreurs (méthode de recouvrement d'erreur 1c) est une implémentation possible de notre logiciel de défense, alternative aux assertions. En effet, la redondance permet intrinsèquement au système d'être réflexif, car la connaissance que le système a de lui même est matérialisée par les unités redondantes.

L'architecture de tolérance aux fautes proposée rentre donc complètement dans le cadre de sûreté de fonctionnement de la norme ISO26262 (cf. Tab.8), et s'adresse en particulier des applications automobiles à fort niveau de criticité (ASIL C et D).

II.4 Méthodologie proposée

La solution de tolérance aux fautes étudiée, à base d'assertions, est flexible par construction. En effet, elle traduit directement les besoins de sûreté des applications, en stratégies de détection et de recouvrement d'erreurs. Chaque logiciel de défense est donc spécifique à une cible donnée.

L'approche de tolérance aux fautes doit être générique et réutilisable pour toute application embarquée modulaire et multicouche. Selon le principe de « réflexivité », il s'agit de construire un « méta-modèle » générique du comportement et de la structure du logiciel embarqué automobile de manière générale (cf. Chapitre III). De cette manière, chaque propriété de sûreté des systèmes étudiés peut se décomposer selon le méta-modèle de référence, pour être traité de manière générique.

Le traitement générique de chaque élément du méta-modèle de référence, pour aboutir aux mécanismes de protection réflexifs (logiciel de défense et instrumentation associée), est guidé par une méthodologie rigoureuse [Lu *& al.* 2009 ETFA].

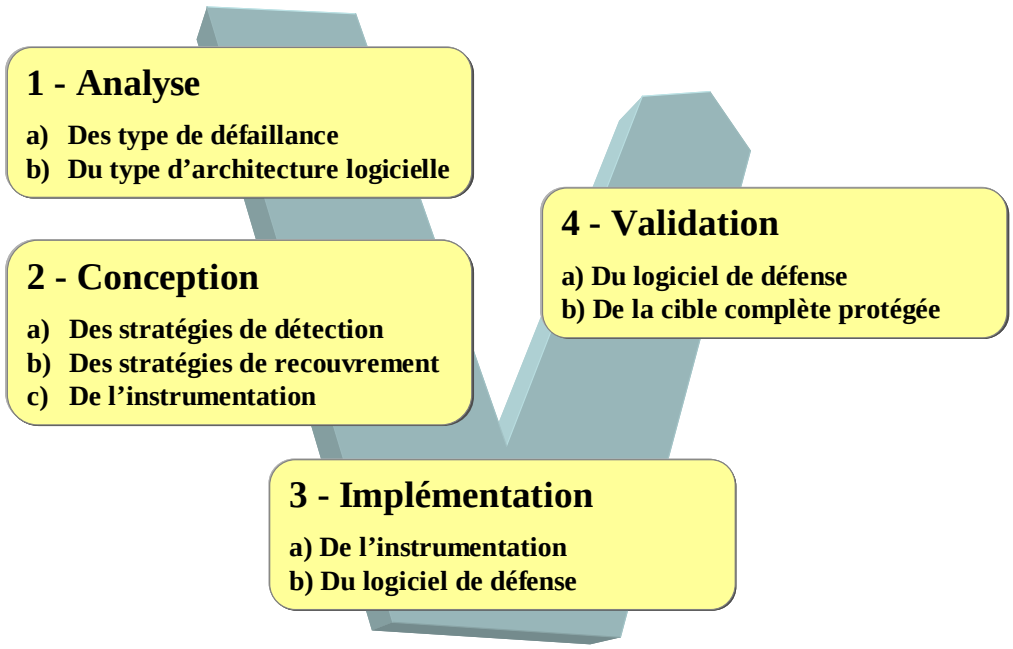


Figure 10 : Méthodologie globale

La démarche (cf. Fig.10) peut être exprimée selon les étapes classiques de développement d'un cycle industriel en « V » : analyse, conception, implémentation et validation. Le fil conducteur est donné dans cette section, tandis que des exemples et de plus amples détails de mise en œuvre font l'objet de chapitres à part entière par la suite.

Analyse. La première étape du travail est de définir le périmètre de propriétés de sûreté à traiter pour un système donné, et les traduire en assertions exécutables, qui constituent la base du logiciel de défense.

> **Etape 1a** L'analyse d'une exigence de sûreté, au niveau applicatif/fonctionnel, consiste en une décomposition en types génériques de défaillance élémentaire du logiciel. La classification des défaillances logicielles selon leur impact sur les flots de données ou de contrôle constitue une table de référence (cf. Tab.9). Elle exprime concrètement le concept de «méta-modèle de référence» (cf. Section II.3). Par ailleurs, si l'exigence de sûreté contient une indication sur les mesures à prendre en cas de défaillance, cette partie de l'exigence est traitée à l'étape 2b.
>
> **Etape 1b** Ensuite, la **connaissance de l'architecture logicielle** (en particulier aux niveaux : couche de communication et exécutif temps réel) permet de spécialiser les modes de défaillance élémentaire au niveau de l'implémentation. L'expression algorithmique définissant l'occurrence de ces défaillances élémentaires (au niveau implémentation) constitue les **assertions exécutables**.

Conception. Après avoir défini les assertions exécutables, faisant parti du logiciel de défense, il s'agit de spécifier le reste du logiciel de défense, ainsi que l'« empreinte réflexive » (interface d'instrumentation avec le logiciel fonctionnel).

> **Etape 2a** La partie du logiciel de défense responsable de la **détection d'erreurs** est constituée d'algorithmes de **vérification d'assertions** qui détectent et notifient les erreurs pour les services de recouvrement. Il y a éventuellement des algorithmes de **filtrage** qui sélectionnent les erreurs à corriger. L'information nécessaire à la détection d'erreurs et sa méthode de **stockage** dans le logiciel de défense doit aussi être précisé dans cette étape.
>
> **Etape 2b** Une fois une erreur notifiée, la partie du logiciel de défense responsable du **recouvrement d'erreurs** est constituée des algorithmes de **commandes** de modification de l'état du système. Soit la stratégie est donnée dans l'exigence de sûreté initiale et elle est appliquée ; soit le concepteur décide au cas par cas si l'application considérée requiert de revenir dans un état précédent, d'entrer dans un mode dégradé ou nécessite une continuité de service. L'information nécessaire au recouvrement d'erreurs (états précédents, états surs par défaut, etc.) et sa méthode de **stockage** dans le logiciel de défense doivent aussi être précisées dans cette étape.

Etape 2c L'**empreinte réflexive** est l'ensemble des services logiciels qui permettent de récupérer l'information nécessaire aux étapes 2a et 2b (sondes logicielles), et ceux qui permettent de réaliser les modifications souhaitées d'état du système pour l'étape 2b (actionneurs logiciels). Il existe deux types de sondes logicielles, qui permettent de réaliser soit de la réification, soit de l'introspection. Lorsque le logiciel fonctionnel produit de lui-même de l'information utilisable par le logiciel de défense, on parle de réification. Les notifications d'erreurs communiquées par le logiciel fonctionnel correspondent par exemple à de telles sondes logicielles. Lorsque la capture de l'information nécessite l'appel explicite à des services du logiciel fonctionnel, on parle d'introspection. Ces services utilisés (par exemple un service qui permet de récupérer l'identifiant d'une tâche) sont des sondes logicielles. L'empreinte réflexive est plus ou moins importante selon le nombre et la complexité des assertions et des types de recouvrement attendus.

Implémentation. Après avoir été spécifiés, le logiciel de défense et l'empreinte réflexive sont implémentés.

Etape 3a Les sondes et actionneurs logiciels de l'empreinte réflexive peuvent être déjà implémentés dans le logiciel fonctionnel, auquel cas les services existants seront directement utilisés par le logiciel de défense. Dans le cas contraire, ils peuvent être ajoutés au logiciel fonctionnel, si le code source est disponible ; sinon, il faut demander aux responsables des composants logiciels en boîte noire considérés d'ajouter les services et l'interface nécessaires à la sûreté. Cela revient en d'autres termes à **augmenter l'observabilité et/ou la commandabilité** des composants logiciels (COTS).

Etape 3b Le **logiciel de défense** doit être implémenté selon les spécifications des étapes 2a et 2b. Dans l'idéal, la robustesse du logiciel de défense (contenant toute la stratégie de tolérance aux fautes) doit être fortement renforcée. Intégrer la protection dans le calculateur fonctionnel entraine systématiquement des dégradations plus ou moins importantes de performance en temps et en mémoire. Dans la plupart des projets actuels à 90% de charge CPU, ce choix est impossible. A l'exemple du système Elektra (ferroviaire), il faudrait séparer le logiciel fonctionnel et le logiciel de défense, dans des calculateurs différents, avec une communication sécurisée entre eux. Moins de ressources matérielles signifient également moins de robustesse aux fautes physiques. En considérant des systèmes complexes à faible charge CPU, si des mécanismes matériels de protection mémoire (MPU) sont disponibles sur le calculateur, il est judicieux de les utiliser pour protéger le logiciel de défense.

Validation. La phase de validation est conduite, à l'aide de techniques d'injection de fautes en deux temps : la vérification du logiciel de défense et la validation du logiciel complet protégé.

> **Etape 4a** Les premiers tests d'injections de fautes visent à stimuler les mécanismes de tolérance aux fautes mis en œuvre. Ils permettent de détecter si l'intégration du logiciel de défense avec le logiciel fonctionnel est opérationnelle et si le logiciel de défense assure la protection spécifiée. La vérification permet donc d'**éliminer en particulier les fautes de programmation**
>
> **Etape 4b** La validation est nécessaire pour chercher des éventuels **problèmes de spécification** du logiciel de défense et évaluer la robustesse du système complet. Les tests d'injections de fautes sont cette fois moins ciblés que pour l'étape 4a et plus nombreux pour avoir des résultats statistiques exploitables.

Chacune de ces 4 phases de la méthodologie est détaillée dans un chapitre dédié. La phase d'analyse est explicité dans le chapitre III « développement des assertions de sûreté ». La phase de conception est décrite dans le chapitre IV « architecture du logiciel de défense ». L'implémentation est abordée dans le chapitre V « étude de cas ». Enfin la validation constitue le chapitre VI.

Conclusion du Chapitre II

Le standard ISO26262 donne un état de l'art des principes de tolérance à fautes actuels, sous formes de listes de méthodes recommandées. Malgré ces exigences, le concepteur du système reste libre de choisir l'association de mécanismes de tolérance aux fautes pour son système.

Différentes stratégies de détection et de recouvrement d'erreur existent, pour sélectionner les techniques de manière à former une architecture cohérente de tolérance aux fautes. Nous avons choisi le principe de la « réflexivité multiniveaux », qui est bien adaptée aux nouvelles architectures logicielles multicouches telles qu'AUTOSAR. La combinaison de méthodes de détections d'erreurs de l'ISO26262 utilisées est une unité de contrôle externe, appelée « logiciel de défense », à conception diversifiée (pour détecter en particulier les fautes de conception logicielles) et basée sur des contrôles de vraisemblance (adaptés à la détection de fautes physiques et de codage). Le recouvrement d'erreur est de type statique ou dégradation acceptable. Notre architecture de tolérance aux fautes est donc compatible avec le standard ISO26262.

Pour mettre en œuvre notre solution technique, nous proposons une méthodologie de développement rigoureuse en 4 phases classiques (analyse, conception, implémentation et validation), qui feront l'objet des 4 chapitres suivants. L'étape d'analyse vise à définir des assertions exécutables à partir de l'analyse des modes de défaillance de la cible et son architecture logicielle. La phase de conception consiste à définir les stratégies de détection et de recouvrement d'erreurs, relatives aux assertions exécutables, ainsi que l'instrumentation nécessaire au contrôle de la cible. L'implémentation matérialise le logiciel de défense et l'instrumentation (sondes et actionneurs logiciels). Enfin, la validation vérifie la conformité du système aux exigences de sûreté, et l'efficacité des mécanismes de tolérance aux fautes ajoutés.

Chapitre III

Développement des assertions de sûreté

Développement des assertions de sûreté

Ce chapitre présente d'un point de vue pragmatique la première étape de notre méthodologie (cf. II.4). Le logiciel fonctionnel considéré est supposé fortement critique. La connaissance du système sur son comportement nominal et son comportement attendu en cas d'erreur est spécifiée par le concepteur industriel sous forme d'exigences de sûreté. La solution technique proposée dite réflexive se base sur cette connaissance, pour la traduire en assertions exécutables. Pour généraliser l'étude d'applications particulières, nous introduisons une classification de référence des défaillances élémentaires du logiciel, faisant l'objet des exigences de sûreté. Cette classification générique a deux utilisations : la première est celle d'une table de translation des exigences industrielles en assertions exploitables pour construire notre logiciel de défense ; la seconde est la structuration d'une matrice de traçabilité, qui permet de suivre rigoureusement, pour chaque exigence, le développement des mécanismes de détection et de recouvrement d'erreurs associés jusqu'aux tests de vérification de conformité aux exigences.

Dans un premier temps, le contexte des exigences de sûreté industrielles est décrit et analysé. Ceci permet alors de proposer le tableau générique de classification des défaillances logicielles en section III.2. La section suivante explique la méthode pour décliner les exigences de sûreté en assertions. Enfin, la structure de la matrice de traçabilité des exigences est introduite dans la dernière section de ce chapitre et le contenu sera complété au fur et à mesure dans les prochains chapitres.

III.1 Exigences de sûreté industrielles

Une **exigence** correspond à ce qu'un produit doit faire, avec quelles performances, et sous quelles conditions, pour atteindre un but donné. C'est un énoncé qui prescrit une fonction, une aptitude, une caractéristique ou une limitation à laquelle doit satisfaire le produit dans des conditions d'environnement de données. Les exigences spécifiées ont deux origines : elles sont soit issues de l'analyse du besoin ou d'une négociation client-fournisseur, soit elles sont établies au cours des activités de conception ou de réalisation.

Les **exigences de sûreté**, qui nous intéressent, spécifient soit des comportements nominaux à respecter strictement (tout autre comportement est alors considéré défaillant), soit des modes de défaillance à éviter et des mesures à prendre, en cas d'occurrence. Pour définir ces exigences, la manière de procéder en termes de méthodologie, de langage, d'outils, etc. peut différer d'un constructeur ou équipementier à un autre. Il est donc difficile de présenter une vision générique du monde automobile. Deux types d'**approches** peuvent globalement être distingués : « **ascendante** » et « **descendante** ». La première correspond à de la propagation d'erreur, vers l'utilisateur au niveau fonctionnel, en prenant pour point de départ les hypothèses de fautes physiques et logicielles (cf. section I.2) susceptibles d'impacter le système, ou encore les modes de défaillances de chaque composant logiciel de base. La démarche descendante (adoptée en particulier par Renault), à l'inverse, dérive des pannes de très haut-niveau (prestations client), pour les décomposer, les analyser et les spécialiser progressivement au niveau du matériel, puis du logiciel. La préférence d'une approche par rapport à une autre pour l'industriel peut s'expliquer par son fort niveau de maîtrise du système à haut-niveau ou bas-niveau.

Il est plus naturel de partir des applications dont les défaillances ont un effet sur le client. Ainsi, compte tenu des informations (plutôt haut-niveau) dont nous disposons principalement, nous choisissons l'approche descendante, et présentons quelques notions particulières à Renault [RENAULT SdF]. La sûreté de fonctionnement du logiciel hérite du contexte du système (environnement, contraintes, etc.) dans lequel le logiciel est intégré. Une fois les spécifications fonctionnelles du système définies, il s'agit de réfléchir sur les objectifs de sécurité du système. Cette étape consiste à identifier les **EICs (Evénements Indésirables Clients)**. Renault définit un EIC comme suit : «*Libellé des conséquences probables de défaillance(s) du produit et/ou de comportement humain, aboutissant ou pouvant aboutir à un mécontentement client, ou à une atteinte à l'environnement jusqu'à des dommages corporels ou matériels importants*». Il existe différents types d'EICs. Un EIC relatif à la sécurité sera par exemple : l'impossibilité de sortir du véhicule. La perte d'accès au coffre est un EIC relatif à la dégradation de la prestation (e.g. confort) ou de la qualité de service. Les EICs non sécuritaires et non immobilisant sont traités par les démarches d'obtention de la qualité-fiabilité en développement et en production.

Les Evénements Indésirables Clients (EICs), dit sécuritaires, sont évalués selon leur **niveau de criticité**. Actuellement, le calcul du niveau de criticité (*Safety Integrity Level SIL*) est conforme à la norme IEC 61508, en attendant la publication de l'ISO26262. Il n'y a pas d'évènement catastrophique directement déclenché par un système véhicule aujourd'hui (au sens du SIL 4 de l'IEC61508). De nombreux systèmes embarqués automobiles répondent par contre à des enjeux de niveau de SIL

2, parfois 3 (correspondant aux niveaux d'ASIL C et D de l'ISO26262). Chez Renault, ce calcul correspond à l'évaluation du risque et de la probabilité d'occurrence des EICs (perte d'assistance dans une direction assistée électrique par exemple). Cette cotation prend déjà en compte les 3 critères de l'ISO 26262 : probabilité d'occurrence de la situation dangereuse, contrôlabilité par le conducteur, gravité de l'accident s'il se produit. Actuellement, Renault considère que le niveau de criticité du logiciel d'un calculateur est par défaut celui de l'événement de criticité la plus élevée à laquelle une défaillance du calculateur pourrait conduire.

Pour les EICs sécuritaires ou immobilisant, des outils comme l'AMDE ou les arbres de défaillance sont utilisés pour identifier les sources de défaillances matérielles ou logicielles pouvant mener à des **Evènements Redoutés Système (ERS)**. Un ERS est un mode de défaillance d'un sous-système ou d'un composant du système considéré. Plus particulièrement, pour les logiciels critiques, les ERS sont reformulées en termes logiciels, c'est-à-dire en termes de modes de défaillance sur les sorties du logiciel. On étudie chaque constituant par rapport à ce que l'on ne veut pas qu'il se produise sur sa sortie. La figure 11 donne un exemple sur un modèle Matlab/Simulink. L'ERS sur la donnée de sortie *ValueY* est traduite par rapport à ses entrées *SpeedOne*, *WheelAngle* et *InputYYY*. Les composants logiciels (*GlobalValueCheckOne* et *CalculateValueY*) sont également susceptibles de défaillir.

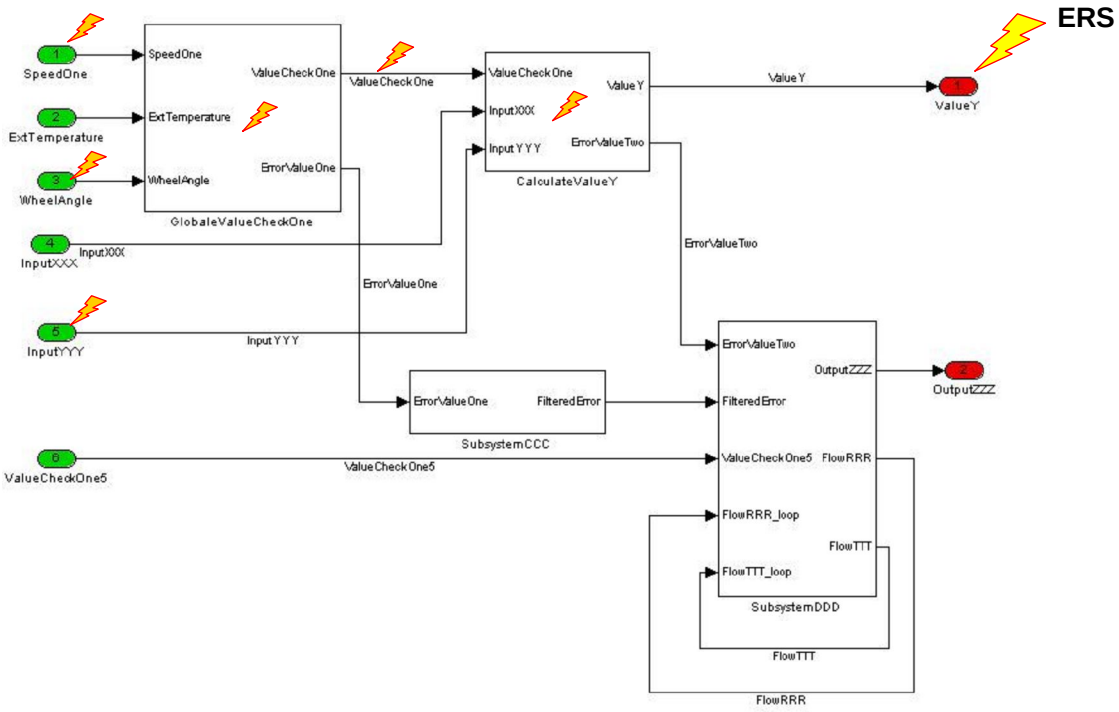


Figure 11 : Exemple d'Evènement Redouté Système (ERS)

Pour éviter les Evènements Redoutés Système, des « barrières de sécurité » sont mises en place. Elles peuvent être des contraintes sur le codage (programmation défensive, redondance logiciel, surveillance, vote, etc.) mais aussi sur l'architecture logicielle, par exemple à travers l'utilisation de média de communication robuste et déterministe garantissant le transit des messages. Par ailleurs, des mécanismes de surveillance sont aussi proposés pour détecter des pannes dangereuses ou des pertes

d'intégrité des barrières pour informer le client (invitation à adopter un comportement permettant de réduire la durée d'exposition au risque : incitation à faire réparer, incitation à s'arrêter).

Pour résumer, la figure 12 illustre les notions d'ERS, d'EIC et de barrières de sécurité, dans le cadre d'un véhicule composé de trois systèmes. Chaque partie de système susceptible de conduire à un ERS est protégée par une barrière de sécurité. **Une solution de tolérance aux fautes (en particulier celle proposée dans la thèse) pour éviter un ERS correspond donc au concept de « barrière de sécurité » dans le vocabulaire Renault.** Un ERS peut conduire à plusieurs EIC (cas de l'ERS1) et plusieurs ERS peuvent conduire ensemble ou indépendamment au même EIC (cas de l'EIC2).

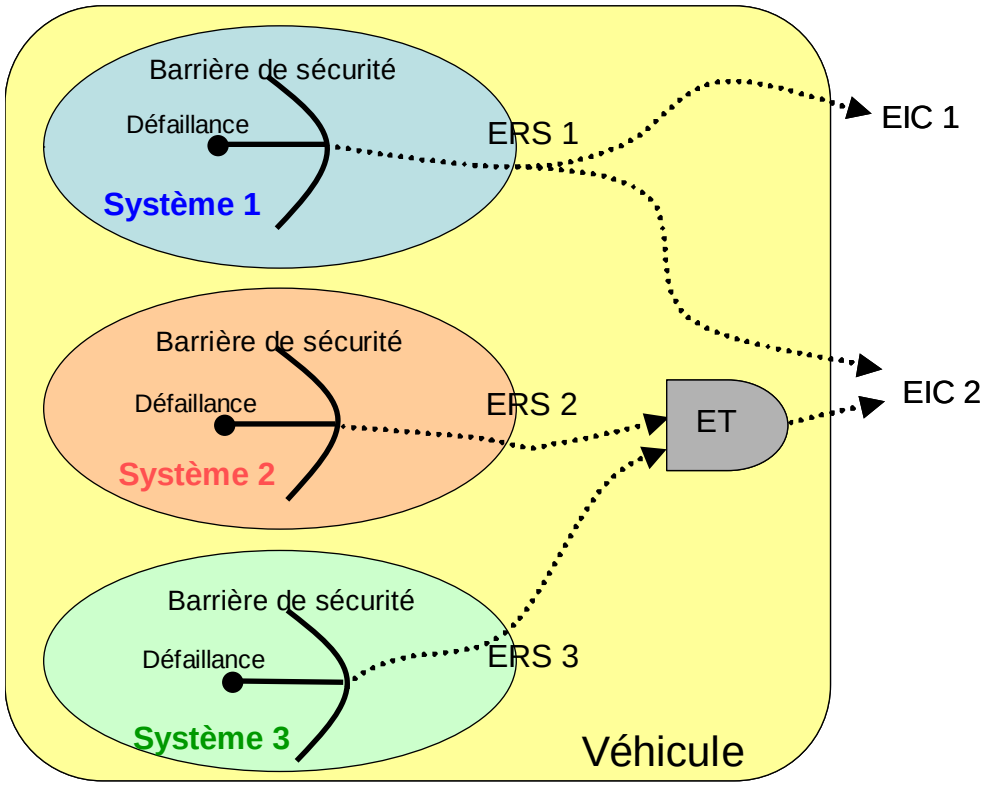


Figure 12 : Approche de sûreté de fonctionnement Renault

III.2 Classification des modes de défaillances logicielles

L'idée de la classification est d'être la plus générique possible et de couvrir le maximum d'exigences de sûreté concernant le logiciel (au niveau applicatif) à l'exécution. A partir de la synthèse d'une centaine d'exigences Renault [RENAULT spec.], et de modélisations classiques des systèmes temps réels à base de flot de données [Simulink], de flot de contrôle [Stateflow] ou de combinaison des flots données et de contrôle [Pernet & Sorel 2005], nous avons choisi de structurer les défaillances logicielles en **flot de données** et **flot de contrôle** (cf. Tab.1). Ces flots ne sont pas toujours séparables, la relation entre les données et le contrôle peut être complexe, mais le problème ici n'est pas de définir une classification orthogonale des défaillances logicielles. Pour les défaillances impliquant à la fois des données et du contrôle, une décomposition en défaillances élémentaires ou un choix arbitraire détermine le type de défaillance.

Flot	Type de Défaillance spécifique	
Flot de données	Valeur échangée	Invalide
		Non voulue
	Temps d'échange de donnée	Trop long
		Trop court
Flot de contrôle	Transition (déclenchement de calcul)	Invalide
		Non voulue
	Séquence d'exécution	Invalide
		Non prévue
	Temps d'exécution	Trop long
		Trop court

Table 9 : Classification de référence des défaillances du logiciel

De manière générique, un **flot de données** décrit une dépendance entre des données (variables, messages, ou paramètres) mais aussi entre des opérations de calcul ou de contrôle. Selon les langages de programmation, la modélisation séparée ou conjointe des flots de données et de contrôle peut sensiblement nuancer la définition des flots. Les catégories élémentaires d'exigences sur les flots de données, au niveau du logiciel applicatif, concernent la valeur échangée et le temps d'échange. Nous considérons qu'une contrainte sur la disponibilité d'une donnée peut se traduire plus par une contrainte en valeur (la donnée peut être différente d'une valeur par défaut) et/ou en temps (la donnée peut être produite dans un temps imparti).

Défaillance sur la valeur échangée. Au niveau du producteur et/ou du consommateur d'une donnée, la valeur échangée peut être erronée invalide ou non voulue**.** Une valeur est considérée invalide si elle sort de sa plage de validité (e.g.

vitesse à 300 km/h). Elle est non voulue, si elle est à l'intérieur de sa plage de validité (e.g. vitesse à 114 km/h au lieu de 50 km/h), mais une condition de dépendance de la valeur par rapport à d'autres informations (données et/ou évènements de contrôle) n'est pas vérifiée.

Défaillance sur le temps d'échange. En termes de durée d'émission et/ou de réception d'une donnée, le temps d'échange peut être trop court ou trop long, de manière absolue ou relative par rapport à une condition sur des données et/ou des évènements de contrôle.

Un **flot de contrôle** définit un ordre entre les algorithmes de calcul. L'exécution est déclenchée soit par des évènements de contrôle temporels ou discrets, soit par un état atteint, caractérisé par un ensemble de valeurs. Les catégories élémentaires d'exigences sur les flots de contrôle, au niveau du logiciel applicatif, concernent :

Défaillance sur le déclenchement de transition. L'occurrence d'évènements de contrôle, ou d'un état donné (ensemble de valeurs), ou encore d'une condition en logique universelle faisant intervenir des évènements de contrôle et des données, pour permettre le déclenchement d'un calcul (transition), peut être invalide (transition inhibée) ou non voulue (transition suractivée). Une condition est invalide lorsqu'elle n'est pas vérifiée alors qu'elle le devrait (e.g. un capteur physique est en panne, ce qui perturbe la perception de l'environnement et inhibe des conditions au niveau logiciel). Une condition est non voulue lorsqu'elle est vérifiée alors qu'elle ne devrait pas (e.g. un capteur physique est en panne, ce qui perturbe la perception de l'environnement et valide à tort des conditions au niveau logiciel).

Défaillance sur la séquence d'exécution. La séquence (linéaire) ou graphe (avec branchements) d'exécution des algorithmes de calcul peut être invalide ou non prévue, de manière absolue ou relative par rapport à une condition sur des données et/ou des évènements de contrôle. Une séquence est invalide si elle a été identifiée par une analyse de sûreté comme séquence défaillante à éviter. Une séquence est non prévue si elle n'a ni été identifiée comme fonctionnelle, ni comme défaillante. Le système est alors dans un état indéfini, potentiellement non sûr.

Défaillance sur le temps d'exécution. Le temps d'exécution d'un algorithme de calcul isolé ou d'un ensemble d'algorithmes (durée, délai, périodicité, etc.) peut être trop court ou trop long de manière absolue ou relative par rapport à une condition sur des données et/ou des évènements de contrôle.

Ainsi les défaillances sur la valeur d'une donnée, le temps d'échange d'une donnée, une transition, une séquence d'exécution et un temps d'exécution, sont les 5 catégories de référence, qui structurent notre analyse. Les sous-catégories (invalide, non voulu, trop court, trop long) sont différentiées, car elles conduisent à des stratégies de détection ou de recouvrement d'erreurs parfois sensiblement différents (cf. Tab.10, Tab.11).

III.3 Assertions exécutables spécifiques

Un système complètement réflexif, en traitant une centaine d'exigences de sûreté pour une application donnée, pourrait fortement augmenter la complexité du logiciel et diminuer les performances temps réel du système (dans le cas où le logiciel de défense est intégré dans le même calculateur que le logiciel fonctionnel). Un tri préalable des exigences les plus critiques (ASIL C ou D) à traiter est donc judicieux.

Une fois le jeu d'exigences de sûreté sélectionné, la démarche décrite ci-après permet d'obtenir les assertions exécutables qui seront directement utilisées par le logiciel de défense.

III.3.1 Décomposition d'une exigence selon la Table 9

Chaque exigence est décomposée (étape 1a, section II.4) de manière indicative selon la table de référence (cf. Tab.9). Le principe est expliqué par quelques exemples. Nous pourrions reprendre les exigences Renault (Exemple1) qui sont à l'origine de la table de référence. Cependant, il est intéressant de présenter d'autres exemples automobiles, pour montrer que l'approche est généralisable. Les exemples 2 et 3 sont donc extraits de l'annexe du concept de sécurité EGAS [EGAS], sur le contrôle des moteurs essence et diesel. Les exigences se présentent sous la forme d'une erreur à éviter et les mesures à prendre en cas d'erreur pour chaque type de moteur (essence à injection multiple, essence à injection directe et diesel).

Exemple1. « La fiabilité des séquences logicielles d'envoi d'ordre de fermeture client et d'autorisation de fonctionnement de fermeture du toit ouvrant est exigée. ».

La défaillance considérée est une défaillance sur le *flot de contrôle*, concernant une *séquence d'exécution* soit *invalide*, soit *non attendue* (« fiabilité des séquences logicielles»). A l'implémentation, il sera nécessaire de préciser la nature (fonctions applicatives, tâches ou évènements) des éléments séquencés (« ordre de fermeture client», « autorisation de fonctionnement de fermeture du toit ouvrant »).

Exemple2. « Un message défectueux/manquant (antiblocage des roues sur lever de pied, transmission, etc.) pour le requête d'augmentation de couple externe est une erreur. Pour les trois types de moteur, la requête est bloquée ».

Ici plusieurs interprétations sont possibles. A ce niveau (étape 1a de notre méthodologie, II.4), nous ne pouvons pas faire de choix d'interprétation. L'étape suivante d'analyse au niveau architectural (étape 1b de notre méthodologie, II.4) le permettra.

La première solution identifie les types de défaillances, comme des défaillances sur le *flot de donnée*, concernant soit une *valeur échangée invalide* (« message défectueux »), soit un *temps d'échange de donnée trop long* (« message manquant »). La donnée considérée est un message. A ce niveau fonctionnel, dans cette exigence, les contraintes en valeur ne sont pas données précisément, ni les contraintes temporelles qui décident qu'une donnée est manquante. Elles seront nécessaires pour exprimer l'assertion exécutable au niveau implémentation.

La deuxième solution identifie le type de défaillances, comme une défaillance sur le *flot de contrôle*, concernant une *transition invalide* (« requête d'augmentation de couple externe »). La nature de la requête n'est pas précisée, à ce niveau.

Le recouvrement « la requête est bloquée » est mis de côté pour la stratégie de recouvrement du logiciel de défense.

Exemple3. « Une durée inappropriée de déclenchement d'injection est une erreur. Pour un moteur à essence en injection directe, une transition à une opération homogène doit être effectuée. Pour un moteur diesel, le moteur doit être coupé après un nombre applicable de réinitialisations ».

Sans information complémentaire sur l'implémentation, plusieurs interprétations sont aussi possibles dans cet exemple.

La première solution identifie le type de défaillances, comme une défaillance sur le *flot de contrôle*, concernant un *temps d'exécution trop long* (« durée inappropriée de déclenchement d'injection »). Le déclenchement d'injection peut représenter un élément de contrôle (tâche, évènement, etc.).

La deuxième solution identifie le type de défaillances, comme une défaillance sur le *flot de donnée*, concernant un *temps d'échange de donnée trop long* (« durée inappropriée de déclenchement d'injection »). Le déclenchement d'injection peut représenter un échange de message.

III.3.2 Déclinaison au niveau architectural (implémentation)

La décomposition des exigences au niveau fonctionnel/applicatif selon la table 9 donne un fil conducteur de l'analyse, elle est indicative, mais elle manque de détails. Elle est donc insuffisante pour exprimer une assertion exécutable.

Le méta-modèle de l'architecture logicielle étudiée doit être connu. Plus particulièrement, en décomposant le flot d'exécution en données et contrôle, les services de support d'exécution qu'il est important de dégager sont ceux qui gèrent les données et le contrôle, soit les **services de communication et de l'exécutif temps réel**.

Nous présenterons plus précisément l'architecture logicielle AUTOSAR en déployant une étude de cas complète, dans le chapitre V. Pour rester général, ce qu'il faut tirer de la connaissance de l'architecture pour préciser les modes de défaillances sont les types d'informations suivants :

- Globalement, pour le *flot de données*, les éléments échangés sont les données. Les caractéristiques nécessaires au traçage de la donnée et à la détection d'erreurs sont classiquement : sa **nature** (variable, message, paramètre), son **identifiant**, ses **producteurs et consommateurs**, son **mode** de production et de consommation (e.g. client-serveur), sa **plage de validité**, son **type**, ses **contraintes temporelles** d'émission et de réception, etc.

- Pour le *flot de contrôle*, les éléments dynamiques principaux sont les tâches (caractérisés par un **identifiant,** un **comportement temporel** périodique ou

apériodique attendu). Leur cycle de vie dépend d'actions de contrôle (e.g. service de l'exécutif temps réel) ou d'occurrence d'objets de contrôle (e.g. évènement, alarme, etc.), pour activer, désactiver ou mettre en attente les taches. Les actions de contrôle sont caractérisées par une **source** émettrice de l'action (e.g. routine d'interruption) et la **tâche impactée par l'action**. Les objets de contrôle sont caractérisés par un **identifiant**.

Reprenons le cas de l'exemple 1 de la section III.3.1.

Exemple1. « La fiabilité des séquences logicielles d'envoi d'ordre de fermeture client et d'autorisation de fonctionnement de fermeture du toit ouvrant est exigée. ».

Au niveau fonctionnel, la défaillance considérée est une défaillance sur le *flot de contrôle*, concernant une *séquence d'exécution* soit *invalide*, soit *non attendue* (« fiabilité des séquences logicielles»).

Les éléments séquencés sont des fonctions applicatives différentes pour la commande de fermeture de toit ouvrant du client (identifiant : F1) et le traitement de la commande (identifiant : F2). Ces fonctions sont implémentées dans des tâches différentes :

- F1 dans la tâche apériodique T1 activée sur interruption. A la fin de l'exécution de F1, la commande est implémentée par l'appel d'un service d'activation de l'OS.

- F2 dans la tâche apériodique T2 activée sur demande d'activation de la part de T1.

L'exigence impose que la séquence {F1, F2} doit être atomique. En d'autres termes, l'exécution de F1 s'est terminée avant que F2 s'exécute. L'assertion exécutable suivante (1a) décrit donc le cas nominal en pseudo code :

Assertion1a.

T2 = tâche courante ;

Fin d'exécution F1 = ok ;

Notons que les détails d'implémentation peuvent appuyer ou modifier le type de défaillance analysé a priori. Ici, si l'information de fin d'exécution de F1 est aisée à obtenir, l'assertion convient. Si ce n'est pas le cas, il est possible d'interpréter l'exigence de *séquence d'exécution* en termes de *transition* de F1 à F2 par le service d'activation de l'OS. Une assertion alternative (1b) peut alors être utilisée :

Assertion1b.

T2 = tâche courante ;

Activation réalisée par T1 = ok ;

Le choix optimisé en fonction de l'architecture est laissé au concepteur du logiciel de défense. Le produit de cette étape d'analyse est d'obtenir des assertions utilisables pour la détection d'erreur. Les informations qui les composent sont récupérables dans

le système par une instrumentation appropriée, potentiellement distribuée dans des couches différentes du logiciel.

Une approche aussi flexible est potentiellement une source d'erreurs si elle n'est pas maîtrisée. Pour faire face à cet inconvénient, il est nécessaire de tracer chaque exigence, leur décomposition, leurs assertions, et la suite du développement du logiciel de défense qui leur correspond. C'est pourquoi, nous introduisons une « matrice de traçabilité ».

III.4 Matrice de traçabilité des exigences de sûreté

La notion de **traçabilité** peut faire référence soit à la trace entre le besoin et la solution, soit à la trace des choix, soit à la logique de déclinaison des exigences selon l'arborescence technique, ou encore à une logique de validation ou preuve de la pertinence de la solution. Pour nous, il s'agit de suivre chaque problème critique depuis sa spécification de haut niveau, sa déclinaison au niveau de l'architecture, puis l'implémentation des mécanismes de protection associés, jusqu'à l'étape de validation du système. C'est une méthode qui a pris son essor dans l'industrie aérospatiale (DO178B, IEC61508), et qui représente un élément essentiel de la certification.

Notre **matrice de traçabilité** (cf. Tab.10) se décompose en **8 colonnes** numérotées de 1 à 8 (sans compter les exigences industrielles d'origine – colonne 0), qu'il faut détailler pour chaque exigence étudiée (correspondant à chaque ligne de la matrice). La description des colonnes est donnée ci-dessous. Les colonnes impaires (1, 3, 5, 7) représentent une **analyse conceptuelle générique**, selon des tables de références que nous définissons (cf. Tab. 9, 11, 12, 13). Les colonnes paires (2, 4, 6, 8) sont spécifiques à la cible considérée, car elles regroupent des **informations d'implémentation**.

0. Exigence de sûreté d'origine.

1. **Type de défaillance logicielle :** Les exigences industrielles sont décomposées en défaillances élémentaires du logiciel sur les flots de données et de contrôle, selon la table de référence 9. Le procédé est décrit en section III.3.1.

2. Assertion exécutable : Les assertions exécutables sont à la fois le produit de la phase d'analyse (cf. Section III.3) et le point de départ de la conception du logiciel de défense.

3. **Stratégie de détection d'erreur :** La stratégie consiste à exprimer que si une assertion n'est pas vérifiée, il y a erreur. Une table générique de stratégie de détection d'erreur correspondant à chaque type de défaillance de référence est donnée au chapitre suivant (cf. Tab.11, Chap. IV).

4. Instrumentation pour la détection d'erreur (sondes logicielles) : Il s'agit d'identifier les services utilisables ou manquants du logiciel fonctionnel permettant de capturer les informations nécessaires à la stratégie de détection d'erreur.

5. **Stratégie de recouvrement d'erreurs :** Une table générique de stratégie de recouvrement d'erreur correspondant à chaque type de défaillance de référence est donnée au chapitre suivant (cf. Tab.12). Les stratégies ne sont pas uniques. Le concepteur doit donc faire un choix en tenant compte des modes dégradés exigés.

6. Instrumentation pour le recouvrement d'erreur (actionneurs logiciels) : Il s'agit d'identifier les services utilisables ou manquants du logiciel fonctionnel permettant d'agir sur l'état du système, pour effectuer le recouvrement d'erreur souhaité.

7. **Stratégie de vérification :** La stratégie consiste à provoquer une erreur telle que les assertions ne soient pas vérifiée, de manière à stimuler les mécanismes de tolérance aux fautes. Une table générique de stratégie de vérification d'erreur correspondant à chaque type de défaillance de référence est donnée au chapitre VI (cf. Tab.13).

8. Instrumentation pour l'injection de fautes : Il s'agit d'identifier les services utilisables ou manquants du logiciel fonctionnel permettant de perturber les assertions.

Par rapport à l'architecture de tolérance aux fautes (cf. Fig.9), le logiciel de défense implémente sous forme d'algorithmes les colonnes de stratégies 2, 3 et 5. L'instrumentation (empreinte réflexive) entre les logiciels fonctionnel et de défense est déterminée dans les colonnes 4 et 6.

Par rapport aux étapes de la méthodologie suivie (cf. Fig.10), les colonnes 0, 1 et 2 font partie de la phase d'analyse. Les colonnes 3 à 6 définissent la phase de conception du logiciel du logiciel de défense et de l'empreinte réflexive. Les colonnes 7 et 8 concernent la validation.

Notons que lorsque la colonne 1 est définie, grâce aux tableaux de référence, les colonnes 3, 5 et 7 sont aussi automatiquement déterminées.

0- Exigence	**1- Type de défaillance logicielle**	**2-** Assertion exécutable	**3- Stratégie de détection d'erreurs**	**4-** Sondes logicielles	**5- Stratégies de recouvrement d'erreurs**	**6-** Actionneurs logiciels	**7- Stratégie de vérification**	**8-** Instrumentation pour injection de fautes
E1	**D1**	**A1**						
E2	**D2**	**A2**						
E3	**D3**	**A3**						
…	**…**	**…**						

Table 10 : Structure de la matrice de traçabilité

Conclusion du Chapitre III

Nous adoptons une approche descendante d'analyse des exigences de sûreté industrielles, pour en déduire des assertions exécutables.

Les exigences de sûreté au niveau applicatif et fonctionnel peuvent être classées suivant le type de défaillances qu'elles cherchent à éviter. Les défaillances sur la valeur d'une donnée, le temps d'échange d'une donnée, une transition, une séquence d'exécution et un temps d'exécution, sont les 5 catégories élémentaires de référence, qui structurent notre analyse.

Une fois qu'un ou plusieurs types de défaillances élémentaires ont été identifiés pour une exigence industrielle donnée, une déclinaison au niveau de l'architecture logicielle précise les éléments de l'implémentation susceptibles de conduire aux défaillances considérées. Selon l'exigence de sûreté, l'assertion exécutable est soit l'expression du comportement des éléments de l'architecture amenant aux défaillances considérées, soit l'expression du comportement des éléments de l'architecture assurant un fonctionnement correct de l'application.

Comme cette étape d'analyse est à la fois flexible (plusieurs interprétations sont possibles pour une exigence donnée) et déterminante pour la conception du logiciel de défense, nous proposons de suivre rigoureusement les étapes de développement du logiciel de défense, à l'aide d'une matrice de traçabilité de chaque exigence de sûreté. Elle permet de conserver un historique des choix du concepteur, depuis les exigences de sûreté jusqu'à la phase de vérification des mécanismes de tolérance aux fautes, vis-à-vis des exigences.

Chapitre IV

Architecture du logiciel de défense

Architecture du logiciel de défense

L'étape d'analyse des exigences de sûreté est supposée avoir été réalisée. Les assertions exécutables, qui définissent les objectifs spécifiques de robustesse du système étudié, sont donc définies. Ce chapitre vise à définir comment concevoir les mécanismes réflexifs pour un jeu d'assertions exécutables donné.

Les mécanismes de tolérance aux fautes peuvent être divisés en mécanismes de détection d'erreurs et de recouvrement d'erreurs. Ils sont implémentés par un logiciel de défense externe au logiciel fonctionnel, et de l'interface d'instrumentation, dite empreinte réflexive.

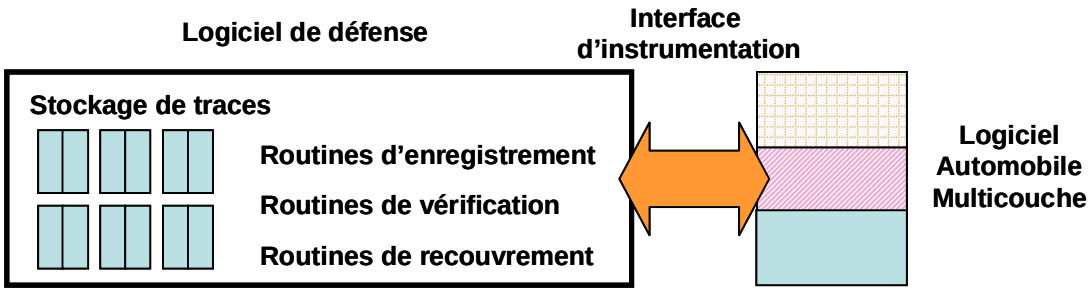


Figure 13 : Organisation du logiciel de défense

La stratégie globale du logiciel de défense [Lu & al. 2009 RTNS] consiste à se référer, à des moments prédéfinis, à des traces d'exécution fiables, pour détecter des erreurs et commander des corrections. En fonction des problèmes surveillés, la détection peut être faite au plus tôt si nécessaire, et le recouvrement est capable d'assurer de la continuité de service dans le meilleur des cas. Le logiciel de défense (cf. Fig.13) est organisé en tables de stockage et trois types de services qui contrôlent : la récupération d'information, la détection d'erreurs, et le recouvrement d'erreurs. Le rôle de l'instrumentation est double : le premier est de capturer des informations nécessaires à l'évaluation des assertions ; le deuxième est de mettre en œuvre du recouvrement pour rétablir un fonctionnement nominal ou dégradé. De plus l'instrumentation et plus particulièrement l'interception d'évènements permet de synchroniser la vérification de l'assertion.

Comme la détection et le recouvrement d'erreur du logiciel de défense sont des traitements séquentiels et indépendants, nous les décrivons séparément. Enfin, nous terminons par l'instrumentation.

IV.1 Logiciel de défense : Détection d'erreurs (Etape 2a)

Des informations sont récupérées au cours de l'exécution et stockées idéalement dans une zone mémoire protégée. La détection d'erreurs repose sur des assertions exécutables prédéfinies. Il s'agit d'évaluer les assertions, en interprétant les informations stockées, à des moments bien choisis. En particulier, lors de la vérification d'une assertion, les informations requises sont sensées être disponibles. Une information requise et absente peut alors signifier une erreur.

La Table 11 détaille selon chaque type de défaillance logicielle (cf. III.3), la stratégie de détection d'erreur du logiciel de défense. Elle est considérée comme une table de référence pour remplir la colonne « 3- stratégie de détection d'erreur » de la matrice de traçabilité (cf. III.4). Pour chaque assertion exécutable définie par analyse, la stratégie de détection d'erreurs est composée de 3 parties :

- **L'assertion.** Les défaillances logicielles ont été classées en 5 catégories (cf. III.3) de cibles : les valeurs de données échangées, les temps d'échange de données, les déclenchements de transition, les séquences d'exécution, les temps d'exécution. Les assertions expriment des **conditions** sur ces cibles logicielles. La complexité des conditions est variable. Elles peuvent consister en de simples tests mathématiques ou logiques (ex. fréquence du signal > 50Hz), ou encore en des expressions logiques sur des données et/ou des évènements de contrôle (ex. si évènement A ou évènement B et donnée a=3 et donnée b=10 et temps de réception donnée b<10ms et séquence {F1, F2, F3} alors donnée c<100).

- **L'information nécessaire à l'évaluation de l'assertion.** Pour évaluer les conditions, il faut tracer tous les éléments (**donnée** et/ou **évènement de contrôle** et/ou **algorithme de calcul**) qui interviennent. Le détail des traces est étudié en section IV.1.2. Des ***routines d'enregistrement*** dans le logiciel de défense sont dédiées à l'utilisation de l'instrumentation logicielle (cf. IV.2) pour récupérer les informations et les écrire dans des structures de stockage.

- **L'évaluation de l'assertion et la notification d'erreur.** A l'implémentation, des ***routines de vérification*** dans le logiciel de défense évaluent l'état courant du système par rapport aux assertions et aux informations stockées. Le moment d'activation de la routine de vérification dépend à la fois de la propriété de sûreté et du besoin ou non de mettre en œuvre pour le système une détection d'erreur « au plus tôt ». Dans le cas où l'assertion exprime une condition interdite, une erreur est signalée lorsque l'assertion est vérifiée. Inversement, si l'assertion exprime une condition d'exécution nominale, une erreur est signalée lorsque l'assertion n'est pas vérifiée. Cependant, une exécution anormale n'est pas toujours dangereuse. Un filtrage des erreurs doit alors être envisagé.

1- Type de Défaillance spécifique	3- Stratégie de Détection d'erreur
Flot de données : Valeur échangée invalide (/non voulue)	L'assertion exprime les conditions dans lesquelles la valeur d'une donnée est interdite (/nominale). Les éléments de la **condition** et la **donnée** considérée sont donc tracés. Si l'assertion est vérifiée (/n'est pas vérifiée), une erreur est notifiée.
Flot de données : Temps d'échange de donnée trop long (/ trop court)	L'assertion exprime les conditions dans lesquelles le temps d'échange d'une donnée est interdit. Les éléments de la **condition** et la **donnée** considérée sont donc tracés. Si l'assertion est vérifiée, une erreur est notifiée.
Flot de contrôle : Transition invalide (/ non voulue)	L'assertion exprime les conditions dans lesquelles un évènement de contrôle ne doit pas (/doit) se produire. Les éléments de la **condition** et **l'évènement de contrôle** considéré sont donc tracés. Si l'assertion est vérifiée (/n'est pas vérifiée), une erreur est notifiée.
Flot de contrôle : Séquence d'exécution invalide (/ non voulue)	L'assertion exprime les conditions dans lesquelles une séquence d'algorithmes de calcul ou d'évènements de contrôle sont interdites (/nominale). Les éléments de la **condition** et **l'algorithme de calcul** ou **évènement de contrôle** considéré sont donc tracés. Si l'assertion est vérifiée (/n'est pas vérifiée), une erreur est notifiée.
Flot de contrôle : Temps d'exécution trop long (/ trop court)	L'assertion exprime les conditions dans lesquelles le temps d'exécution d'un algorithme de calcul est interdit. Les éléments de la **condition** et **l'algorithme de calcul** considéré sont donc tracés. Si l'assertion est vérifiée, une erreur est notifiée.

Table 11 : Stratégie de détection d'erreurs

IV.2 Logiciel de défense : Recouvrement d'erreurs (Etape 2b)

La correction d'erreur consiste idéalement à assurer la continuité de service, si les informations stockées (donnée correcte à échanger, ordre d'exécution attendu, etc.) et l'instrumentation permet de rétablir le bon fonctionnement de l'application, en respectant ses contraintes temporelles. Faute de mieux, la correction peut simplement mettre le système en mode dégradé.

Les modes dégradés des applications automobiles sont généralement très sophistiquées au niveau applicatif et système. Une fois qu'une erreur est détectée, l'application est mise dans un état sûr. Les erreurs sur les données amènent à utiliser des valeurs dégradées prédéfinies. Une contrainte temporelle non respectée ou une notification d'erreur de transmission d'une donnée peut être réglée par un renouvellement de la transmission ou encore une fois l'utilisation de données dégradées. Les actions de recouvrement concernant le flot de contrôle sont limitées. Il consiste au mieux à commander un changement de mode de fonctionnement ou inhiber des fonctions applicatives.

Au niveau du logiciel de base, les actions de recouvrement sur le flot de contrôle sont basiques : terminaison et réinitialisation de taches ou d'un ensemble d'objets OS. Bien que les capacités des services de l'exécutif à modifier le flot d'exécution soient grandes, il est difficile de définir un recouvrement générique, sans tenir compte de la connaissance de l'application. Décider d'arrêter et de relancer un module de la climatisation, de l'airbag ou du contrôle de couple moteur n'a pas le même impact. D'un point de vue applicatif, le support d'exécution n'est pas supposé prendre seul de telles décisions, qui pourraient mettre le véhicule dans un état non sûr.

La Table 12 détaille selon chaque type de défaillance logicielle (cf. III.3), la stratégie de recouvrement d'erreur du logiciel de défense. Elle est considérée comme une table de référence pour remplir la colonne « 5- stratégie de recouvrement d'erreur » de la matrice de traçabilité (cf. III.4). Pour chaque assertion exécutable définie par analyse, la stratégie de détection d'erreurs est décrite en 3 parties :

- **L'erreur notifiée.** Les *routines de vérification* déclenchent les *routines de recouvrement* une fois une erreur détectée (et consolidée).

- **L'utilisation de l'information stockée.** Selon le type de recouvrement (par reprise, poursuite ou compensation), qui dépend du choix du concepteur, l'information utilisée par les *routines de recouvrement* est différente. Pour le recouvrement par reprise, les traces d'exécution permettent de réutiliser les informations d'états précédents et surs du système. Pour le recouvrement par poursuite, les modes dégradés et les valeurs par défaut sont prédéfinis statiquement, et appliqués. Pour le recouvrement par compensation, le comportement attendu de l'application est prédéfini et appliqué.

- **La commande des services de recouvrement.** Dans tous les cas de recouvrement, le flot de donnée et/ou le flot de contrôle du système doit être modifié. Les *routines de recouvrement* déclenchent des services de recouvrement (actionneurs logiciels), présentés dans la section IV.3.3.

1- Type de Défaillance spécifique	5- Stratégie de Recouvrement d'erreur (indicatif)
Flot de données : Valeur échangée invalide /non voulue	Erreur notifiée : la valeur de la donnée est erronée. Le type de recouvrement choisi peut être par reprise, poursuite ou compensation : annuler la valeur échangée, transmettre une valeur particulière (ancienne, par défaut, ou correcte)
Flot de données : Temps d'échange de donnée trop long / trop court	Erreur notifiée : le temps d'échange de la donnée est erronée. Le type de recouvrement choisi peut être par reprise, poursuite ou compensation : si le temps d'échange est trop long, mettre à jour la donnée par un renouvellement de la communication ; si le temps d'échange est trop court, filtrer et mettre en file d'attente la donnée.
Flot de contrôle : Transition invalide / non voulue	Erreur notifiée : La requête de transition est erronée. Recouvrement par compensation : Inhiber la transition en cours. Erreur notifiée : La requête de transition est manquante. Recouvrement par compensation : Déclencher la transition.
Flot de contrôle : Séquence d'exécution invalide / non voulue	Erreur notifiée : La séquence d'exécution est erronée. Interrompre la séquence en cours. Recouvrement par reprise ou compensation : Rétablir la bonne séquence. Recouvrement par poursuite : Déclencher une séquence dégradée
Flot de contrôle : Temps d'exécution trop long / trop court	Erreur notifiée : Le temps d'exécution d'un algorithme de calcul est erronée. Interrompre l'algorithme de calcul en cours. Recouvrement par poursuite : Déclencher un mode dégradée

Table 12 : Stratégie de recouvrement d'erreurs

IV.3 Logiciel de défense : Traces d'exécution

Les mécanismes de stockage et de traçage sont utilisés en général pour les activités de débogage et de diagnostic. La complétude de l'information tracée dépend des objectifs de l'utilisateur : analyse de défauts avec possibilité de reproduire les scenarii de défaillance (traçage étendu), analyse de défauts seulement pour éliminer une erreur particulière (traçage local), profilage de performance pour déterminer où le système passe du temps à l'exécution (traçage sélectif), etc.

La capture des traces logicielles introduit un ralentissement d'exécution significatif. Dans un calculateur, les applications automobiles classiques (e.g. Body Control Module) peuvent échanger plusieurs milliers de données, et être contrôlées par plusieurs douzaines de tâches applicatives ou infrastructurelles. Réduire la dégradation des performances temporelles demande soit de sacrifier des détails, soit utiliser des extensions matérielles.

En conséquence, seule l'information critique nécessaire et suffisante est récupérée à l'exécution, en fonction des problèmes de tolérance aux fautes considérés. La structure de stockage est un facteur majeur pour réduire le temps d'accès à l'information. De plus, l'architecture logicielle pour le stockage doit être conçue de manière à favoriser la réutilisabilité, l'adaptabilité pour traiter la diversité des applications automobiles, et la portabilité sur des plateformes différentes.

L'archivage de l'information est décomposé en plusieurs traces courtes pour réduire leur temps d'accès en lecture par les routines de détection d'erreur, et en écriture par les routines de stockage d'information ou de recouvrement d'erreur. Ces traces sont mises à jour et utilisées à l'exécution. Concrètement, chaque occurrence d'évènement de flots d'information critique est enregistrée, et les évènements de même nature sont stockés dans la même trace. Chaque trace est potentiellement utilisable par plusieurs routines de vérification d'assertions. Le nombre de traces dépend du nombre et de la complexité des assertions à vérifier. Il est intéressant de noter que chaque trace comporte des informations symétriques, car les types d'évènements peuvent être regroupés par paire : début-fin, activation-désactivation, lecture-écriture, etc. Par exemple, l'information de début de tâche et celle de fin de tache doivent être enregistrées dans la trace d'exécution des tâches critiques.

D'après le tableau 6, les types d'éléments à tracer sont les données, les évènements de contrôle et les algorithmes de calcul. Pour ces derniers, nous distinguons les algorithmes de niveau exécutif, c'est-à-dire les tâches, et les algorithmes de niveau application, à savoir les fonctions applicatives. Nous classons donc les traces selon 4 catégories, définies ci-après.

- **La trace des algorithmes de calcul (tâche OS) :** quand une tâche critique démarre son exécution, l'évènement est repéré par l'identifiant de tâche et une estampille temporelle. Le même type d'information est enregistré, quand la tâche se termine (non quand elle est préemptée). La longueur de la trace dépend de la complexité des propriétés de sûreté. Par exemple, si une séquence périodique de l'exécution doit être vérifiée, la longueur est celle de la séquence.

- **La trace des algorithmes de calcul (fonction applicative) :** quand une fonction critique de niveau application démarre, l'évènement est repéré par l'identifiant de la fonction, de la tâche et une estampille temporelle. Le même type d'information est enregistré, quand la fonction se termine (non quand elle est préemptée). Comme précédemment, la longueur de la trace dépend de propriété de sûreté.

- **La trace des évènements de contrôle :** quand un évènement d'activation (qui impacte directement ou indirectement l'activation de la tâche) est émis, l'évènement est repéré par les paramètres qui caractérise l'évènement, l'identification de la tâche courante, et une estampille temporelle. Le même type d'information est enregistré, quand les évènements sont traités.

- **La trace des données :** quand une donnée critique est écrite, l'évènement est repéré par la donnée, l'identifiant de la fonction qui produit la donnée, l'identifiant de la tâche dans laquelle elle tourne et une estampille temporelle (date d'observation et de traçage). Le même type d'information est enregistré, quand la donnée est lue.

IV.4 Instrumentation (Etape 2c)

L'instrumentation est le lien entre le logiciel de défense et le logiciel fonctionnel. Elle fournit au logiciel de défense des services qui permettent de récupérer (*observateurs logiciels*) les informations nécessaires à la détection d'erreurs, et d'agir (*actionneurs logiciels*) sur le logiciel fonctionnel en fonction des commandes du logiciel de défense. Par ailleurs, pour intervenir dans le flot d'exécution et déclencher la détection ou le recouvrement d'erreurs, l'instrumentation est également composée de points d'arrêt (*hooks*) insérés dans le code source.

Chaque élément d'instrumentation peut être potentiellement utilisé par différentes stratégies de détection et de recouvrement d'erreurs du logiciel de défense. Dans ce point de vue, l'instrumentation est générique, flexible et réutilisable. D'un autre côté, la quantité, la localisation, et le moment d'utilisation des éléments d'instrumentation sont spécifiques aux exigences de sûretés (assertions exécutables) considérées pour une application donnée.

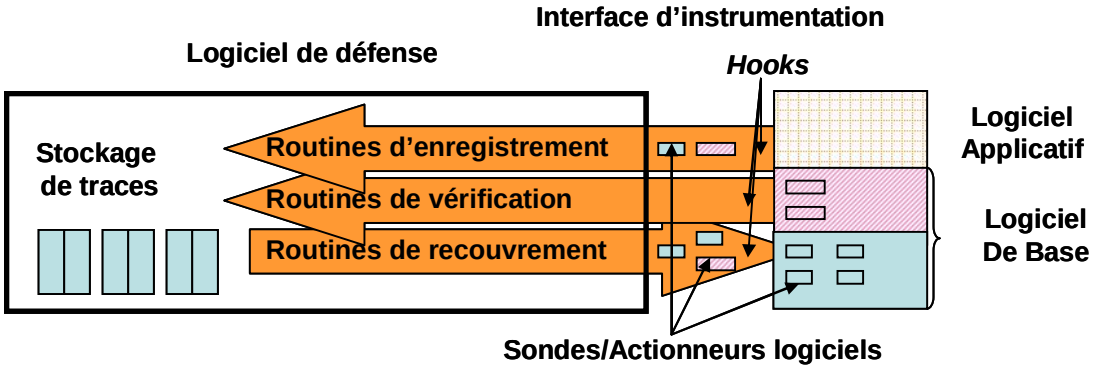


Figure 14 : Organisation de l'instrumentation

IV.4.1 Hooks

En langage C, les *hooks* sont des points d'entrée, avec des routines vides, situées à des endroits sélectionnés du programme. Ils sont couramment utilisés comme points d'arrêt pour le débogage ou pour déclencher des traitements d'exception.

Pour certains exécutifs temps réels comme OSEK et AUTOSAR OS, certaines routines de *hooks* sont définies et implémentées par l'utilisateur. Le système d'exploitation invoque les *hooks* à des moments prédéfini, tels les changements de contexte de tâches, le démarrage et l'arrêt de l'OS, ou encore lorsque l'OS détecte une erreur.

Ces *hooks* sont utilisés pour appeler les routines de stockage, de vérification et de recouvrement du logiciel de défense. L'insertion d'un *hook*, à un endroit choisi du code source, représente l'introduction d'un capteur logiciel à une place particulière de l'architecture logicielle. En effet, la portion de code, où le *hook* est ajouté, appartient à une des couches d'abstraction logicielle. D'autre part, nous cherchons à accrocher les *hooks* aux flots de contrôle et de données critiques. Ainsi, un *hook* représente également le choix d'un moment de l'exécution, pour faire de la capture d'information, de la vérification d'assertion ou du recouvrement d'erreur.

Concrètement, l'identification des différents types de traces (IV.1 .1) que nous avons besoin d'enregistrer, détermine les *hooks* qui servent pour déclencher les routines de stockage d'information. Concernant les traces de flot d'exécution du point de vue OS ou applicatif, les *hooks* doivent être positionnés au début et à la fin de l'exécution des taches critiques ou des fonctions applicatives critiques. Pour les traces de flot de contrôle, si des appels systèmes explicites sont utilisés, les *hooks* peuvent être positionnés avant ou après ces appels systèmes en question. Si des objets de contrôle (ex : tâche, évènement, etc.) sont à surveiller, avec des appels systèmes implicites ou non disponibles à l'intégrateur, le problème se traite au cas par cas en fonction de l'implémentation étudiée. Enfin pour les traces de flot de données, les *hooks* peuvent être positionnés avant ou après l'utilisation de services de production et de consommation de données.

L'instant d'exécution du déclenchement de la routine de vérification, et donc l'endroit où le *hook* est placé est crucial. Pour cela, les *hooks* précédents appelant des routines de stockage peuvent être réutilisés, ou de nouveaux peuvent être introduits au cas par cas.

Les routines de recouvrement sont en général déclenchées par les routines de vérification lorsqu'il y a une erreur détectée. Une autre solution dépendant de l'assertion exécutable étudiée, consiste à retarder le recouvrement à la fin de l'exécution d'une tâche par exemple. Dans ce cas, un *hook* à la fin de la tache considérée contient la routine de recouvrement, qui n'est activée que si la routine de détection d'erreur correspondante l'a bien décidé.

Plusieurs types d'implémentation des *hooks* peuvent être considérés. Pour les appels à des services exécutifs ou de communication, les *hooks* peuvent être insérés juste avant ou après l'instruction. Sinon, ils peuvent être ajoutés à l'intérieur de ces services, au début ou à la fin de leur programme. Le choix d'une solution par rapport à l'autre est important pour les appels systèmes, si le système supporte la séparation

entre les modes utilisateur et superviseur. Dans le premier cas, la routine du *hook* s'exécute en mode utilisateur, tandis que dans le second cas, elle s'exécute en mode noyau et possède d'avantage de droits d'accès aux informations (priorité des tâches, etc.) si besoin.

Enfin une autre façon de différencier les *hooks* est l'utilisation de paramètres d'entrée ou non à l'interface du *hook*. C'est une alternative à l'utilisation de services de récupération de données (observateurs logiciels). Par exemple, après un service d'écriture de donnée, la valeur de la donnée peut être récupérée comme paramètre d'entrée d'un *hook*. La routine de stockage de la donnée appelée par ce *hook* récupère alors le paramètre au lieu d'appeler un service de lecture de la donnée.

IV.4.2 Services de base pour la capture d'informations

Les services de capture d'information sont utilisés par les routines de stockage du logiciel de défense. Les types d'éléments récupérés et stockés sont définis par la stratégie de traçage (IV.3). En rangeant ces éléments par nature et non par type de trace, les catégories qui se dégagent sont :

- Les **identifiants** (de fonction applicative, de tâche ou de routine d'interruption, du mode de fonctionnement courant, de l'évènement de contrôle considéré, etc.) ;
- Les **paramètres** (données, paramètres de retour, etc.) ;
- Les **estampilles temporelles**.

Pour capturer ces différents types d'information, quelques exemples de services standardisés AUTOSAR sont donnés ci-après.

Capture des identifiants. Si l'identifiant recherché n'est pas implémenté au préalable dans la cible étudiée, il doit être ajouté. En l'occurrence, au niveau applicatif, le marquage des fonctions par des identifiants est utile pour surveiller, par exemple, une séquence d'exécution critique de fonctions applicatives. Dans le standard AUTOSAR, ces identifiants ne sont pas spécifiés.

Si l'identifiant recherché existe, deux cas sont possibles : un service de lecture de l'identifiant est implémenté ou non. Ces services de base servent en général à consulter l'état du système, pour du débogage, en phase de développement. L'utilisation de ces services est la technique la moins intrusive, pour récupérer l'information. Autrement, il s'agit d'accéder à la variable correspondant à l'identifiant. Ceci est possible, si le code source est à disposition et maîtrisé, ou encore si la localisation de la variable dans la mémoire est connue. Par exemple, via l'interface d'un exécutif temps-réel conforme au standard OSEK-VDX, la plupart des informations concernant le contexte d'exécution courant sont accessibles. Le service *GetTaskID* donne l'identifiant de la tâche qui est en train de s'exécuter. L'état courant du masque des évènements reçus par une tâche en attente d'évènements peut être obtenu avec *GetEvent*. AUTOSAR OS, comme extension d'OSEK, possède des interfaces d'observation additionelles, parmi lesquelles *GetISRID*. *GetISRID* permet de récupérer l'identifiant des routines d'interruptions. L'interface d'observation d'AUTOSAR OS est relativement riche. Elle nous convient pour récupérer les informations concernant le contexte d'exécution courant, du point de vue de l'OS.

Capture des paramètres. De même que pour la capture des identifiants, les paramètres doivent être disponibles. Des services de lecture des paramètres peuvent exister. Par exemple, dans OSEK OS, *GetAlarm* retourne la valeur relative en unité de temps avant que l'alarme expire. *GetTaskState* renseigne l'état d'une tache (i.e. en cours d'exécution, prête à s'exécuter, en attente d'évènement, ou suspendue). Si le code source est accessible, la routine de stockage et le *hook*, dans lequel la routine s'exécute, peuvent définir des paramètres d'entrée, pour pouvoir récupérer les informations directement. En particulier, cette technique est pratique pour enregistrer les données échangées. Enfin, si la localisation de la variable dans la mémoire est connue, un accès direct à la mémoire évite d'introduire des paramètres aux *hooks*. Cette solution peut être appliquée, lorsque les *hooks* utilisés sont standardisés et ne comportent pas de paramètres d'entrée.

Capture des estampilles temporelles. L'estampille temporelle dépend de la précision attendue pour les mesures de temps d'exécution. Le compteur de référence peut donc être l'horloge, une alarme, un compteur matériel, etc. L'exécutif temps réel peut offrir des services pour récupérer la valeur des compteurs. Avec un exécutif OSEK, nous utilisons par exemple une alarme de référence et le service *GetAlarm*. Dans ce cas une alarme est définie par un cycle qui évalué en unités de temps (*ticks*). Chaque appel de *GetAlarm* donne le nombre d'unités de temps qui reste avant la fin du cycle en cours. La précision est alors d'une unité de temps.

IV.4.3 Services de base pour le recouvrement

Une fois une erreur détectée, les routines de recouvrement sont appelées. En fonction de la stratégie de recouvrement choisie lors des exigences de sûreté du système. Les services de base pour le recouvrement peuvent être présentés selon les erreurs sur les flots de contrôle et de données critiques. Les services du logiciel exécutif, qui permettent de contrôler l'exécution des applications, peuvent être utilisés également pour corriger le flot d'exécution. Pour le flot de données, les services fonctionnels de communication (lecture, écriture de données) peuvent aussi servir d'actionneurs logiciels pour le recouvrement.

Recouvrement du flot de contrôle. Les actions de recouvrement principales concernent le cycle de vie des taches, au niveau de l'OS :

- Terminaison de l'exécution d'une tâche : l'idée est de terminer un traitement en cours erroné.

- Activation d'une tâche : l'objectif est par exemple de déclencher une tâche dégradée si une mise en mode dégradée est décidée ; ou de lancer la tâche attendue dans le cas ou une mauvaise séquence d'exécution a été détectée ; ou encore de relancer la même tâche pour ré-exécuter le même traitement avec des données d'entrée correctes. L'activation de la tâche voulue peut être transitoire ou périodique.

- Suspension de l'exécution d'une tâche : l'idée est d'arrêter temporairement l'exécution pour permettre celle d'autres tâches.

Recouvrement du flot de données. Au niveau de la communication de données, les actions de recouvrement consistent à rétablir l'échange de données en temps et en valeur :

- Production de données correctes ou dégradée : la stratégie de recouvrement écrase la donnée erronée précédente par une bonne valeur.

- Consommation de données correctes ou dégradées : l'instruction de consommation est appelée une nouvelle fois pour récupérer la donnée correcte mise à jour par la stratégie de recouvrement.

- Renouvellement de requête à la donnée : l'instruction de production ou de consommation est appelée une nouvelle fois, lorsqu'un dépassement de durée de réception ou non-émission de message a été détecté.

- Inhibition ou retard de donnée : lorsqu'une donnée invalide ou intempestive est reçue, il s'agit de filtrer la donnée, pour ne transmettre que la bonne information à l'application.

Il est intéressant de noter que l'observabilité (existence ou possibilité d'ajout de sondes logicielles) et la contrôlabilité (existence ou possibilité d'ajout d'actionneurs logiciels) de la cible fonctionnelle influencent l'efficacité de la détection et du recouvrement d'erreurs. Un composant logiciel applicatif critique qui ne met pas à disposition ses données internes critiques et/ou qui gère son flot de contrôle critique (transitions, séquences d'exécution) au niveau applicatif, est impossible à maîtriser. En conséquence pour améliorer la tolérance aux fautes d'une application critique, les flots de données et de contrôle doivent être externes à la couche applicative, pour que le logiciel de défense puisse détecter les erreurs et pour que les services de communication et de contrôle d'exécution puissent intervenir.

Conclusion du Chapitre IV

Pour concevoir le logiciel de défense, les stratégies de détection d'erreurs sont composées des assertions, de l'information nécessaire à l'évaluation de l'assertion, de l'évaluation de l'assertion et de la notification d'erreurs.

Les stratégies de recouvrement d'erreurs explicitent l'erreur notifiée, l'utilisation de l'information stockée et la commande des services de recouvrement.

Les traces d'exécution sur lesquelles reposent les stratégies de détection et de recouvrement peuvent être classées en 4 types : les traces d'algorithmes de calcul de niveau OS, les traces d'algorithmes de calcul de niveau applicatif, les traces des évènements de contrôle et les traces des données.

Pour maîtriser les flots de données et de contrôle du système, l'insertion de hooks, tels des points d'arrêt, dans le code est nécessaire, pour effectuer la récupération d'information, la vérification d'assertions ou le recouvrement d'erreurs.

La récupération d'information (sondes logicielles) et les actions de recouvrement d'erreurs (actionneurs logiciels) peuvent être implémentées par des services fonctionnels existants du logiciel de base.

Chapitre V

Etude de cas

Etude de cas

Nous avons mis en œuvre la méthodologie et implémenté l'architecture de tolérance aux fautes, sur plusieurs plateformes logicielles AUTOSAR, à la fois en simulation et sur carte d'évaluation. Ces cas d'études simples ont servi à développer séparément le logiciel de défense pour chaque type d'assertions exécutable identifié. Nous avons étudié les défaillances de séquences d'exécution (flot de contrôle), et de valeurs échangées (flot de données), à partir d'une application de « climatisation » Renault, portée sur une infrastructure logicielle AUTOSAR. Nous avons retrouvé le même type de propriétés sur une application « ABS-ESP ». Nous avons synthétisé, sur le même type de support d'exécution, une application inspirée d'un « airbag », illustrant une condition complexe sur des données, impactant le temps d'exécution d'un traitement (flot de contrôle). Nous avons étudié les défaillances sur le déclenchement de transition (flot de contrôle), à partir d'une application de « mise en veille du véhicule », et une autre de « transmission du couple moteur ».

Ces différents prototypes, réalisés au fil de la thèse, ont montré chacun la faisabilité de la solution de tolérance aux fautes proposée. Cependant, comme la généralisation de l'architecture de tolérance aux fautes s'est construite progressivement, la structure des mécanismes s'est améliorée de prototype en prototype. Le dernier cas d'étude, ciblant les déclenchements de transition, est le plus abouti : d'une part, la méthodologie est déployée jusqu'à l'étape de validation par injection de fautes ; d'autre part un exécutif temps réel gérant les mécanismes de protection mémoire matérielle a été utilisé. Pour toutes les autres applications, la phase de vérification n'a pas été abordée et aucune hypothèse de protection mémoire matérielle n'est prise en compte.

L'inconvénient de ne présenter que le dernier cas d'étude complet serait, malgré tout, qu'il n'illustre qu'une seule propriété de sûreté (assertion). Dans ce chapitre, nous choisissons donc de présenter l'application cible comme un multiplexage de nos applications étudiées indépendamment. Nous déploierons la méthodologie jusqu'à la phase d'implémentation. La vérification sera abordée dans le dernier chapitre, en exploitant les résultats que nous avons obtenus sur une seule propriété de sûreté.

V.1 Cible logicielle AUTOSAR

Le système étudié est embarqué sur un microcontrôleur 16 bits, S12XEP100 de Freescale. Il a la particularité de comporter des mécanismes de protection mémoire matérielle (MPU), que nous utiliserons pour séparer spatialement le logiciel fonctionnel du logiciel de défense.

L'architecture logicielle de notre cible est de type AUTOSAR. Les applications AUTOSAR sont décomposées en composants logiciels qui interagissent entre eux. Ils peuvent s'échanger soit des données, soit des services, via des interfaces. Ils ne peuvent pas appeler directement les services du logiciel de base, c'est à dire l'ordonnanceur, les bus de communication, le matériel, etc. La couche d'abstraction intermédiaire entre les applications et le logiciel de base est appelée AUTOSAR RTE (*Runtime Environment*). Elle contrôle l'interface standardisée de l'OS et utilise les fonctionnalités de l'OS (tâche, ressource, évènement,…), pour fournir ses propres fonctionnalités aux modules applicatifs.

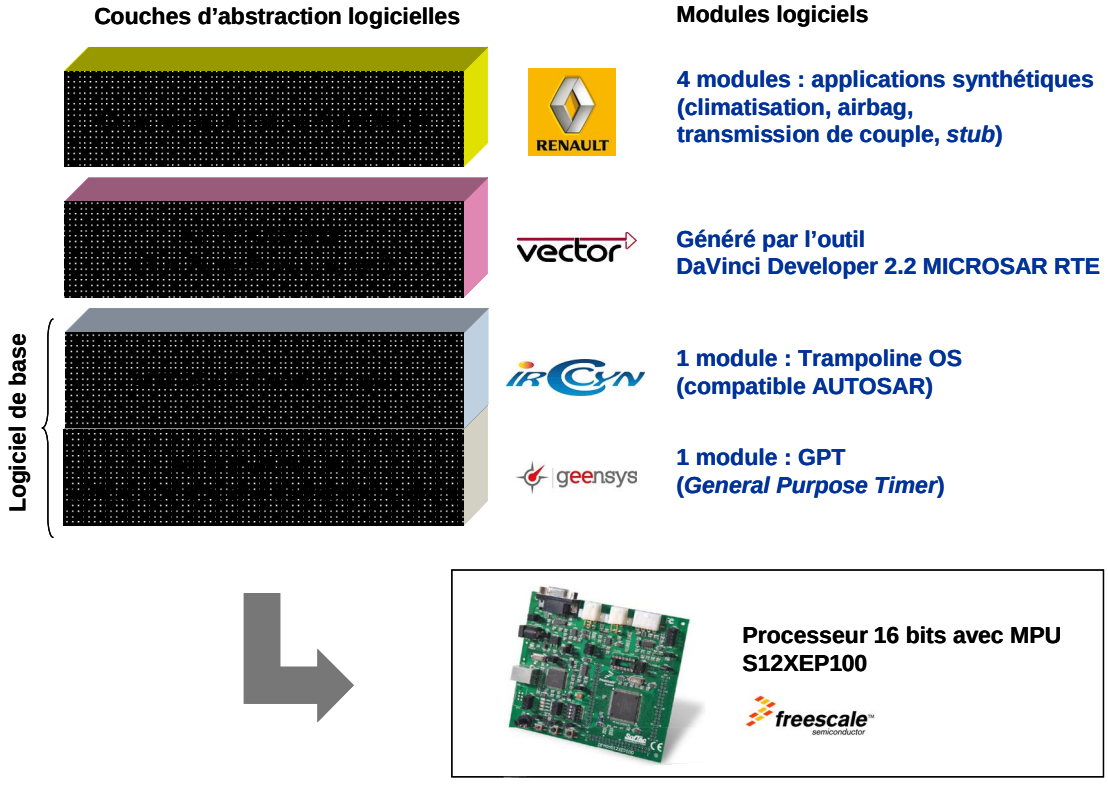


Figure 15 : Architecture de la cible

Notre cible logicielle étudiée (cf. Fig.15) superpose 4 couches. Le logiciel de base comporte ici deux couches d'abstraction : AUTOSAR *Service Layer* et AUTOSAR MCAL *MicroController Abstraction Layer*. La couche d'abstraction du microcontrôleur contient uniquement un module *General Purpose Timer* (GPT), qui gère les *timers* matériels. La couche de services est réduite à un exécutif temps réel, appelé « Trampoline OS» [Bechennec & al. 2006]. C'est un logiciel en source libre, développé par l'Irccyn, à partir des spécifications OSEK OS et AUTOSAR OS.

La couche de communication AUTOSAR RTE est produite automatiquement, à partir d'informations de configuration de la couche applicative et du logiciel de base. Cette génération de code a été réalisée à l'aide de l'outil commercial DaVinci 2.2 MICROSAR RTE ^{TM} de Vector.

La couche applicative regroupe 4 composants logiciels, possédant les interfaces AUTOSAR. Le composant « climatisation » consiste en l'adaptation et le portage d'un module automobile existant. Les composants « airbag » et « transmission de couple » sont synthétiques. En effet, nous avons implémenté la partie de contrôle et de gestion de données qui nous intéressent, le reste étant représenté par des temporisations. Le dernier composant « *stub* » synthétique représente le reste de l'environnement applicatif. Il envoie aux 3 autres composants les données provenant des capteurs et des autres calculateurs, dont ils ont besoin. La cohabitation des fonctions climatisation, airbag et transmission de couple, dans le même calculateur est seulement illustratif et sans doute peu réaliste aujourd'hui.

Nous considérons arbitrairement qu'une partie de la climatisation est critique, puisque nous étudions une exigence de sûreté de la climatisation. Pour les autres applications, nous considérons une exigence de l'airbag et deux de la transmission de couple, que nous supposons toutes sécuritaires. Les parties des applications n'intervenant pas dans les exigences sélectionnées ne sont pas décrites ici. De même le composant logiciel applicatif « *stub* » n'est pas abordé.

L'objectif de l'étude cas est de déployer la méthodologie (cf. Section II.4), pour des types d'assertions différentes, jusqu'à l'implémentation des mécanismes de tolérance aux fautes, en traçant les différentes étapes :

- Analyse des types de défaillances (étape 1a) ;
- Analyse du type d'architecture logicielle (étape 1b) ;
- Conception des stratégies de détection (étape 2a) ;
- Conception des stratégies de recouvrement (étape 2b) ;
- Conception de l'instrumentation (étape 2c) ;
- Implémentation de l'instrumentation (étape 3a) ;
- Implémentation du logiciel de base (étape 3b) ;
- Vérification du logiciel de défense (étape 4a) ;
- Validation du logiciel complet (étape 4b).

La matrice de traçabilité est construite au fur et à mesure des étapes, exigence par exigence. Les exigences, les types de défaillances et les assertions sont numérotées, pour favoriser leur suivi.

V.2 Phase d'analyse

V.2.1 Etape 1a : Analyse des types de défaillances

La climatisation. Cette application est représentative de la complexité de la plupart des applications automobiles actuelles. Elle contient de la régulation, des tables d'interpolation, des automates d'états, et beaucoup d'interaction avec l'extérieur. Nous nous intéressons plus particulièrement à la partie qui détermine le mode de fonctionnement de la climatisation (cf. Fig.16), en fonction de la température extérieure, de plusieurs données sur l'état du moteur et du logiciel de base (mémorisation de variables en EEPROM). Les trois modes de fonctionnement de la climatisation sont « forte », « faible », et « inactive ». Une machine à état principale gère 11 entrées (provenant d'un module de filtrage et un module de commande manuelle) et 6 sorties.

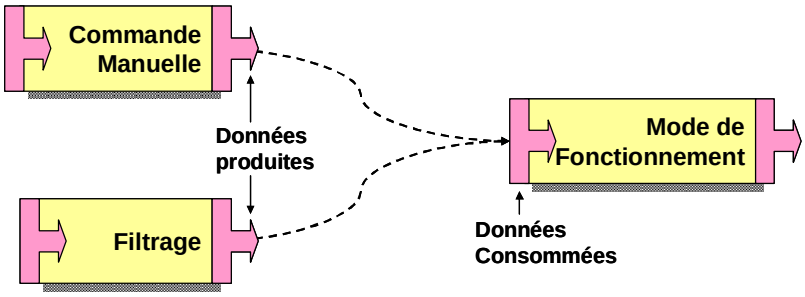


Figure 16 : Schéma fonctionnel des modules « climatisation »

L'exigence de sûreté (**E1**) en question et sa décomposition en types de défaillances, dont le système veut se prémunir, sont données ci-après.

0- Exigence de sûreté	1- Type de Défaillance spécifique
E1 : Le calcul du mode de fonctionnement ne doit être fait que lorsque toutes ses données d'entrée, provenant de la commande manuelle et du filtrage sont disponibles.	**D1a : Flot de données** : Valeur échangée non voulue
	D1b : Flot de contrôle : Séquence d'exécution non voulue

- La défaillance (**D1a**) sur le **flot de données**, concerne soit des valeurs échangées invalides, soit un temps d'échange de donnée trop long (« toutes ses données d'entrée» « sont disponibles»). Les données considérées sont les variables produites par «commande manuelle» et par «filtrage» et consommées par «calcul du mode de fonctionnement».

- La défaillance (**D1b**) sur le **flot de contrôle**, concerne une séquence d'exécution non attendue («commande manuelle», «filtrage», «calcul du mode de fonctionnement»). Les deux fonctions productrices («commande manuelle», «filtrage») doivent être exécutées avant la fonction consommatrice («calcul du mode de fonctionnement»). Mais l'ordre n'est pas imposé entre «commande manuelle» et «filtrage».

L'airbag. Le système doit analyser périodiquement le signal issu du capteur et déclencher dans le temps imparti le gonflage des coussins si nécessaire. La fonction « prise de décision » (cf. Fig.17) constitue le cœur même du système airbag. Des conditions complexes permettent de filtrer les perturbations de bruit électronique, des imperfections du capteur, etc. susceptibles d'impacter la décision et le temps de décision de déclenchement de l'airbag.

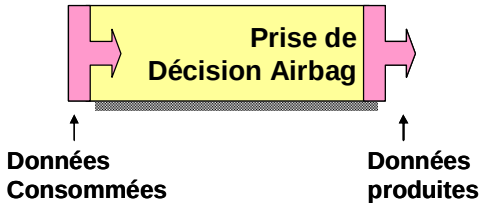


Figure 17 : Schéma fonctionnel du module « airbag »

L'exigence de sûreté (**E2**) en question et sa décomposition en types de défaillances, dont le système veut se prémunir, sont données ci-après.

0- Exigence de sûreté	1- Type de Défaillance spécifique
E2 : L'exécution de la commande de mise à feu de l'airbag après analyse des données de capteur ne doit pas excéder 10ms.	**D2 : Flot de contrôle** : Temps d'exécution trop long

La défaillance (**D2**) porte sur le **flot de contrôle**, concernant un temps d'exécution trop long (« commande de mise à feu ne doit pas excéder 10 ms»). Le déclenchement de la mise à feu porte une condition sur les données (« analyse des données »), qui n'est pas détaillée.

La transmission de couple. Pour un groupe moto-propulseur hybride, un système de contrôle interprète les commandes du conducteur et l'environnement pour basculer d'un mode (moteur thermique) à un autre (moteur électrique). La fonction de transmission (cf. Fig.18) est composée de deux parties de contrôle (statique et dynamique). Le contrôle statique récupère les commandes du conducteur et de l'environnement, puis calcule des valeurs de consigne pour la vitesse du moteur, la puissance de batterie et le couple aux roues. A partir de ces données, le contrôle dynamique calcule les couples du moteur thermique et des moteurs électriques.

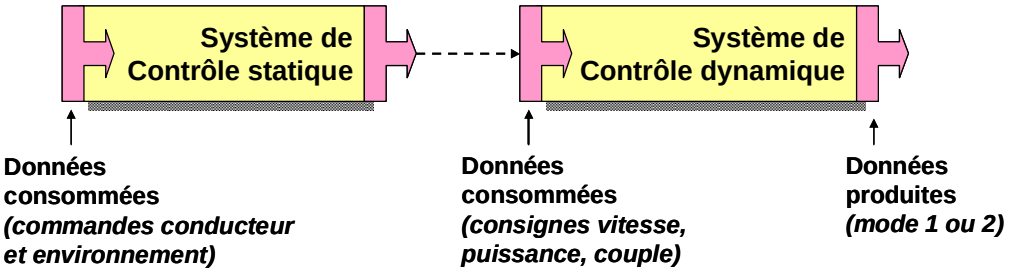


Figure 18 : Schéma fonctionnel des modules « transmission de couple »

La première exigence de sûreté considérée (**E3**) et sa décomposition en types de défaillances, dont le système veut se prémunir, sont données ci-après.

0- Exigence de sûreté	1- Type de Défaillance spécifique
E3 : Le blocage en mode 1 alors que le système doit passer en mode 2 (idem pour mode 2) est une défaillance.	**D3 : Flot de contrôle :** Transition invalide

La défaillance (**D3**) porte sur le **flot de contrôle**, concernant une transition invalide de mode 1 vers le mode 2 (« blocage en mode 1»). La nature de la requête de changement de mode n'est pas précisée, à ce niveau.

La deuxième exigence de sûreté considérée (**E4**) est la suivante :

0- Exigence de sûreté	1- Type de Défaillance spécifique
E4 : Le passage intempestif «du mode 1 au mode 2» ou «du mode 2 au mode 1» est une défaillance.	**D4 : Flot de contrôle :** Transition non voulue

La défaillance (**D4**) porte sur le **flot de contrôle**, concernant une transition non attendue (« passage intempestif»). La nature de la requête de changement de mode n'est pas précisée, à ce niveau.

V.2.2 Etape 1b : Analyse du type d'architecture logicielle

Quelques termes spécifiques au vocabulaire AUTOSAR sont d'abord introduits, pour décrire plus précisément l'implémentation de la plateforme étudiée. L'analyse fine de chaque partie d'application, relative aux exigences de sûreté considérées, permet d'exprimer les assertions exécutables.

V.2.2.1 Notions de base de l'architecture AUTOSAR

Runnable. Le concept de *runnable* a été introduit pour décrire le comportement interne d'un composant logiciel applicatif. Il s'agit aussi de faciliter et orienter l'intégration et le travail d'ordonnancement de l'OS en décomposant le code applicatif de manière à manipuler des entités ordonnançables et indivisibles en termes d'exécution. Un *runnable* encapsule une séquence d'instructions, par exemple une fonction en langage C qui exécute un algorithme simple ou un programme complexe, à l'intérieur d'un composant logiciel. Par rapport à une tâche, chaque *runnable* est susceptible d'être appelé par une tâche de l'OS qui lui est associée. En termes de communication, un *runnable* peut interagir par l'intermédiaire de ports pour écrire ou lire une donnée, ou pour invoquer un service.

Tâche. Une tâche fournit le contexte et les ressources communes pour l'exécution de fonctions. C'est un concept générique qui ne distingue pas les notions de *thread* et *process* qu'on rencontre dans POSIX par exemple.

Ressource. Une ressource peut représenter de l'espace mémoire, des composants matériels, des applications ou encore l'ordonnanceur. L'accès aux ressources est géré par le *Priority Ceiling Protocol*, qui permet d'éviter les inversions de priorité et les blocages. Ainsi, l'ordonnanceur peut être considéré comme une ressource, car il peut être bloqué. Sa priorité devient alors la priorité maximale du système.

Evènement. Un évènement conditionne l'attente d'une tâche. Il est ainsi utilisé comme moyen de synchronisation entre tâches ou pour signaler la réception ou l'envoi de messages.

Alarme. Les alarmes sont affectées statiquement à un compteur et à une tâche, pour cadencer une tâche périodique par exemple. L'alarme se déclenche dès qu'elle atteint une valeur de référence du compteur, qui peut être modifiée en cours d'exécution.

OS-Application. Une OS-Application est une collection d'objets OS (tâches, interruptions, ressources, alarmes, compteurs, tables d'ordonnancement) qui forme une unité fonctionnelle cohérente. L'OS-Application a seulement une existence structurelle (comme un identificateur). A l'exécution, l'OS gère le comportement de ses objets de manière transparente à l'encapsulation par les OS-Application. Cependant, chaque objet OS est caractérisé par son appartenance à une OS-Application. Ainsi, les interactions entre OS-Applications peuvent être contrôlées, dans la mesure où un objet OS peut être autorisé ou non à interagir avec d'autres objets, en fonction de son identifiant d'OS-Application. Les OS-Applications sont considérées comme des mécanismes de protection mémoire. Les « *Trusted OS-Applications* » n'ont aucune restriction d'accès à la mémoire et à l'API de l'OS. Leur comportement temporel ne nécessite pas d'être renforcé à l'exécution. Elles sont autorisées à s'exécuter en mode privilégié s'il est supporté par le processeur. Les « *Non-Trusted OS-Applications* » ont un accès restreint à la mémoire et à l'API de

l'OS. Elles ne sont pas autorisées à s'exécuter en mode privilégié s'il est supporté par le processeur.

Trusted-Function. Une « *trusted-function* » est un service fourni par une « *Trusted OS-Applications* », qui s'exécute en mode superviseur (ou noyau). Elle peut être utilisée par les autres OS-applications, en mode utilisateur ou superviseur.

V.2.2.2 Assertions exécutables

La traduction des exigences de sûreté en assertions exécutables passe par une connaissance de l'implémentation. Dans l'architecture AUTOSAR, les différentes applications considérées se retrouvent dans des composants logiciels, appelés respectivement **SWC_Clim**, **SWC_Air** et **SWC_Trans**, pour les applications climatisation, airbag et transmission de couple (cf. Fig.19).

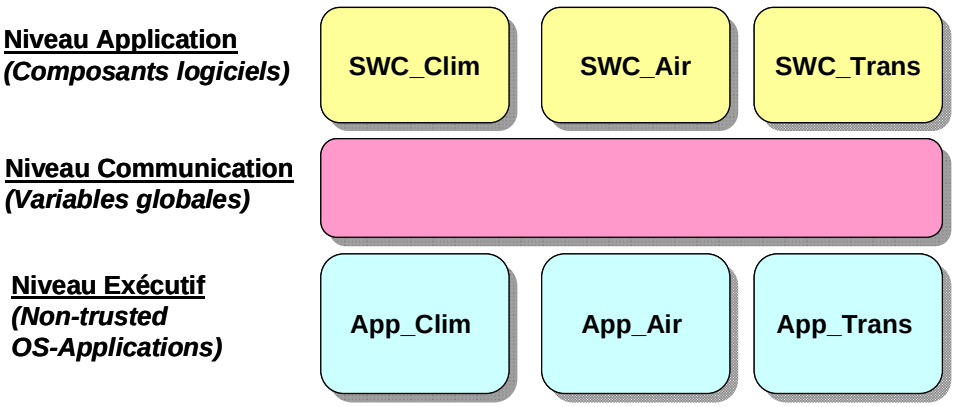


Figure 19 : Schéma architectural de la plateforme logicielle

Comme ces composants sont fonctionnellement indépendants les uns des autres, une séparation de leur espace d'adressage dans la mémoire est judicieuse. Ainsi, au niveau de l'exécutif, les objets OS (tâches, alarmes, ressources) sont regroupés en *OS-Applications*, respectivement `App_Clim`, `App_Air` et `App_Trans`. Comme chacune des applications doit traiter ses données dans la zone mémoire qui lui est affectée et non celle des autres, les *OS-Applications* sont conçues « *non-trusted* ». Revenons maintenant au détail de chaque application et de ses exigences de sûreté.

La climatisation. Repartons de l'exigence **E1 :** « Le calcul du mode de fonctionnement ne doit être fait que lorsque toutes ses données d'entrée, provenant de la commande manuelle et du filtrage sont disponibles ».

Dans le vocabulaire AUTOSAR, « calcul du mode de fonctionnement », « commande manuelle » et « filtrage » sont des fonctions applicatives (*runnables)*, ayant respectivement comme identifiant : `Runnable_Clim_WM`, `Runnable_Clim_MS`, `Runnable_Clim_HF`. Ces *runnables* appartiennent au composant logiciel `SWC_Clim` (cf. Fig.20).

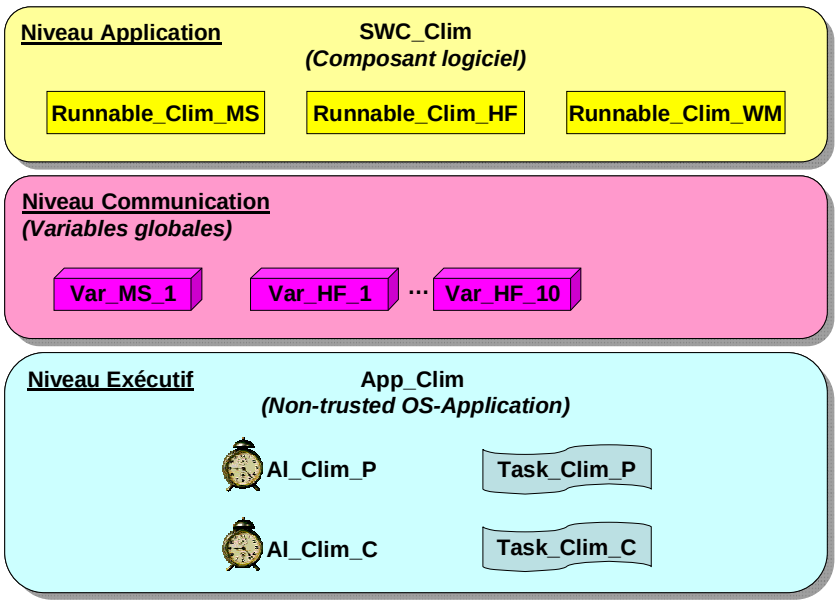


Figure 20 : Schéma architectural de l'application « climatisation »

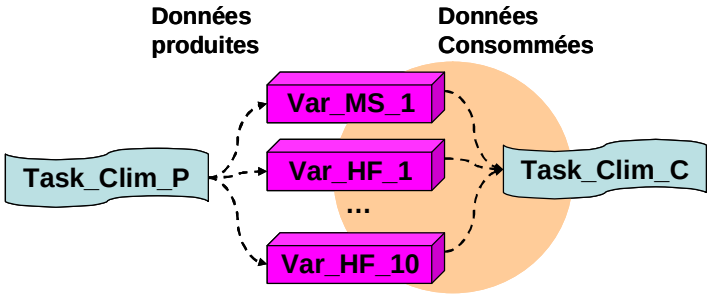
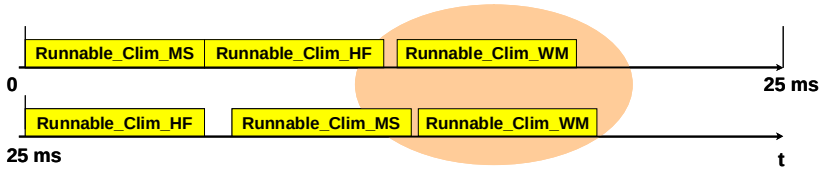
`Runnable_Clim_MS` et `Runnable_Clim_HF` sont implémentés dans la même tâche `Task_Clim_P` périodique à 25 ms, de priorité forte (5). La tâche est réveillée par une alarme `Al_Clim_P` à 25 ms.

`Runnable_Clim_WM` est implémentée dans la tâche `Task_Clim_C` périodique à 25ms, de priorité moins forte (4). La tâche est réveillée par une alarme `Al_Clim_C` à 25ms.

Les objets OS (`Task_Clim_P`, `Al_Clim_P`, `Task_Clim_C`, `Al_Clim_C`) appartiennent à l'OS-Application `App_Clim`, qui est « non-trusted » (en mode utilisateur).

Les 10 données produites par `Runnable_Clim_HF` sont appelées `Var_HF_1` à `Var_HF_10`. La donnée produite par `Runnable_Clim_MS` est appelée `Var_MS_1`. Toutes ces données sont consommées par `Runnable_Clim_WM`.

L'étape (1a) a identifié une défaillance (**D1a**) sur les *valeurs échangées invalides* ou *temps d'échange de donnée trop long* et *séquence d'exécution non attendue.* Avec les informations d'implémentation : avant l'exécution de `Runnable_Clim_WM` (dans la tâche `Task_Clim_C`), les 11 variables doivent avoir été produites. Pour la défaillance (**D1b**) sur une *séquence d'exécution non attendue.* A partir des détails d'implémentation, en termes de séquence, le motif {`Runnable_Clim_HF, Runnable_Clim_MS, Runnable_Clim_WM`} ou {`Runnable_Clim_MS, Runnable_Clim_HF, Runnable_Clim_WM`} doivent être répétées toutes les 25 ms. Les assertions **A1a** et **A1b** correspondant respectivement à **D1a** et **D1b** sont écrites ci-après en pseudo code.

1- Type de Défaillance spécifique	**2- Assertion exécutable**
D1a : Flot de données : Valeur échangée non voulue	 **Assertion A1a (en pseudo-code) :** `Task_Clim_C` = tâche courante ; {`Var_HF_1, Var_HF_2, Var_HF_3, Var_HF_4, Var_HF_5, Var_HF_6, Var_HF_7, Var_HF_8, Var_HF_9, Var_HF_10, Var_MS_1`} produits ;
D1b : Flot de contrôle : Séquence d'exécution non voulue	 **Assertion A1b (en pseudo-code) :** SequenceRef1 = {`Runnable_Clim_HF , Runnable_Clim_MS , Runnable_Clim_WM`}; SequenceRef2 = {`Runnable_Clim_MS , Runnable_Clim_HF , Runnable_Clim_WM`}; Vérification avant chaque runnable appartenant à une séquence : {runnable suit bien son précédent par rapport à SequenceRef1 ou SequenceRef2 ; (temps entre deux runnables de même identifiant de séquences successives) < 50 ms ;}

L'airbag. Repartons de l'exigence **E2** : La commande de mise à feu de l'airbag après analyse des données de capteur ne doit pas excéder 10 ms.

Dans le vocabulaire AUTOSAR, « commande de mise à feu», est un *runnable*, ayant comme identifiant : `Runnable_Air_FC`. Ce runnable appartient au composant logiciel `SWC_Air` (cf. Fig.21).

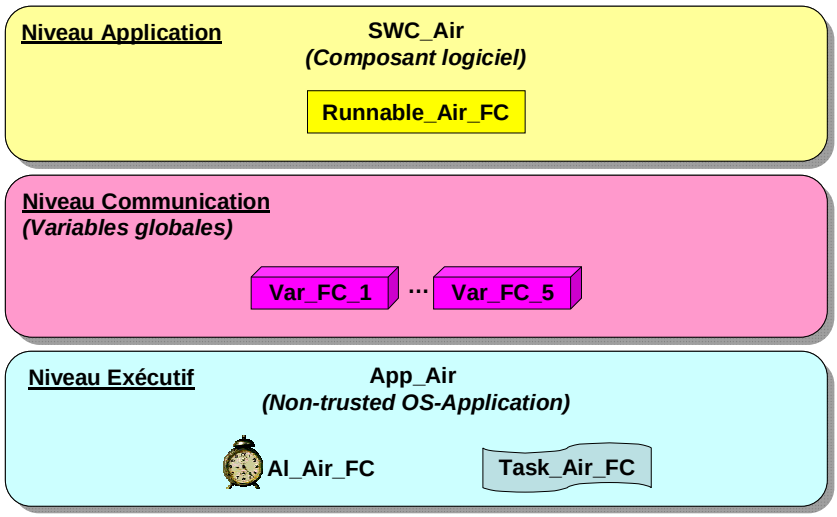
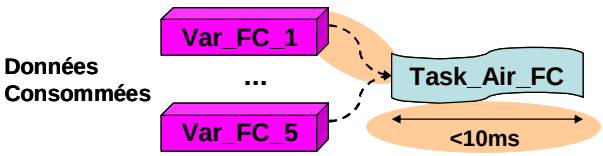


Figure 21 : Schéma architectural de l'application « airbag »

`Runnable_Air_FC` est implémentée dans la tâche `Task_Air_FC` périodique à 1 ms, de priorité forte (6). La tâche est réveillée par une alarme `Al_Air_FC` à 1ms. Les objets OS (`Task_Air_FC`, `Al_Air_FC`) appartiennent à l'OS-Application `App_Air`, qui est « non-trusted » (en mode utilisateur).

Les 5 données consommées par `Runnable_Air_FC` sont appelée `Var_FC_1` à `Var_FC_5`. Ces variables sont liées par une condition mathématique complexe qui déclenche des algorithmes de calculs plus ou moins longs et non observables, à l'intérieur de `Runnable_Air_FC`. Mais si `Var_FC_1` = 3, la durée nominale de `Runnable_Air_FC` est inférieure à 10 ms

L'étape (1a) a identifié une défaillance (**D2**) non attendue sur le temps d'exécution. Avec les informations d'implémentation : si `Var_FC_1`=3 et au bout de 10ms, `Runnable_Air_FC` n'a pas fini de s'exécuter, il y a une erreur. L'assertion **A2** correspondant à **D2** est écrite ci-après en pseudo code.

1- Type de Défaillance spécifique	**2- Assertions exécutables**
D2 : Flot de contrôle : Temps d'exécution trop long	 **Assertion A2 (en pseudo-code) :** `Var_FC_1` = 3 ; durée `Runnable_Air_FC` < 10 ms ;

La transmission de couple. Repartons de l'exigence **E3** : Le blocage en mode 1 alors que le système doit passer en mode 2 (idem pour mode 2) est une défaillance.

Dans le vocabulaire AUTOSAR, « mode 1» et « mode 2 » sont des *runnables*, ayant respectivement comme identifiant : `Runnable_Trans_Mod1` et `Runnable_Trans_Mod2`. Le contrôle de changement de mode est assuré par un *runnable* `Runnable_Trans_SM`. L'interprétation des données de l'environnement et de la commande du conducteur est implémentée dans le *runnable* `Runnable_Trans_DC`. Ces *runnables* appartiennent au composant logiciel `SWC_Trans` (cf. Fig.22).

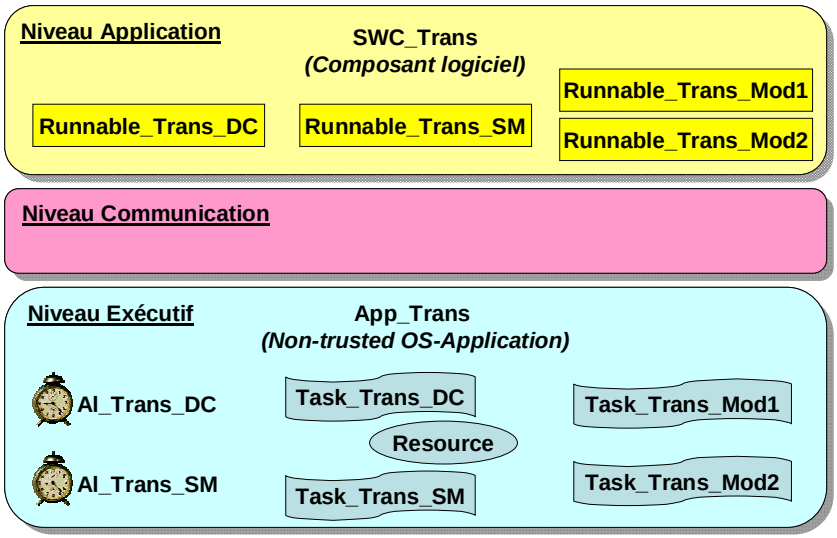


Figure 22 : Schéma architectural de l'application « transmission de couple »

`Runnable_Trans_DC` est implémentée dans la tâche `Task_Trans_DC` périodique à 50 ms, de priorité forte (2). La tâche est réveillée par une alarme `Al_Trans_DC` à 50ms. Cette tâche active la tâche `Task_Trans_SM` lorsque le conducteur demande un changement de mode, via le service `ChainTask()` de l'OS.

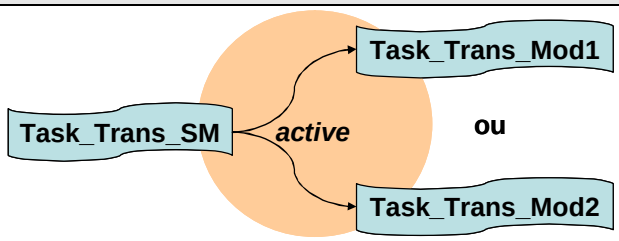
`Runnable_Trans_SM` est implémentée dans la tâche `Task_Trans_SM` périodique à 200ms, de priorité faible (1). La tâche est réveillée par une alarme `Al_Trans_SM` à 200ms. Les tâches `Task_Trans_SM` et `Task_Trans_DC` partagent une ressource. La tâche `Task_Trans_SM` active la tâche `Task_Trans_Mod1` ou `Task_Trans_Mod2` en fonction de la commande, via le service `ChainTask()` de l'OS.

`Runnable_Trans_Mod1` est implémentée dans la tâche `Task_Trans_Mod1` périodique à 200ms, de priorité forte (3), activée par `Task_Trans_SM` uniquement, via le service `ChainTask()` de l'OS.

`Runnable_Trans_Mod2` est implémentée dans la tâche `Task_Trans_Mod2` apériodique, de priorité forte (3).

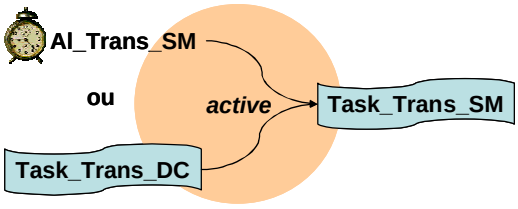
Les objets OS (`Task_Trans_DC`, `Al_Trans_DC`, `Task_Trans_SM`, `Al_Trans_SM`, `Task_Trans_Mod1`, `Task_Trans_Mod2`) appartiennent à l'OS-Application `App_Trans`, qui est « non-trusted » (en mode utilisateur).

L'étape (1a) a identifié une défaillance (**D3**) sur une transition invalide. Le blocage signifie que la commande n'est pas transmise à `Runnable_Trans_Mod1` ou `Runnable_Trans_Mod2`. L'activation de `Task_Trans_SM` par son alarme ou par `Task_Trans_DC` doit provoquer chaque fois l'activation de `Task_Trans_Mod1` ou `Task_Trans_Mod2`. L'assertion **A3** correspondant à **D3** est écrite ci-après en pseudo code.

1- Type de Défaillance spécifique	**2- Assertions exécutables**
D3 : Flot de contrôle : Transition invalide	 **Assertion A3 (en pseudo-code) :** Activation `Task_Trans_SM` pour mode 1 alors activation `Task_Trans_Mod1` ; Activation `Task_Trans_SM` pour mode 2 alors activation `Task_Trans_Mod2` ;

Repartons de l'exigence **E4** : Le passage intempestif « du mode 1 au mode 2 » ou « du mode 2 au mode 1 » est une défaillance.

Les informations sur l'implémentation sont les mêmes que précédemment. L'étape (1a) a identifié une défaillance (**D4**) sur une transition non voulue. Le changement de mode intempestif peut signifier que `Task_Trans_SM` est suractivé par son alarme défaillante ou par `Task_Trans_DC` trop fréquent (<1s) ou d'autres sources d'activation qui n'ont pas lieu d'être. L'assertion **A4** correspondant à **D4** est écrite ci-après en pseudo code.

1- Type de Défaillance spécifique	**2- Assertions exécutables**
D4 : Flot de contrôle : Transition non voulue	 **Assertion A4 (en pseudo-code) :** Cas d'erreur 1 : Activation `Task_Trans_SM` et source différente de `Task_Trans_DC` et `Al_Trans_SM` ; Cas d'erreur 2 : Activations successives de `Task_Trans_SM` par `Task_Trans_DC` inférieures à 1s ;

V.3 Phase de conception

V.3.1 Etape 2a : Conception des stratégies de détection

Les stratégies de détection d'erreurs sont déterminées à partir de la table 11. Dans la matrice de traçabilité, le tableau suivant regroupe les colonnes « 1- Types de défaillance logicielle » et « 3- Stratégie de détection d'erreur » définies en section III.4.

1- Type de Défaillance spécifique	3- Stratégie de Détection d'erreur
D1a : Flot de données : Valeur échangée non voulue	L'assertion exprime les conditions dans lesquelles la valeur d'une donnée est nominale. Les éléments de la **condition** et la **donnée** considérée sont donc tracés. Si l'assertion n'est pas vérifiée, une erreur est notifiée.
D1b : Flot de contrôle : Séquence d'exécution non voulue	L'assertion exprime les conditions dans lesquelles une séquence d'algorithmes de calcul ou d'évènements de contrôle est nominale. Les éléments de la **condition** et **l'algorithme de calcul** ou **évènement de contrôle** considéré sont donc tracés. Si l'assertion n'est pas vérifiée, une erreur est notifiée.
D2 : Flot de contrôle : Temps d'exécution trop long	L'assertion exprime les conditions dans lesquelles le temps d'exécution d'un algorithme de calcul est interdit. Les éléments de la **condition** et **l'algorithme de calcul** considéré sont donc tracés. Si l'assertion est vérifiée, une erreur est notifiée.
D3 : Flot de contrôle : Transition invalide	L'assertion exprime les conditions dans lesquelles un évènement de contrôle ne doit pas se produire. Les éléments de la **condition** et **l'évènement de contrôle** considéré sont donc tracés. Si l'assertion est vérifiée, une erreur est notifiée.
D4 : Flot de contrôle : Transition non voulue	L'assertion exprime les conditions dans lesquelles un évènement de contrôle doit se produire. Les éléments de la **condition** et **l'évènement de contrôle** considéré sont donc tracés. Si l'assertion n'est pas vérifiée, une erreur est notifiée.

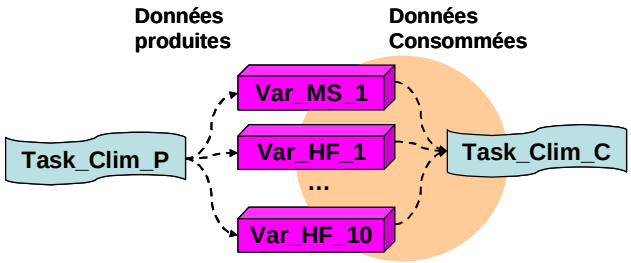
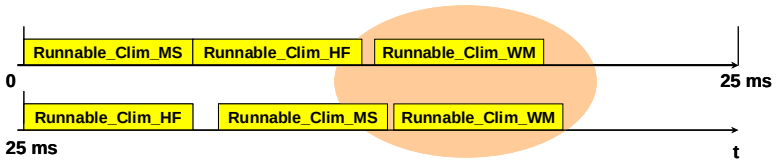
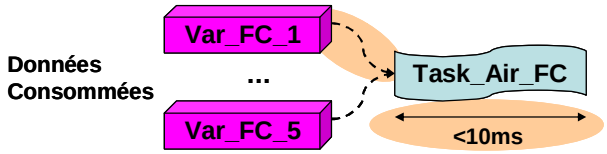
V.3.2 Etape 2b : Conception des stratégies de recouvrement

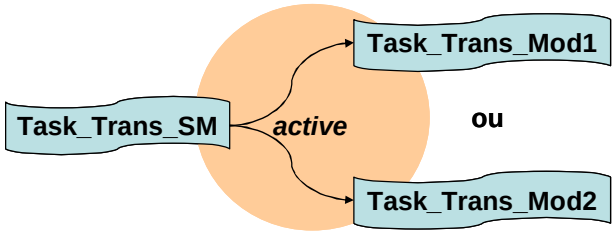
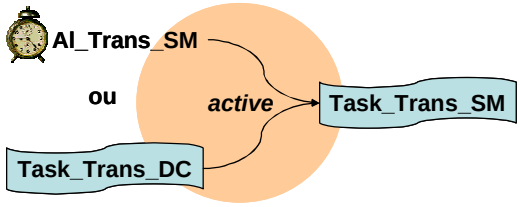
Les stratégies de recouvrement d'erreurs sont déterminées à partir de la table 12. Dans la matrice de traçabilité, le tableau suivant regroupe les colonnes « 1- Types de défaillance logicielle » et « 5- Stratégie de recouvrement d'erreur » définies en section III.4.

1- Type de Défaillance spécifique	5- Stratégie de Recouvrement d'erreur
D1a : Flot de données : Valeur échangée non voulue	Erreur notifiée : la valeur de la donnée est erronée. Le type de recouvrement choisi est par poursuite : annuler la valeur échangée, transmettre une valeur par défaut
D1b : Flot de contrôle : Séquence d'exécution non voulue	Erreur notifiée : La séquence d'exécution est erronée. Interrompre la séquence en cours. Recouvrement par reprise : Rétablir une bonne séquence.
D2 : Flot de contrôle : Temps d'exécution trop long	Erreur notifiée : Le temps d'exécution d'un algorithme de calcul est erroné. Interrompre l'algorithme de calcul en cours. Recouvrement par poursuite : Déclencher un mode dégradé
D3 : Flot de contrôle : Transition invalide	Erreur notifiée : La requête de transition est manquante. Recouvrement par compensation : Déclencher la transition.
D4 : Flot de contrôle : Transition non voulue	Erreur notifiée : La requête de transition est erronée. Recouvrement par compensation : Inhiber la transition en cours.

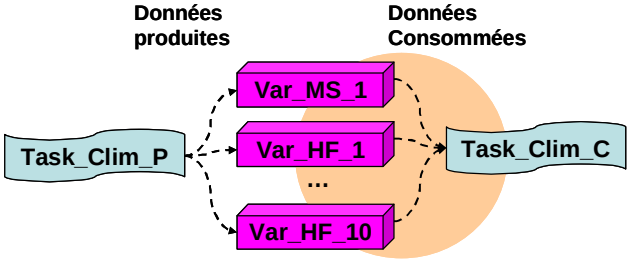
V.3.3 Etape 2c : Conception des stratégies de l'instrumentation

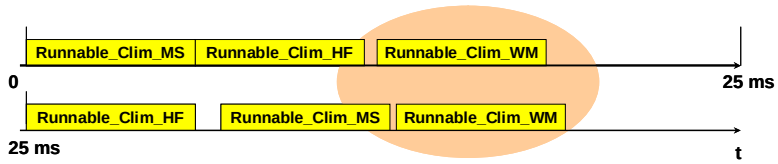
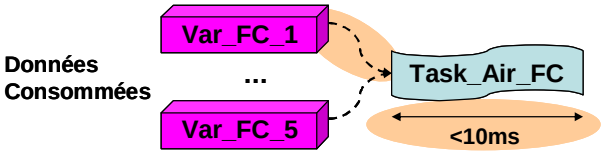
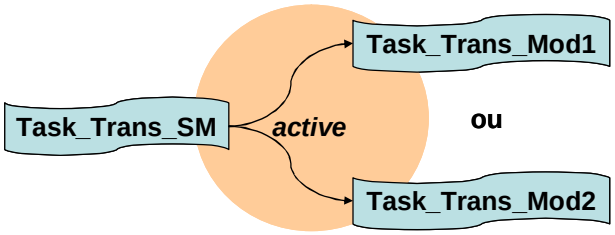
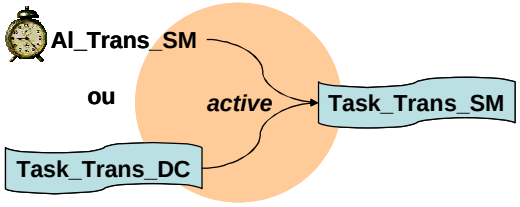
La stratégie de détection d'erreur (V.2.3) indique les informations à capturer pour les différentes assertions. L'implémentation est précisée en V.2.6. Dans la matrice de traçabilité, le tableau suivant regroupe les colonnes « 1- Types de défaillance logicielle » et « 4- Sondes logicielles » définies en section III.4.

1- Type de Défaillance spécifique	4- Sondes logicielles
D1a : Flot de données : Valeur échangée non voulue	 **Condition :** capturer l'identifiant de `Task_Clim_C` avant son exécution **Données :** capturer l'identifiant et la valeur de `Var_HF_1` à `Var_HF_10` et `Var_MS_1` à leur écriture
D1b : Flot de contrôle : Séquence d'exécution non voulue	 **Condition** : **Algorithmes de calcul** : capturer l'identifiant, l'instant de démarrage, l'instant de terminaison de `Runnable_Clim_HF, Runnable_Clim_MS, Runnable_Clim_WM`
D2 : Flot de contrôle : Temps d'exécution trop long	 **Condition** : capturer l'identifiant et la valeur de `Var_FC_1` à sa lecture **Algorithme de calcul** : capturer l'identifiant, l'instant de démarrage, l'instant de terminaison, et 10ms après le démarrage de `Runnable_Air_FC`

D3 : Flot de contrôle : Transition invalide	 **Condition** : capturer l'identifiant de `Task_Trans_SM`, `Task_Trans_Mod1`, `Task_Trans_Mod2` avant son exécution **Evènement de contrôle** : capturer l'instant d'appel et le paramètre (tâche activée) de `ChainTask()`
D4 : Flot de contrôle : Transition non voulue	 **Condition** : capturer l'identifiant de `Task_Trans_SM`, `Task_Trans_DC` avant leur exécution **Evènement de contrôle** : capturer l'instant d'appel et le paramètre (tâche activée) de `ChainTask()`, capturer l'état de l'alarme `Al_Trans_SM` avant l'exécution de `Task_Trans_SM`

La stratégie de recouvrement d'erreur (V.2.4) indique les actionneurs logiciels à utiliser pour réaliser le recouvrement. L'implémentation est précisée en V.2.6. Dans la matrice de traçabilité, le tableau suivant regroupe les colonnes « 1- Types de défaillance logicielle » et « 6- Actionneurs logiciels » définies en section III.4.

1- Type de Défaillance spécifique	6- Actionneurs logiciels
D1a : Flot de données : Valeur échangée non voulue	 **Annuler la valeur échangée erronnée, transmettre une valeur dégradée** : les données erronées peuvent être `Var_HF_1` à `Var_HF_10` et `Var_MS_1`. La valeur dégradée est spécifique à chaque donnée.

D1b : Flot de contrôle : Séquence d'exécution non voulue	 **Interrompre la séquence en cours, Rétablir la bonne séquence :** La séquence considérée est une séquence de *runnables*. Mais en termes de tâches : la tâche qui n'intervient pas au bon moment est forcément la tâche consommatrice `Task_Clim_C`, qui doit être arrêtée et la tâche productrice `Task_Clim_P` doit être relancée.
D2 : Flot de contrôle : Temps d'exécution trop long	 **Interrompre l'algorithme de calcul en cours, Déclencher un mode dégradé :** L'incohérence entre la donnée `Var_FC_1` et le temps d'exécution signifie que le calcul de l'algorithme est incorrect. Le mode dégradé est une réinitialisation et nouvelle lecture des données, pour tolérer une faute physique transitoire. Si la faute est permanente, notre architecture permet d'inhiber l'une ou l'autre des informations contradictoires, mais il est judicieux que le concepteur révise l'application.
D3 : Flot de contrôle : Transition invalide	 **Déclencher la transition :** La mémorisation de la commande dans la trace d'appels à `ChainTask()` permet de lancer `Task_Trans_Mod1` ou `Task_Trans_Mod2`
D4 : Flot de contrôle : Transition non voulue	 **Inhiber la transition en cours.** Une détection d'erreur de déclenchement de `Task_Trans_SM` a été faite donc cette tâche doit être terminée.

V.4 Phase d'implémentation

V.4.1 Etape 3a : Implémentation de l'instrumentation

V.4.1.1 Rôle du RTE AUTOSAR pour l'instrumentation

D'un point de vue implémentation, la couche logicielle AUTOSAR RTE est un code glue entre les applications et le logiciel de base. Il comporte deux parties principales (cf. Fig.23). D'une part, il offre des services de communication (lecture/écriture de donnée). D'autre part, il contient une partie de la configuration de l'ordonnanceur. Les *runnables* sont associées à des tâches dans lesquelles ils sont ordonnancés. Tandis qu'une tâche désigne un objet OS, un *Taskbody* désigne un programme situé dans le RTE. Le *Taskbody* contient la séquence des points d'entrée des *runnables* qui doivent s'exécuter dans le contexte d'une tâche donnée, et les divers appels de la tâche aux services de l'OS. Pour simplifier, la figure 23 ne fait apparaître que les *runnables* dans les *taskbodies*, sans les appels aux services OS.

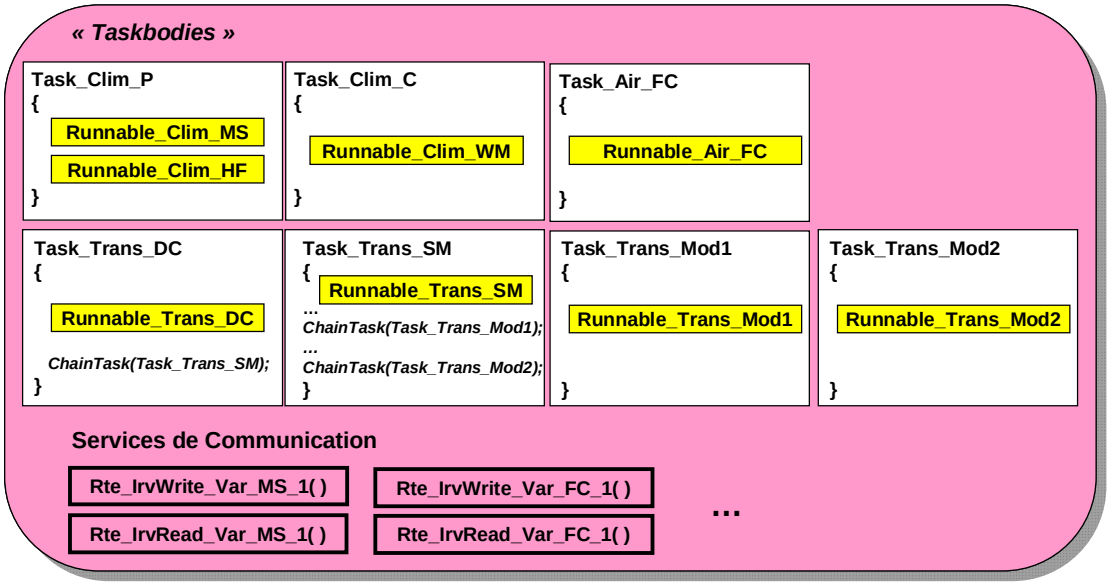


Figure 23 : Implémentation du RTE AUTOSAR

L'intégrateur possède *a priori* la maitrise de cette partie de code. Il peut donc intervenir dans les scripts des *taskbodies* et dans les routines des services de communication. Dans l'architecture AUTOSAR, c'est donc à ce niveau que nous pouvons le plus aisément placer des *hooks* pour l'instrumentation (cf. section IV.4.1). En effet, les scripts des *taskbodies* permettent d'intercepter le début et la fin des tâches à l'exécution. De plus, ils définissent les séquences d'appel des *runnables* et matérialisent ainsi le flot de contrôle des traitements applicatifs. D'autre part, les appels aux services de communication représentent le flot de données échangées.

Dans les sections suivantes, nous précisons les *hooks* (cf. Fig.24) que nous introduisons pour la détection et le recouvrement d'erreurs, ainsi que les sondes et actionneurs logiciels (cf. sections IV.4.2 et IV.4.3).

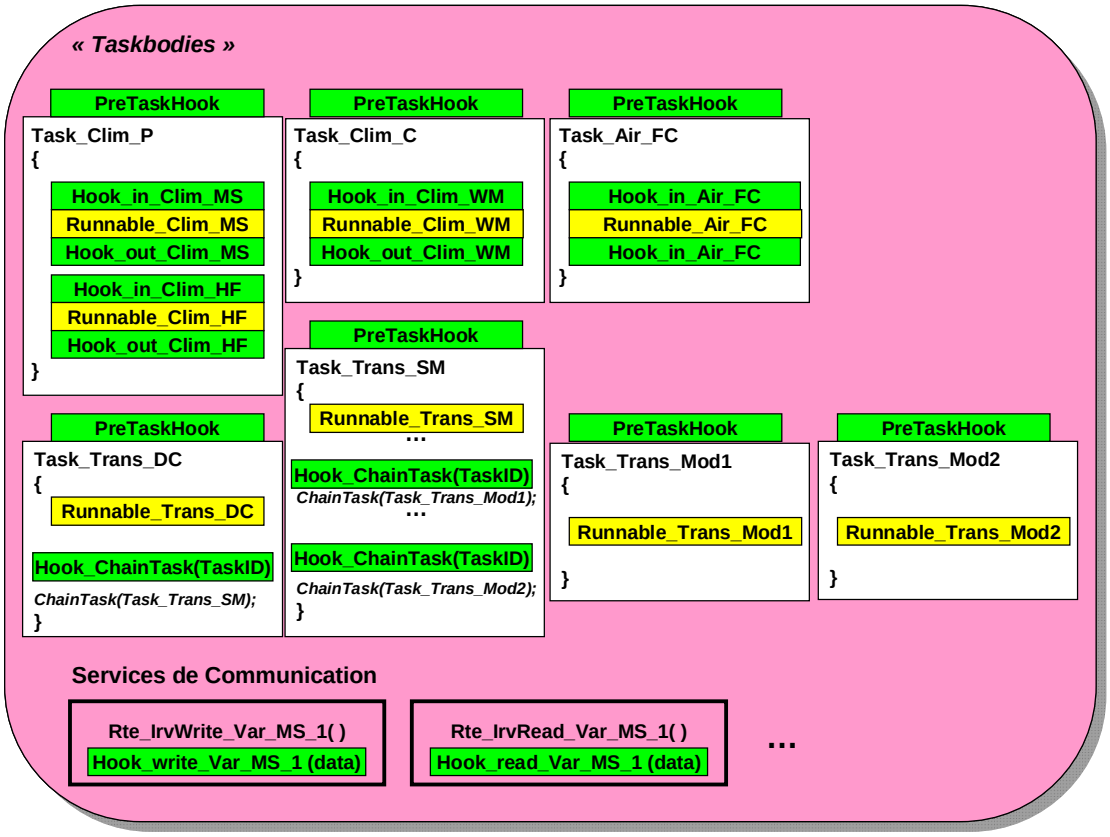


Figure 24 : Hooks utilisés dans le RTE AUTOSAR

V.4.1.2 Détection d'erreurs

En synthèse de la partie « Sondes logicielles » de la matrice de traçabilité (cf. section V.2.5), nous avons à tracer des tâches, des *runnables*, des données, et comme évènements de contrôle des services d'activation de tâche. Nous proposons ici une implémentation possible des sondes logicielles décrites en V.2.5. Elle peut évidemment être optimisée. Nous nous plaçons au niveau d'observabilité et de commandabilité d'un intégrateur qui a accès au code source du RTE, des interfaces de l'application et de l'OS.

Pour capturer les identifiants des tâches `Task_Clim_C`, `Task_Trans_DC`, `Task_Trans_SM`, `Task_Trans_Mod1`, `Task_Trans_Mod2`, avant leur exécution, nous avons utilisé les services présentés ci-après.

- Le service **`PreTaskHook()`** (cf. Fig.24) de l'OS peut être utilisé pour déclencher les routines d'enregistrement d'information et de vérification d'assertion, avant l'exécution de la tâche considérée. Ce service est appelé par toutes les tâches donc un algorithme simple est introduit pour sélectionner le déclenchement des routines seulement pour les tâches qui nous intéressent (énumérées ci-dessus).
- L'information d'identifiant de tâche courante peut être récupéré grâce au service **`GetTaskID()`** de l'OS. L'utilisation de ce service à l'intérieur de **`PreTaskHook()`** est possible car il donne alors l'identifiant de la tâche qui est sur le point de s'exécuter. (Cela permet alors de détecter les erreurs à l'entrée de la tâche avant de l'exécuter).

Pour capturer les *runnables* `Runnable_Clim_HF`, `Runnable_Clim_MS`, `Runnable_Clim_WM` et `Runnable_Air_FC` avant et après leur exécution, nous avons utilisé les services présentés ci-après.

- Les *runnables* sont appelés par leurs tâches respectives dans les scripts *TaskBodies* du RTE. Nous introduisons donc des appels à une fonction *hook* avant et après les runnables considérés, dans le code des *TaskBodies*. Si une seule fonction *hook* générique est utilisée, elle doit distinguer les *runnables* qu'elle traite. Comme les identifiants des *runnables* ne sont pas spécifié dans AUTOSAR, une solution d'implémentation, évitant l'utilisation d'identifiants, consiste à introduire des fonctions *hooks* spécifiques pour chaque *runnable*. Ainsi, `Runnable_Clim_HF` est encadré par exemple par `Hook_in_Clim_HF()` et `Hook_out_Clim_HF()` (cf. Fig.24).
- L'information temporelle est récupérée par le service **`GetAlarm()`**, qui donne une information relative du temps par rapport à une alarme. Il est possible d'introduire une nouvelle alarme ou utiliser les alarmes existantes.

Pour capturer les données `Var_HF_1 à Var_HF_10` et `Var_MS_1` à leur écriture et `Var_FC_1` à sa lecture, nous avons utilisé les services présentés ci-après.

- Nous introduisons après l'écriture et avant la lecture des données considérés des appels à une fonction *hook* spécifique (cf. Fig.29, `Hook_write_Var_HF_1()`, `Hook_read_Var_FC_1()`). Nous maîtrisons complètement les services de lecture et d'écriture de données implémentés dans le RTE. Par conséquent, nous pouvons introduire les appels de *hooks* directement à l'intérieur du programme de ces services de communication. D'ailleurs de tels appels de *hooks* sont implémentés automatiquement par le générateur de code que nous utilisons.
- Les données échangées peuvent être récupérées via le paramètre des fonctions de lecture et d'écriture. Les fonctions *hooks* utilisées doivent donc pouvoir passer des paramètres (cf. Fig.24, `Hook_write_Var_HF_1 (data)`).

Pour capturer l'évènement de contrôle `ChainTask()` à son appel, nous introduisons avant l'appel de `ChainTask()` une fonction *hook* spécifique (cf. Fig.24), passant comme paramètre, la tâche appelée par `ChainTask()`. `ChainTask()` est un service OS appelé dans les *TaskBodies* de `Task_Trans_DC` et `Task_Trans_SM`.

Enfin pour capturer l'information d'alarme de `Al_Trans_SM` avant l'exécution de `Task_Trans_SM`, le service **`GetAlarm()`** permet de savoir si l'alarme a expiré.

V.4.1.3 Recouvrement d'erreurs

Pour les services de recouvrement, nous détaillons ci-dessous les services standardisés AUTOSAR que nous utilisons, en reprenant un à un les actionneurs logiciels spécifiés en V.2.5.

A1a : Le service **`Rte_IrvWrite()`** du RTE initie une transmission de donnée sans file d'attente, par une communication émetteur-récepteur. Les *InterRunnableVariable* (IRV) sont les variables globales internes.

A1b : Le service **`ChainTask()`** de l'OS entraine la terminaison de la tache appelante. Après la terminaison de la tâche appelante, la tâche qui lui succède est activée. L'utilisation de ce service assure que la tâche suivante démarre au plus tôt après que la tâche appelante soit terminée.

A2 : Le service **`TerminateApplication()`** de l'OS termine un ensemble d'objets OS, dit « OS-Application » appelant. Il tue toutes les tâches, désactive les sources d'interruption et relâche toutes les autres ressources associées à l'OS-Application. Ce service peut être choisi si la tâche étudiée est liée à un ensemble d'objets OS et que le recouvrement doit impacter toutes les dépendances. Le paramètre `RESTART` du service `TerminateApplication()` permet de terminer l'« OS-Application » appelant et activer la tâche configurée comme tâche de redémarrage.

A3 : Avec le service **`ActivateTask()`** de l'OS, la tâche est transférée de l'état « suspendu » à l'état « prêt ». L'exécutif assure que le code de la tâche est bien exécuté à partir du premier énoncé.

A4 : Le service **`TerminateTask()`** de l'OS entraine la terminaison de la tâche appelante. L'état de celle-ci passe de « en exécution » à « suspendue ».

V.4.2 Etape 3b : Implémentation du logiciel de défense

Le code implémenté en mode superviseur doit être robuste et parfaitement contrôlé, car il a la possibilité d'agir sur toute la mémoire. Nous utiliserons donc chaque fois que c'est possible l'espace utilisateur.

Les OS-Applications encapsulant le logiciel fonctionnel sont "*non-trusted*", pour ne pas avoir accès à toute la mémoire et particulièrement aux informations critiques qui ne les concernent pas directement. Le logiciel de défense, qui contrôle l'ensemble des applications est implémenté dans une « *trusted OS-application* », appelée `App_Trusted` (cf. Fig.25).

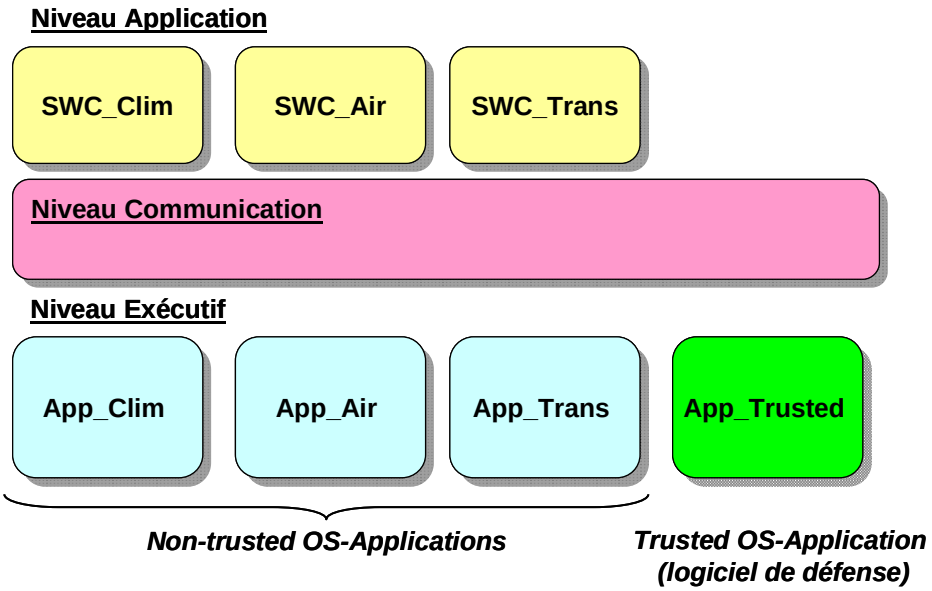


Figure 25 : Intégration du logiciel de défense

Les *hooks* sont principalement implémentés en espace utilisateur, en particulier ceux que nous avons ajoutés dans les *taskbodies* du RTE. Le service `PreTaskHook()` de l'OS est par contre en mode noyau (superviseur), par conception.

Les traces et informations de références des assertions sont les données les plus critiques de l'architecture, puisque toute la politique de tolérance aux fautes repose sur elles. Elles sont donc stockées dans un espace d'adressage protégé, séparé de la partie fonctionnelle.

Les routines d'enregistrement, de vérification et de recouvrement ont accès aux informations enregistrées en lecture et en écriture. Ainsi, elles doivent être implémentées en espace superviseur et être fiables. Ces routines sont appelées par des *hooks* qui appartiennent la plupart du temps au mode utilisateur. Un changement de mode est donc nécessaire. Ces mécanismes sont prévus dans AUTOSAR via les « *Trusted-Functions* ».

Pour résumer, les routines d'enregistrement, de vérification, et de recouvrement sont des "*trusted-fonctions*", appartenant à l'*OS-Application* `App_Trusted` (cf. Fig.26). Toutes les traces d'exécution partagent le même espace d'adressage que cette « *trusted OS-Application* ». Le contenu des "*trusted-fonctions*" est décrit en pseudo-code dans la figure 27, faisant apparaître les différents sondes et actionneurs logiciels utilisés en section V.2.6. Tout ceci constitue le logiciel de défense. L'interface entre le logiciel fonctionnel et le logiciel de défense est réalisée par les *hooks* en zone utilisateur (RTE). Si le logiciel fonctionnel ou de défense évolue, cela impactera indépendamment certaines « *trusted-function* », certains *hooks*, ou certains contenu de traces d'exécution. L'architecture de tolérance aux fautes est donc très flexible.

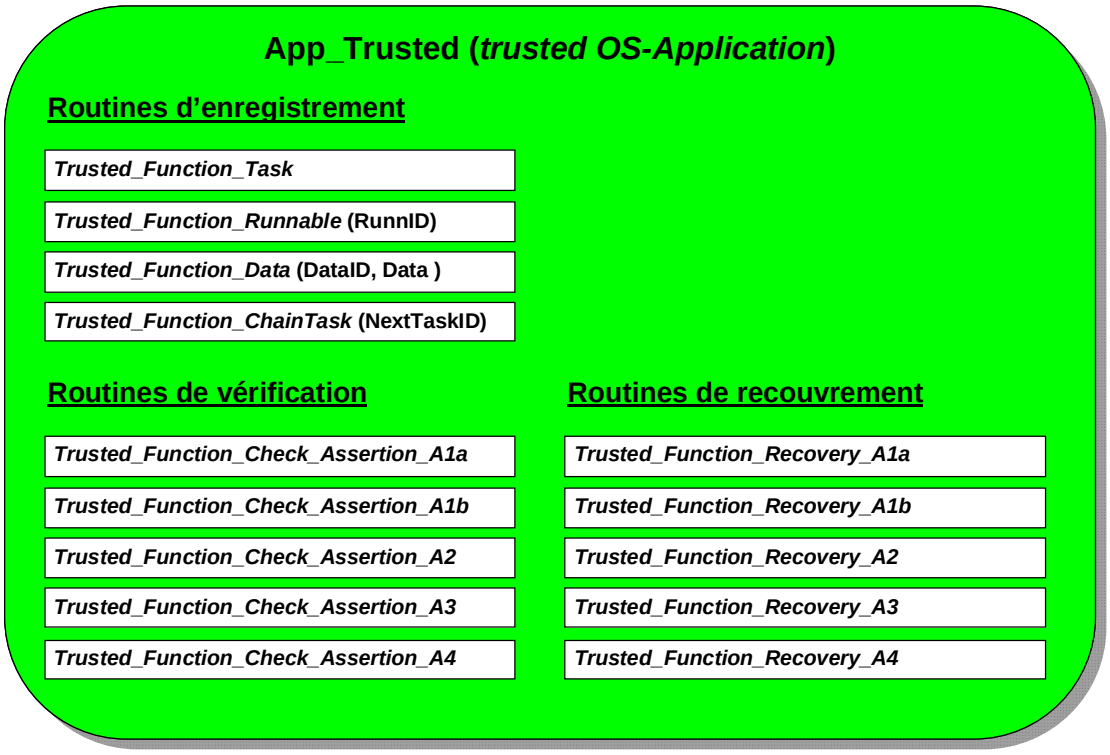


Figure 26 : Implémentation du logiciel de défense

Routines d'enregistrement

```
Trusted_Function_Task
{
    TaskID = GetTaskID();
    TaskID→ LogTable_Task[ ]
}
```

```
Trusted_Function_Runnable (RunnID)
{
    Time = GetAlarm();
    {RunnID, Time} → LogTable_Runnable[ ]
}
```

```
Trusted_Function_Data (DataID, Data )
{
    {DataID, Data} → LogTable_Data[ ]
}
```

```
Trusted_Function_ChainTask (NextTaskID)
{
    TaskID = GetTaskID();
   {TaskID, NextTaskID}→ LogTable_ChainTask[ ]
}
```

Routines de vérification

```
Trusted_Function_Checking_Assertion_A1a
{
    Assertion A1a;
    Si erreur alors Trusted_Function_Recovery_A1a;
}
```

```
Trusted_Function_Checking_Assertion_A1b
{
    Assertion A1b;
    Si erreur alors Trusted_Function_Recovery_A1b;
}
```

```
Trusted_Function_Checking_Assertion_A2
{
    Assertion A2;
    Si erreur alors Trusted_Function_Recovery_A2;
}
```

```
Trusted_Function_Checking_Assertion_A3
{
    Assertion A3;
    Si erreur alors
        Trusted_Function_Recovery_A3(Mod);
}
```

```
Trusted_Function_Checking_Assertion_A4
{
    Assertion A4;
    Si erreur alors Trusted_Function_Recovery_A4;
}
```

Routines de recouvrement

```
Trusted_Function_Recovery_A1a
{
    Rte_IrvWrite_Var_MS_1 (degraded value);
    Rte_IrvWrite_Var_HF_1 (degraded value);
            …
    Rte_IrvWrite_Var_HF_10 (degraded value);
}
```

```
Trusted_Function_Recovery_A1b
{
    ChainTask (Task_Clim_P);
}
```

```
Trusted_Function_Recovery_A2
{
     TerminateApplication (RESTART);
}
```

```
Trusted_Function_Recovery_A3 (Mod)
{
    Si Mod=Mod1 alors ActivateTask (Task_Trans_Mod1);
    Si Mod=Mod2 alors ActivateTask (Task_Trans_Mod2);
}
```

```
Trusted_Function_Checking_Assertion_A4
{
     TerminateTask( );
}
```

Figure 27 : Routines du logiciel de défense en pseudo-code

Conclusion du Chapitre V

L'étude de cas a permis de déployer la méthodologie et mettre en œuvre l'architecture de tolérance aux fautes, sur un système AUTOSAR, embarquant plusieurs applications automobiles synthétiques. Chacune des applications a permis d'illustrer des catégories d'exigences de sûreté différentes et le développement des mécanismes de protection associés (compatibles ISO26262).

L'implémentation de notre solution est compatible avec AUTOSAR. Avec un OS AUTOSAR qui gère la protection mémoire, le logiciel de défense correspond à un ensemble de routines en mode superviseur (« trusted functions *»). Ces routines commandent des services de l'OS et du RTE, pour observer et agir sur le système. A l'exécution, les routines du logiciel de défense sont déclenchées par des* hooks, *principalement en mode utilisateur. En effet, les appels aux* hooks *sont localisés de préférence dans le RTE, dont le code source est a priori accessible à l'intégrateur du système.*

Ainsi, nous avons montré la faisabilité de notre solution pour améliorer la robustesse du logiciel embarqué automobile, dans le contexte des standards AUTOSAR et ISO26262. La mise en œuvre de la méthodologie requiert depuis la phase d'analyse une connaissance partielle de l'implémentation. Ainsi, les acteurs industriels qui pourraient le plus vraisemblablement appliquer notre solution sont les intégrateurs logiciels. La coopération des concepteurs d'application est nécessaire pour donner les exigences de sûreté de haut niveau. De même, les fournisseurs de composants logiciels applicatifs ou du support d'exécution doivent communiquer les informations d'implémentation des flots de contrôle critiques. Ces conditions sont importantes pour permettre la gestion de la coexistence de fonctions de criticités différentes dans un même calculateur, et limiter la propagation d'erreurs liée à l'augmentation de la complexité du logiciel.

Chapitre VI

Validation expérimentale

Validation expérimentale

Le développement d'un système est incomplet sans un contrôle rigoureux vérifiant que l'implémentation est cohérente avec les spécifications. De plus, le logiciel de défense est un composant logiciel tenu d'être plus fiable que le logiciel fonctionnel, comme il contrôle celui-ci. Un processus de vérification renforcé du logiciel de défense est donc nécessaire pour améliorer la fiabilité. Les techniques de validation étudiées ne cherchent pas à remplacer mais s'ajoutent aux mesures de vérification et validation prévues pour un produit logiciel donné, en prenant la norme ISO26262 comme référence.

L'objectif de ce chapitre est double. Au delà des techniques et outillages classiques d'injection de fautes, nous montrons tout d'abord comment la démarche proposée dans la thèse se déroule jusqu'à la phase de validation expérimentale par injection de fautes. En second lieu, ce chapitre montre comment certains résultats d'expérience peuvent être analysés et le bénéfice que l'on peut tirer de la méthode proposée.

VI.1 Généralités sur l'injection de fautes

Le test est une activité importante de vérification et de validation en génie logiciel. Tester un logiciel consiste à l'exécuter en ayant la totale maîtrise des données qui lui sont fournies en entrée tout en vérifiant que son comportement est celui attendu. Le test par injection de fautes est répertorié comme un test fonctionnel unitaire, ou d'intégration, dans la norme [ISO26262-6]. En effet, l'injection de fautes peut être vue comme une méthode de test, telle que le domaine d'entrée relatif à l'activité fonctionnelle du système est élargi pour prendre en compte l'occurrence de fautes. Cette section rappelle des principes fondamentaux de l'injection de fautes, sans rentrer dans l'état de l'art des techniques existantes.

VI.1.1 Notions de fautes, erreurs, défaillances

Pour respecter strictement la chaîne de causalité « faute → erreur → défaillance » [Laprie 1996], une technique d'injection de fautes consiste en réalité à introduire des « erreurs », selon un modèle de « fautes » prédéfini. Enfin, le résultat des tests porte sur les « défaillances » constatées. Les fautes, les erreurs et les défaillances doivent être clairement identifiées pour décrire rigoureusement une technique d'injection de faute.

Le **modèle de fautes** est l'origine des erreurs et des défaillances. Il justifie la pertinence des tests, de la même manière qu'il justifie l'intérêt de mettre en œuvre des mécanismes de tolérance aux fautes pour le système. Dans notre cas, nous considérons donc le modèle de fautes décrit en section I.2, qui tient compte à la fois des fautes physiques et des fautes logicielles.

Les **modes de défaillances** sont spécifiques à une cible d'injection de fautes. Par exemple, pour caractériser la robustesse des services de système d'exploitation de type Unix, l'outil d'injection de fautes Ballista [Koopman & DeVale 1999] définit l'échelle « CRASH » :

- *Catastrophic* : le système a subit une défaillance majeure et doit redémarrer ;
- *Restart* : une application est bloquée et nécessite une relance ;
- *Abort* : une tâche ou une application s'est terminée anormalement, à la suite de la réception d'un signal ou une levée d'exception ;
- *Silent* : un appel système est exécuté avec des valeurs invalides des paramètres mais aucun code d'erreur n'est retourné ;
- *Hindering* : le code retourné est incorrect.

En fonction de l'objectif des tests d'injection de fautes, l'exploitation des résultats de défaillances est différente. L'injection de fautes est le plus couramment utilisée pour évaluer la **robustesse** de systèmes [Arlat *& al.* 1990]. La caractérisation des modes de défaillance de plusieurs composants sur étagère, par exemple, permet de les comparer entre eux. Dans ce cas, une grande quantité de tests est nécessaire pour obtenir des statistiques représentatives de la proportion de chaque mode de défaillances.

Par ailleurs, l'injection de fautes peut être employée pour la **vérification** de la conformité d'un système à ses exigences. Les tests de vérification d'exigences fonctionnelles sont généralement des tests fonctionnels, c'est-à-dire que pour des situations de référence le système se comporte de manière nominale. Il est donc intuitif que pour la vérification d'exigences de sûreté, les tests d'injection de fautes sont nécessaires. Ils garantissent qu'en présence de fautes, le comportement du système est nominal ou défini par l'exigence de sûreté. Dans ce cas, la quantité de tests d'injection de fautes n'est pas primordiale, comme l'intérêt n'est pas statistique. De la même manière que pour les tests fonctionnels, le résultat idéal attendu est que tous les tests amènent le système à tolérer les fautes, ou que la dégradation du service est acceptable. Les défaillances applicatives résiduelles montrent que le système n'est pas complètement conforme à ses exigences.

Les **erreurs** à injecter sont définies à partir du modèle de fautes et des modes de défaillances. Elles peuvent être réalistes ou non selon les objectifs recherchés. Si l'objectif est par exemple d'analyser les **conséquences d'une faute** physiques particulière et alléger l'environnement de tests par du logiciel, les erreurs injectées doivent être réalistes. Des études ont démontré que l'utilisation d'inversions isolées de bit entraîne des erreurs semblables à celles résultant de techniques d'injection physiques [Rimén & al. 1994] et permet de simuler les erreurs provoquées par les fautes logicielles avec une bonne fidélité [Madeira & al. 2000].

Pour une **approche orientée par les défaillances**, la représentativité des erreurs injectées n'est pas indispensable. La fin justifie les moyens, dans la mesure où le but des tests est uniquement de provoquer les défaillances. Ce type d'approche est particulièrement utilisé dans l'industrie pour améliorer les modes dégradés applicatifs. Par exemple, une fonction ou un calculateur est défaillant (peu importe la raison), le véhicule doit alors être mis dans un état sûr.

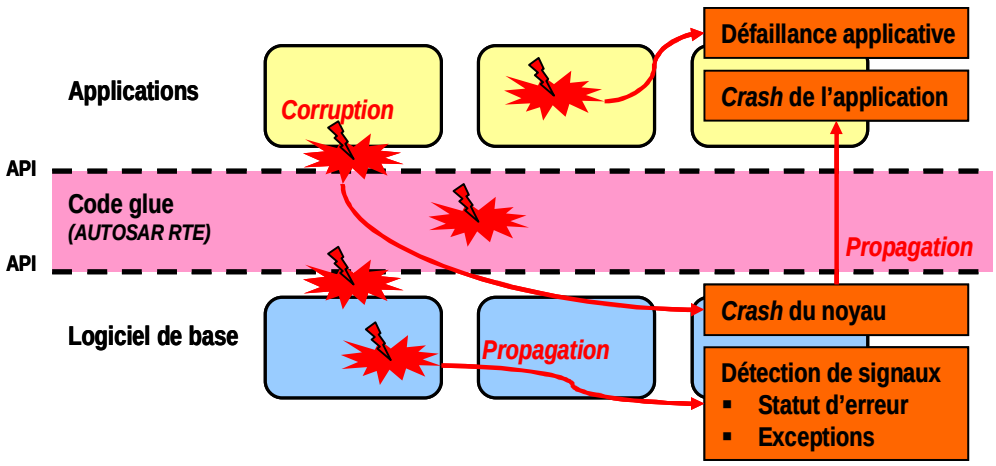


Figure 28 : Injection de fautes dans le code source

VI.1.2 Introduction d'erreurs dans un logiciel modulaire multicouche

Plus précisément, la spécification d'un test d'injection de fautes doit décrire **où, quand et comment** introduire l'erreur. La durée d'introduction des erreurs permet de simuler des fautes permanentes ou transitoires. Le moment de l'injection de fautes dépend de l'objectif de vérification (selon l'exigence de sûreté considérée) ou de robustesse (distribution statistique).

Ayant comme cible le code source d'un logiciel modulaire et multicouche, les divers endroits possibles pour corrompre le code sont : les **interfaces**, les **composants logiciels** et le **code glue** qui assure l'interaction entre les modules. La figure 28 illustre les différents sites d'injection de fautes, pour un logiciel en couches de type AUTOSAR. Elle montre aussi comment les erreurs peuvent se propager à travers les couches de l'architecture et provoquer des modes de défaillances plus ou moins importants, au niveau des applications ou du logiciel de base.

La **corruption de paramètres** des interfaces est une technique non intrusive : les modules logiciels sont considérés comme des boîtes noires avec des vecteurs d'entrée et de sortie à faire varier. Cette méthode est très utilisée dans l'industrie, grâce à l'existence de nombreux outils commerciaux de tests fonctionnels (*Software in the Loop*), pouvant facilement être étendus à des tests d'injection de fautes. L'inconvénient de ce type de tests est qu'il ne s'adresse qu'aux flots de données.

La perturbation des composants logiciels et du code glue peut être réalisée par une **modification, un ajout ou une suppression de code**. Une autre possibilité est d'intégrer au système un **composant applicatif** (configurable en terme de priorité, de durée, de déclenchement d'exécution, etc.) **dédié au sabotage** du comportement fonctionnel. Ces deux techniques permettent de corrompre aussi bien les flots de données que de contrôle. Cependant, comme elles sont intrusives dans des parties du code de la cible, elles ne sont réalisables que si le code source de la cible est disponible (au moins partiellement).

La corruption du code source, au niveau des interfaces, des modules ou du code glue peut également être effectuée directement dans la **mémoire**. Pour injecter les fautes pendant l'exécution, il faut arrêter le flot d'exécution, écrire dans la mémoire puis reprendre l'exécution. Cette fois-ci, la technique est intrusive temporellement, si l'injection de fautes n'est pas suffisamment rapide. Ces tests n'ont alors plus de représentativité vis-à-vis des contraintes temps réel du système.

Comme les techniques d'injections de fautes représentent un sujet très riche qui dépasse largement le périmètre de notre travail, nous choisirons de mettre en œuvre la technique qui nous semble la plus simple et complète pour nos besoins. Nous disposons du code source de notre cible et nous souhaitons perturber à la fois les flots de contrôle et de données. **Nous avons donc étudié une méthode de mutation de code**. Le code inséré pour provoquer une défaillance est appelé **« mutant »**.

VI.2 Technique d'injection de fautes utilisée

Notre objectif est d'évaluer la tolérance aux fautes de notre architecture, en d'autres termes l'efficacité du logiciel de défense ajouté. Les hypothèses de fautes que nous considérons sont les **fautes physiques et logicielles** décrites en section I.2.

Même si nous voulons vérifier le logiciel de défense, la cible d'injection de fautes est en réalité les applications. En effet, le logiciel de défense vise à éviter des défaillances applicatives. Il s'agit donc de perturber la partie fonctionnelle du logiciel pour évaluer les capacités de la tolérance aux fautes du logiciel de défense.

Les **modes de défaillances** considérés sont alors différents des modes de défaillance d'un système d'exploitation (échelle *CRASH, section VI.1.1*).

Les 3 modes de défaillances étudiés sont :

- **Les défaillances applicatives.** Nos mécanismes réflexifs ont pour but d'éviter ces défaillances et permettre au système de rester opérationnel, en dépit de la présence d'erreurs. Ce mode de défaillance est spécifique et doit être décrit plus précisément pour un cas d'étude donné.

- **Le *crash* ou erreur de segmentation.** En général, il est difficile de remédier à ce type de défaillance avec l'architecture de tolérance aux fautes proposée, à moins d'introduire de la redondance matérielle. Nous nous intéressons donc peu aux tests conduisant à ce type de défaillance.

- **La tolérance aux fautes (TaF) ou pas d'observation.** L'application fonctionne correctement, son comportement respecte les spécifications. Cela signifie soit que l'erreur injectée est latente, soit que les mécanismes de tolérance aux fautes ont traité correctement l'erreur injectée. Le temps de réponse est plus long (mais acceptable selon les contraintes temporelles imposée), lorsque les mécanismes de tolérance aux fautes ont été activés.

Les défaillances du logiciel embarqué peuvent affecter globalement les flots de données et/ou de contrôle, selon les catégories de référence que nous avons définies au chapitre III (cf. Tab.9). **L'injection de fautes consiste donc à provoquer de façon contrôlée ces défaillances, peu importe la manière.** En effet, les mécanismes de tolérance aux fautes mis en œuvre ciblent les défaillances, indifféremment de leur origine physique ou logicielle.

Les **erreurs que nous injectons** correspondent donc à l'ajout dans le code « **d'actionneurs logiciels** », tels que nous les avons définis en section IV.4.3. En effet, les actionneurs logiciels, qui permettent d'agir sur les flots de données et de contrôle, peuvent être considérés soit comme des outils de correction de l'exécution par le logiciel de défense, soit comme des outils de perturbation de l'exécution pour l'injection de fautes.

Le **flot de contrôle** étant principalement contrôlé par l'exécutif temps réel, une méthode relativement simple pour le perturber consiste à utiliser les services de

l'exécutif temps réel. Une connaissance fine des appels système utilisés par l'application permet même une inhibition ciblée du comportement attendu, conduisant à une défaillance particulière. Par exemple, la demande explicite d'une activation de tâche, peut être perturbée par un service de terminaison de tâche.

Pour le **flot de données**, une démarche analogue peut être menée en utilisant des services de communication qui inhibent ou perturbent le comportement prévu. Cela dit, dans l'état de l'art de nombreuses techniques existent pour corrompre les données (*Software-In-the-Loop*, *Bit-Flip*, etc.).

Nous nous plaçons au niveau de l'intégrateur qui maîtrise le code glue entre les applications et le logiciel de base, dont il ne dispose que des interfaces. La technique d'injection de fautes que nous utilisons consiste à utiliser des services fonctionnels du logiciel de base (services exécutifs et de communication), pour provoquer des erreurs au niveau du code glue, et entraîner des défaillances applicatives.

Le **réalisme** des erreurs injectées n'est pas fondamental dans notre approche orientée par les défaillances. Notons que la probabilité d'occurrence de telles erreurs de programmation dans le code est certainement très faible, pour des développeurs expérimentés ou des générateurs de codes matures ou certifiés. Cela dit, le déploiement du standard d'architecture logicielle AUTOSAR favorise l'utilisation d'outils de génération automatique de code, développé par des tiers. Malgré les nombreux avantages de cette méthode, l'utilisation d'outils commerciaux limite la maîtrise du code produit, et déplace les efforts manuels au niveau de la configuration des outils. La complexité des paramétrages, ajoutée à une connaissance limitée des outils de génération de code, augmentent le risque d'erreurs de concurrence, d'entrelacement ou d'inhibition de services de support d'exécution.

L'injection de fautes par ajout d'appels à des services exécutifs ou de communication est donc finalement représentative d'erreurs de configuration et de génération de code. Ces erreurs lorsqu'elles apparaissent réellement dans un système sont généralement permanentes. Pour notre technique d'injection de fautes, il est possible d'injecter les erreurs de manière transitoires ou permanentes. Dans le cas d'une erreur permanente, il s'agit d'insérer avant la compilation, dans une tâche, une ligne de code contenant un appel de service, qui sera appelé chaque fois que la tâche est appelée. Une erreur transitoire consiste à associer l'appel de service à un compteur qui détermine à quel moment déclencher l'appel une seule fois. Notre logiciel de défense traite a priori aussi bien les défaillances transitoires que permanentes, comme il est conçu pour traiter une défaillance chaque fois qu'elle apparaît en ligne. Il n'est donc pas réducteur de tester son efficacité seulement pour des **erreurs permanentes**.

La **matrice de traçabilité** des exigences structure notre technique d'injection de fautes. Nous définissons pour chaque type de défaillance une stratégie de vérification (cf. Tab.13). Elle consiste à déterminer les éléments du système à perturber. Elle est proche de la stratégie de détection d'erreur (cf. Tab.11, Chap.IV), à la seule différence que les éléments qui étaient tracés sont maintenant les cibles d'injections d'erreurs.

1- Type de Défaillance spécifique	7- Stratégie de Vérification
Flot de données : Valeur échangée invalide (/non voulue)	L'assertion exprime les conditions dans lesquelles la valeur d'une donnée est interdite (/nominale). Il s'agit de corrompre les éléments de la **condition** ou la **donnée** considérée.
Flot de données : Temps d'échange de donnée trop long (/ trop court)	L'assertion exprime les conditions dans lesquelles le temps d'échange d'une donnée est interdit. Il s'agit de corrompre les éléments de la **condition** ou la **donnée** considérée.
Flot de contrôle : Transition invalide (/ non voulue)	L'assertion exprime les conditions dans lesquelles un évènement de contrôle ne doit pas (/doit) se produire. Il s'agit de corrompre les éléments de la **condition** ou **l'évènement de contrôle** considéré.
Flot de contrôle : Séquence d'exécution invalide (/ non voulue)	L'assertion exprime les conditions dans lesquelles une séquence d'algorithmes de calcul ou d'évènements de contrôle sont interdites (/nominale). Il s'agit de corrompre les éléments de la **condition** ou **l'algorithme de calcul** ou l'**évènement de contrôle** considéré.
Flot de contrôle : Temps d'exécution trop long (/ trop court)	L'assertion exprime les conditions dans lesquelles le temps d'exécution d'un algorithme de calcul est interdit. Il s'agit de corrompre les éléments de la **condition** ou **l'algorithme de calcul** considéré.

Table 13 : Stratégie de vérification

VI.3 Protocole d'injection de fautes

Nous considérons un programme cible implémenté en langage C. Pour injecter les erreurs de manière contrôlée, nous suivons un protocole rigoureux. La première partie consiste à terminer de remplir la **matrice de traçabilité**, qui guide la technique d'injection de fautes :

- Partir de chaque défaillance identifiée dans l'étape (1a) de la méthodologie et des informations d'implémentation ;
- Identifier et lister les éléments du système à corrompre. La stratégie de vérification est donnée dans le tableau de référence Table 13, section VI.2.
- Identifier et lister les services de l'architecture logicielle qui permettent de corrompre les éléments ciblés.

La suite de la description du protocole détaille **une expérience** d'injection de fautes :

- Identifier la partie de code qui contient le flot de contrôle du système, plus précisément les appels de fonctions applicatives. Par exemple, pour une cible AUTOSAR, cette partie de code correspond aux *Taskbodies*.
- Chaque expérience d'injection de faute consiste à insérer un appel de service déterminé précédemment, dans une ligne de code de la partie de code ciblée (cf. Fig.29), conduisant à la génération de « mutants ».

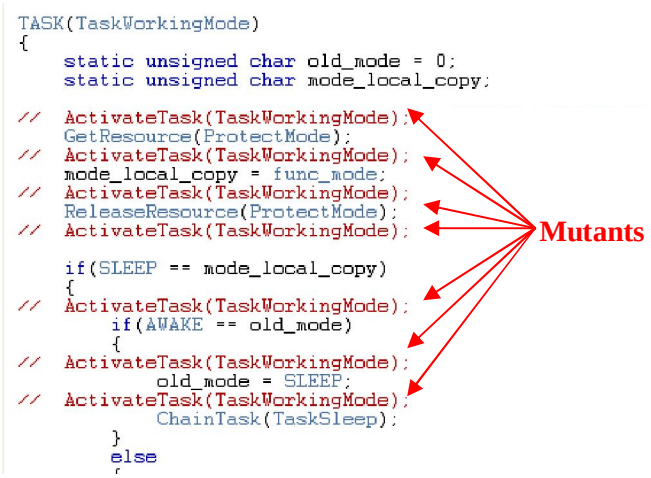


Figure 29 : Injection de fautes

Enfin, la dernière partie du protocole décrit **la campagne de tests**.

- L'expérience est répétée pour toutes les lignes de code de la partie de code ciblée. L'intérêt d'injecter une erreur sur des lignes différentes est de stimuler un moment particulier du flot d'exécution (au début d'une tâche, à la fin d'une fonction, avant un appel système, etc.).
- L'expérience est répétée pour les différents services identifiés pour perturber la cible.

Simuler les erreurs que nous considérons pour la tolérance aux fautes permet d'évaluer l'**efficacité des mécanismes de protection** introduits. En fonction du type de recouvrement choisi, la présence du logiciel de défense doit permettre la mise du système en mode dégradé ou la continuité du service, au lieu d'une défaillance applicative. Si la défaillance applicative ou un autre type de défaillance (ex : crash OS) a lieu, le logiciel de défense n'est alors pas suffisamment robuste. Dans ce cas une analyse fine de la défaillance au niveau de l'implémentation peut permettre de raffiner l'assertion exécutable à vérifier et/ou le recouvrement à réaliser. Notons qu'il est conseillé de reprendre la totalité de la campagne de test, une fois que le logiciel de défense a été modifié (tests de non-régression).

VI.4 Outils d'injection de fautes

Les outils d'injection de fautes, qui permettent l'automatisation des campagnes de tests, sortent du cadre de notre étude. En nous intéressant en premier à la vérification de la conformité des exigences, le nombre de tests que nous avons effectué dans notre cas d'étude (quelques dizaines de tests par exigence) ne justifie pas immédiatement le besoin d'automatisation. Cela dit, nous avons étudié la possibilité d'automatiser nos campagnes de tests en concevant différents outils d'injection de fautes. Nous donnerons seulement brièvement les conclusions des prototypes implémentés rapidement. Cette étude et les développements associés nous ont permis de nous approprier les concepts, les techniques et les architectures logicielles permettant de mettre en œuvre une phase de validation expérimentale par injection de fautes.

VI.4.1 1er outil en simulation sur UNIX

Le premier outil (cf. Fig.30) a été développé en C++, pour la simulation de nos plateformes logicielles embarquées sur un simulateur de processeur virtuel (« *Viper* ») supporté par UNIX. L'outil en lui-même est globalement composé de la gestion de **lancement** des campagnes de test et de la **récupération** des résultats et leur analyse. Le service de lancement charge les données (OS, application) dans la cible, avant une expérience. Après une durée de test prédéfinie, la cible est redémarrée et réinitialisée. Lorsque le redémarrage logiciel est impossible, un redémarrage matériel peut être envisagé par l'intermédiaire d'une carte extérieure connectée à la cible. Le service de récupération collecte les données d'une séquence de test pour les sauvegarder sur l'hôte.

La séparation de l'outil sur une machine hôte et du logiciel à tester sur une machine cible est représentative de l'environnement de test si la cible était embarquée sur un microcontrôleur. Une partie de l'outil d'injection de fautes est intrusive dans la cible : les services d'**observation**. En effet, l'outil doit pouvoir capturer automatiquement le mode de défaillance de la cible après l'injection de fautes.

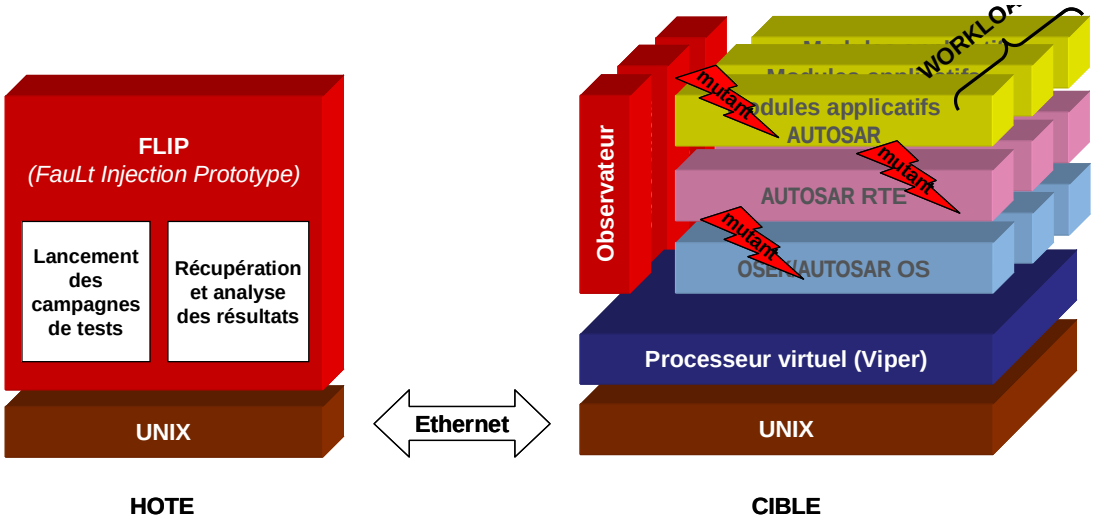


Figure 30 : Outil d'Injection de fautes n°1

L'injection de fautes devrait idéalement être réalisée (dans l'hôte ou la cible) par un service de **sabotage** qui permettrait d'ajouter de manière automatique et configurable une ligne de code (mutant) dans un programme.

Cet outil d'injection de fautes a le mérite d'identifier les services principaux pour l'automatisation des tests, et de montrer la faisabilité d'un outil simple. La structure de cet outil peut être réutilisée pour développer un environnement de test élaboré. Les difficultés et limitations que nous avons rencontrées sont liées au manque de temps que nous avons consacré à l'optimisation de l'outil. Nous n'avons pas cherché à implémenter de service de sabotage. Les erreurs (mutants) ont été insérées manuellement dans des logiciels cibles différents (« *workload* »), avant la compilation. Finalement, l'outil revient à l'automatisation du lancement des tests et de la récupération de leurs résultats.

Une autre difficulté était de définir l'instrumentation de l'application cible pour obtenir la trace d'exécution, à partir de laquelle les défaillances applicatives peuvent être détectées, par comparaison à un oracle (c'est-à-dire une trace de référence si l'application est déterministe). Cette instrumentation devrait idéalement être indépendante des mécanismes mis en place pour la détection d'erreurs de notre logiciel de défense. Lorsque l'instrumentation d'observation de l'outil d'injection de fautes est la même que celle du logiciel de défense, il est possible que l'injection de faute ait produit une défaillance applicative, qui ne puisse être détectée ni par le logiciel de défense, ni par l'outil d'injection de fautes. Ce problème, combiné à un système de communication perturbant les contraintes temps réel, donnait des traces d'exécution peu fiables.

VI.4.2 2eme outil de type « *bit-flip* » sur microcontrôleur

Nous avons mis en œuvre l'automatisation de tests sur microcontrôleur pour évaluer qualitativement la complexité et les difficultés de développement, en comparaison avec la simulation.

Encore une fois, la méthode d'injection de fautes (sabotage) a proprement dit n'a pas été étudiée de manière approfondie. Nous avons donc choisi d'injecter des erreurs, en inversant des bits aléatoirement dans la mémoire (*bit-flip*). Nous pouvons comparer cet outil avec le précédent (cf. section VI.4.1), seulement en termes d'automatisation du lancement de test et de la récupération des résultats.

La technique (cf. Fig.31) repose sur l'utilisation d'une sonde de débogage (Trace32 de Lauterbach) et d'un dispositif matériel BDM (*Background Debug Mode*) implémenté dans le processeur (S12XEP100). L'interface BDM offre un mode spécial qui autorise à la sonde de contrôler le processeur et d'avoir accès à la mémoire (RAM, EEPROM, Flash, registres), et aux entrées/sorties. La sonde est associée à un environnement logiciel de débogage, permettant la programmation *batch* de scripts, en langage propriétaire. Dans notre approche, l'automatisation dépend entièrement des services offerts par la sonde pour contrôler la cible. Le protocole d'injection de fautes est traduit en un algorithme appelant les services de la sonde, implémenté dans un script.

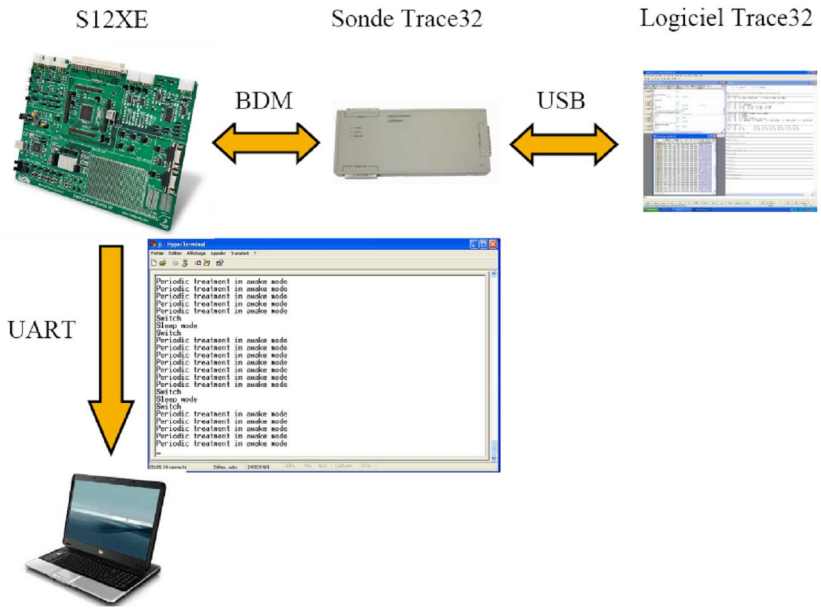


Figure 31 : Outil d'Injection de fautes n°2

L'architecture de cet outil est inspirée de l'outil précédent sur simulation (cf. section VI.4.1). Un service de lancement effectue des étapes d'initialisation de la cible (vérification que la sonde est connectée, réinitialisation de la carte, identification du type de processeur, sélection du mode spécial du processeur). Puis, il charge le code exécutable de l'application, active le mode spécial du processeur, et exécute l'application. Après une durée de test prédéfinie, le service de récupération des résultats détermine si le processeur s'est réinitialisé ou si l'application continue de s'exécuter. Les résultats sont enfin écrits automatiquement dans un rapport, sous forme de fichier « .dat ».

Les limitations de l'étude concernent encore une fois l'observation des modes de défaillance du système. Seuls les *crash* peuvent être détectés facilement, car la sonde sait les détecter. En ce qui concerne les défaillances applicatives, nous les avons observées par l'intermédiaire d'un affichage de traces sur le terminal de la machine hôte. Ces traces doivent être enregistrées pour pouvoir les comparer à une trace de référence. Par ailleurs, le service de sabotage est chargé ici d'arrêter l'application pour réaliser une inversion de bit dans la mémoire. L'étude de l'intrusion temporelle de l'outil, due à l'injection de fautes, n'a pas été menée.

Les points positifs qui ressortent de cet outil, par rapport à la simulation, sont la disparition du problème de communication entre hôte et cible : la gestion matérielle de la communication est efficace (peu de perturbation temporelle à ce niveau). De plus, la complexité du développement de l'outil d'injection de fautes est faible (~200 lignes de code, dans un fichier *batch*), par rapport à l'outil précédent (37 ko).

VI.5 Environnement de tests

Ces deux prototypes d'outils d'injection de fautes, décrits dans la section précédente, peuvent être considérés comme une base pour développer un outil plus performant et industrialisable. Ils possèdent une structure générale simple et soulignent les aspects à améliorer pour être vraiment utilisables.

Pour la validation de notre étude de cas, nous n'avons finalement pas utilisé ces outils, qui demandent un effort d'amélioration dépassant le cadre de la thèse, mais dont l'étude a été très profitable au déroulement des campagnes que nous avons réalisées par une approche manuelle. Nous rappelons ici que notre objectif consiste à tester les mécanismes de tolérance aux fautes par rapport au modèle de faute qui leur est associé. Il s'avère que, dans notre cas d'étude, le nombre de cas de test est assez limité (quelques dizaines), une approche manuelle est donc faisable en s'inspirant des outils étudiés. Il est clair que ceci n'est pas possible pour des tests de robustesse statistiques pour lesquels l'usage d'outils est indispensable [Rodriguez *et al.* 2002 DSN].

L'environnement de développement et le débogueur CodeWarrior™ de Freescale, s'exécutant sur une machine hôte, suffit donc pour réaliser les tests d'injection de fautes. La cible est un microcontrôleur S12XEP100™ de Freescale, connecté à la machine hôte via une liaison USB. CodeWarrior™ permet d'éditer le code de l'application, le modifier manuellement (insertion de mutants) puis compiler et exécuter l'application à partir du débogueur (cf. Fig.32). Pour observer le comportement de l'application, test par test, nous avons envoyé des informations de traçage sur un terminal d'affichage (HyperTerminal Windows), via une liaison série.

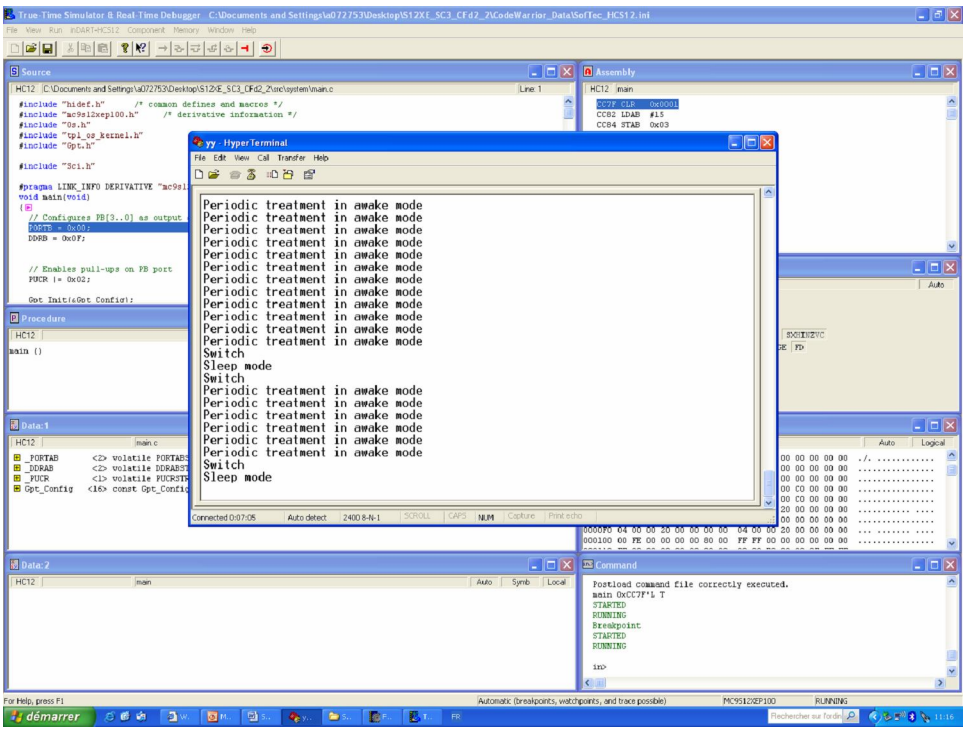


Figure 32 : Débogueur et terminal dans un environnement Windows

VI.6 Illustration de l'approche de vérification

Dans l'étude de cas présentée au chapitre précédent, nous avons réalisés des tests d'injection de fautes ciblant les différentes applications séparément. Nous présentons ici la vérification des exigences de sûreté E4 de l'application « transmission de couple » (cf. section V.2.1).

Rappelons que la fonction de transmission est composée de deux parties de contrôle. Le contrôle statique récupère les commandes du conducteur et de l'environnement (`Task_Trans_DC`), puis calcule des valeurs de consigne pour la vitesse du moteur, la puissance de batterie et le couple aux roues. A partir de ces données, le contrôle dynamique (`Task_Trans_SM`) calcule les couples du moteur thermique et des moteurs électriques (`Task_Trans_Mod1 et Task_Trans_Mod2`).

Dans l'application transmission de couple, pour simplifier, le « mode 1 » est le mode où le moteur électrique est actif, tandis que le « mode 2 » est celui où le moteur électrique est inactif (devenu générateur en fait) et remplacé par le moteur thermique. Le mode 1 est repéré par le message « ***Periodic treatment in awake mode*** » (point de vue du moteur électrique), envoyé par la tâche **périodique** `Task_Trans_Mod1`. Le mode 2 est repéré par le message « ***Sleep mode*** » (point de vue du moteur électrique), envoyé par la tâche **apériodique** `Task_Trans_Mod2`. La requête de changement de mode est tracé par le message « Switch », envoyé par la tâche `Task_Trans_SM`. La figure 32 montre ces messages dans l'hyperterminal.

Le mode de défaillance applicative considéré pour l'exigence E4 est une transition intempestive non voulue. La figure 33 montre que deux transitions seulement ont été demandées, alors que la trace en montre 3.

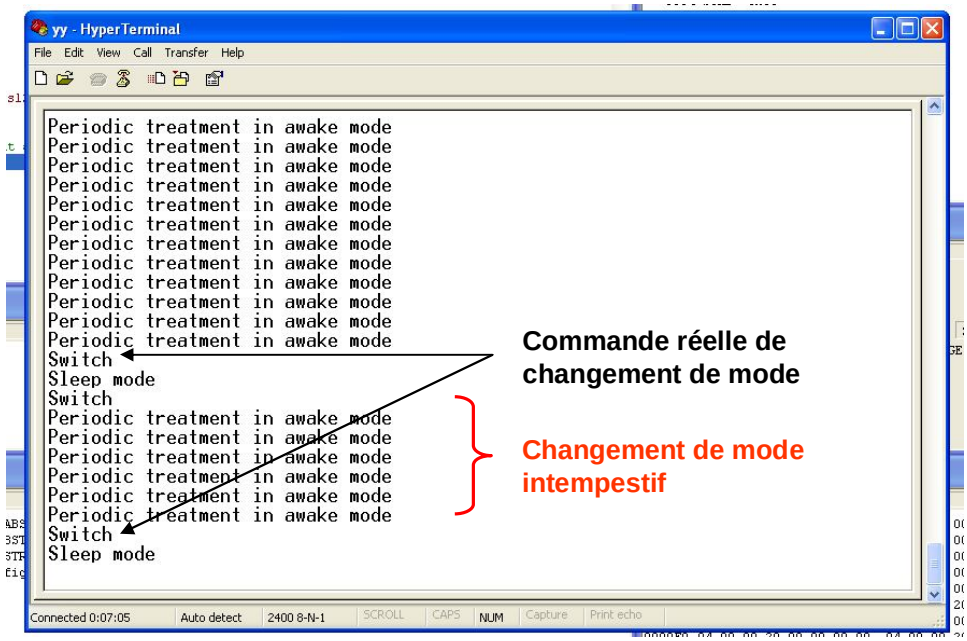


Figure 33 : Défaillance applicative (transition non voulue)

La figure 34 illustre l'implémentation de la partie de l'application « transmission de couple » qui nous intéresse. La tâche `Task_Trans_DC` est réveillée par une alarme `Al_Trans_DC` à 50ms. Cette tâche active la tâche `Task_Trans_SM` lorsque le conducteur demande un changement de mode, via le service `ChainTask()` de l'OS. Par ailleurs, la tâche `Task_Trans_SM` est réveillée par une alarme `Al_Trans_SM` à 200ms. Les tâches `Task_Trans_SM` et `Task_Trans_DC` partagent une ressource. La tâche `Task_Trans_SM` active la tâche `Task_Trans_Mod1` ou `Task_Trans_Mod2` en fonction de la commande, via le service `ChainTask()` de l'OS.

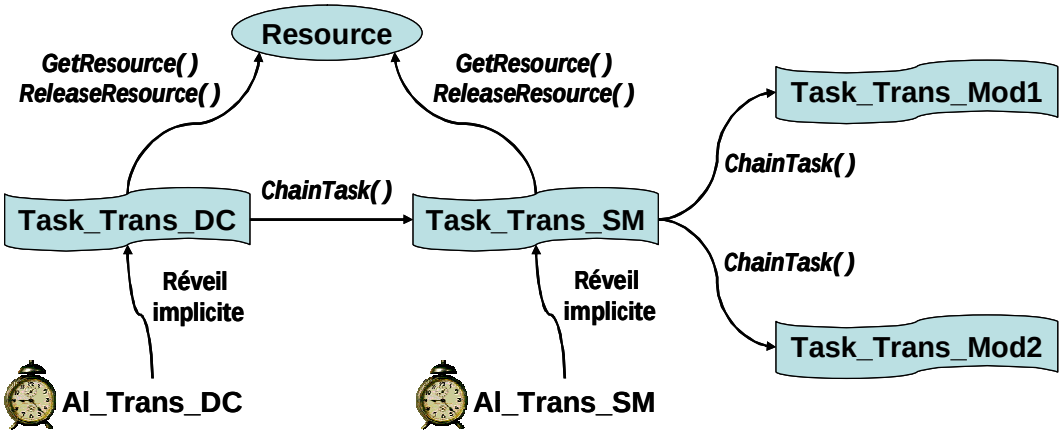


Figure 34 : Schéma comportemental de l'application « transmission de couple »

VI.6.1 Matrice de traçabilité

Une partie de la matrice de traçabilité a été déterminée dans le chapitre précédent, pour l'exigence E4. Elle est récapitulée ci-dessous. L'entrée « 8- Instrumentation pour injection de fautes » est celle que nous cherchons à déduire.

0- Exigence de sûreté	**1- Type de Défaillance spécifique**
E4 : Le passage intempestif «du mode 1 au mode 2» ou «du mode 2 au mode 1» est une défaillance.	**D4 : Flot de contrôle :** Transition non voulue
2- Assertions exécutables	
Assertion A4 (en pseudo-code) : Cas 1 : Activation `Task_Trans_SM` et source différente de `Task_Trans_DC` et `Al_Trans_SM` ; Cas 2 : Activations successives de `Task_Trans_SM` par `Task_Trans_DC` inférieures à 1s ;	
3- Stratégie de Détection d'erreur	
L'assertion exprime les conditions dans lesquelles un évènement de contrôle doit se produire. Les éléments de la **condition** et **l'évènement de contrôle** considéré sont donc tracés. Si l'assertion n'est pas vérifiée, une erreur est notifiée.	

4- Sondes logicielles
Condition : capturer l'identifiant de `Task_Trans_SM, Task_Trans_DC` avant leur exécution **Evènement de contrôle** : capturer l'instant d'appel et le paramètre (tâche activée) de `ChainTask()`, capturer l'état de l'alarme `Al_Trans_SM` avant l'exécution de `Task_Trans_SM`
5- Stratégie de Recouvrement d'erreur
Erreur notifiée : La requête de transition est erronée. Recouvrement par compensation : Inhiber la transition en cours.
6- Actionneurs logiciels
Inhiber la transition en cours. Une détection d'erreur de déclenchement de `Task_Trans_SM` a été faite donc cette tâche doit être terminée.
7- Stratégie de Vérification
L'assertion exprime les conditions dans lesquelles un évènement de contrôle doit se produire. Il s'agit de corrompre les éléments de la **condition** ou **l'évènement de contrôle** considéré.

D'après la stratégie de vérification, il s'agit de corrompre les éléments de la condition ou l'évènement de contrôle considéré. En l'occurrence, d'après l'entrée « 4-sondes logicielles », les évènements de contrôle considérés sont :

- l'appel au service `ChainTask()` (condition : par la tâche `Task_Trans_DC` pour activer `Task_Trans_SM`)
- le réveil de `Task_Trans_SM` par son alarme `Al_Trans_SM`.

Le problème est de déterminer comment corrompre ces éléments pour obtenir la défaillance D4, à savoir une transition non voulue. En d'autres termes, il s'agit de provoquer une suractivation de la tâche `Task_Trans_SM`. Pour cela, le service **`ActivateTask()`** de l'OS permet d'activer la tâche `Task_Trans_SM`. Cela perturbe alors l'évènement de contrôle `ChainTask()`. La tâche `Task_Trans_DC` est normalement responsable de l'activation de `Task_Trans_SM`. Nous pouvons perturber cette condition, en utilisant `ActivateTask()` en dehors de la tâche `Task_Trans_DC`.

Il est important de souligner ici que dans la réalité, cette mutation peut correspondre à une erreur de génération de code liée à un défaut de conception d'outils de développement COTS ou encore à un mauvais paramétrage de cet outil. Une transition non voulue peut aussi être due à une faute transitoire du matériel ou encore à une erreur résiduelle du support d'exécution. Dans notre cas, il s'agit du moyen d'instrumentation le plus simple pour créer ce type d'erreur.

La 8ème colonne de la matrice de traçabilité peut alors être remplie, et le protocole d'injection de fautes décrit en section VI.3 peut être appliqué.

8- Instrumentation pour injection de fautes
Service : `ActivateTask`

VI.6.2 Campagne de tests et exemple de résultats

Pour une cible de type AUTOSAR, le flot de contrôle peut être intercepté au niveau des *taskbodies*, scripts des tâches contenant les appels de fonctions applicatives *runnables*. Une expérience d'injection de fautes correspond à l'ajout d'un appel au service `ActivateTask()`, dans un *taskbody*.

L'application que nous avons testée n'est pas exactement celle décrite dans l'étude de cas (chapitre V). Elle ne fait pas cohabiter les autres composants applicatifs (climatisation, airbag). Seule la partie du contrôle critique qui nous intéresse de l'application « transmission de couple » est implémentée. Il y a donc 4 *taskbodies*, correspondant aux 4 tâches (`Task_Trans_SM`, `Task_Trans_DC`, `Task_Trans_Mod1 et Task_Trans_Mod2`). Le nombre de ligne de code dans l'ensemble des *taskbodies* est 28, soit 28 mutants. Il y a donc 28 possibilités d'injection de fautes. Comme `ActivateTask()`est le seul type de code à insérer. Le nombre d'expériences d'injection de fautes pour l'exigence E4 s'élève à 28 tests par campagne.

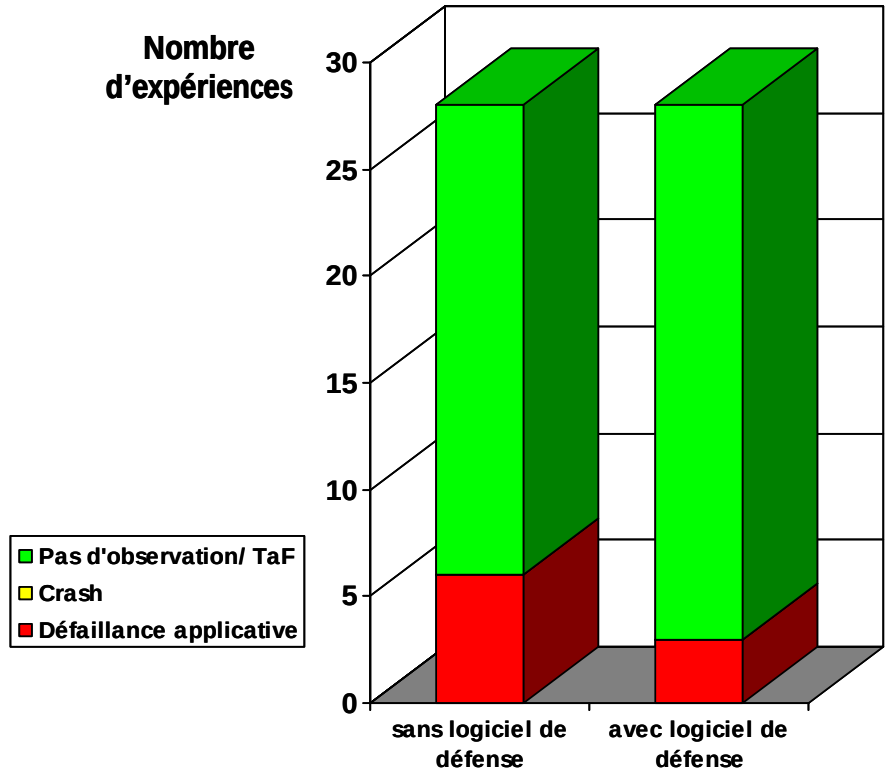


Figure 35 : Injection de fautes sur cible avec et sans logiciel de défense pour E4

Le résultat des campagnes d'injection de fautes est donné par la figure 35. Deux campagnes sont comparées. La première a été réalisée sur l'application originale non

empaquetée par le logiciel de défense. La deuxième campagne a ciblé l'application protégée par un logiciel de défense dédié à l'exigence E4. Les histogrammes montrent la proportion de défaillances applicatives (en rouge), de tolérance aux fautes (en vert), et de crash (en jaune).

Il apparaît clairement que l'injection de fautes ne conduit pas nécessairement à une défaillance. Déjà sans mécanismes de protection, aucune observation n'est détectée pour 22 expériences sur 28. Cette constatation mène à plusieurs conclusions : 1) l'application considérée est robuste à l'activation intempestive de tâche réalisée ; 2) notre technique d'injection de fautes ne permet pas de simuler suffisamment les défaillances considérées pour cette application ; 3) notre méthode d'observation manuelle n'est peut être pas assez précise (nous avons tracé uniquement les changements de modes et non le temps d'exécution des traitements).

Pour l'application sans logiciel de défense, il y a 6 tests qui ont conduit à une défaillance applicative. Pour l'application, dans laquelle nous avons ajouté nos mécanismes de défense, la tolérance aux fautes a été obtenue pour 3 tests, et 3 autres tests conduisent encore à une défaillance applicative. Ce résultat peut vraisemblablement être amélioré en analysant plus finement ces 3 derniers tests, et en raffinant l'assertion exécutable (A4). Malgré tout **l'amélioration de la tolérance aux fautes** apportée par nos mécanismes de défense a été montrée, et la faisabilité de la solution a été prouvée par l'exemple en déployant la totalité de la méthodologie de l'analyse à la validation.

VI.7 Résultats d'expériences et analyses préliminaires

Cette section présente les résultats d'autres expériences d'injection de fautes que nous avons menées, en simulation sur Unix, en suivant le même protocole de tests (cf. section VI.3). L'environnement de test est le compilateur « gcc », et les sorties de l'application sont envoyées sur un terminal.

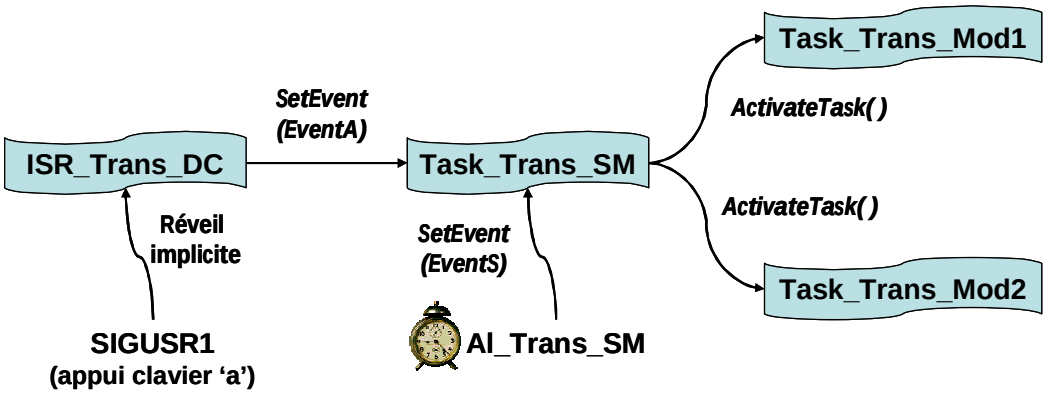


Figure 36 : Schéma comportemental de l'application en simulation

L'application étudiée est toujours la « transmission de couple » compatible AUTOSAR. Cependant, nous avons conçu cette application différemment de l'implémentation de la figure 34. La diversification se situe au niveau du RTE, et plus précisément des scripts de tâches *taskbodies*. En effet, il est possible d'utiliser différents types d'appels système et objets OS (tâche, interruption, évènement), pour

déclencher une transition. Ici l'application (cf. Fig. 36) utilise des évènements `EventA` et `EventS` (objets de synchronisation gérés par l'OS) et une interruption `ISR_Trans_DC`, ce qui n'était pas le cas précédemment.

Nous donnons les résultats détaillés pour une implémentation en simulation, sans mécanismes de protection et avec. Comme la méthodologie a déjà été illustrée depuis l'analyse à la validation, nous donnerons simplement l'instrumentation déterminée pour l'injection de fautes et les résultats, pour chaque expérience qui suit. Le tableau suivant rappelle que les exigences de sûreté nous surveillons sont E3 et E4, et les services utilisés pour l'injection de fautes dans chaque cas.

0- Exigence de sûreté	**1- Type de Défaillance spécifique**
E3 : Le blocage en mode 1 alors que le système doit passer en mode 2 (idem pour mode 2) est une défaillance.	**D3 : Flot de contrôle :** Transition invalide
E4 : Le passage intempestif «du mode 1 au mode 2» ou «du mode 2 au mode 1» est une défaillance.	**D4 : Flot de contrôle :** Transition non voulue
8- Instrumentation pour injection de fautes	
Services pour E3 : `SetEvent(EventS), ClearEvent(EventA), ClearEvent(EventS), DisableAllInterrupts, SuspendAllInterrupts, SuspendOSInterrupts`	
Services pour E4 : `SetEvent(EventA)`	

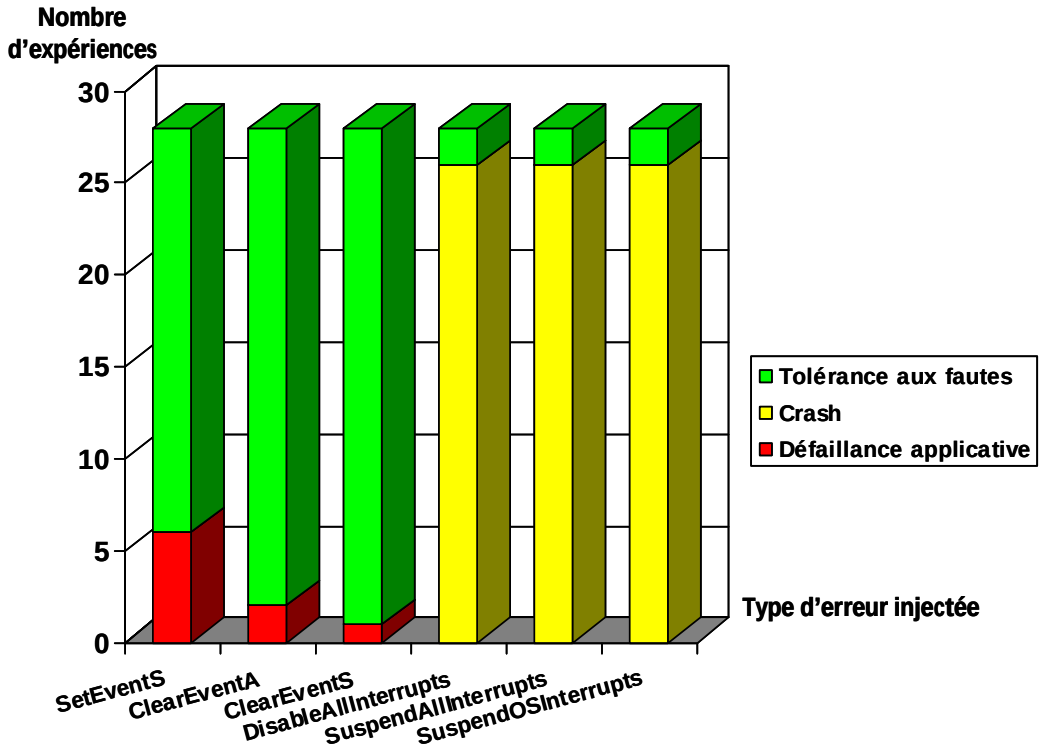


Figure 37 : Injection de fautes ciblant E3 sur application sans logiciel de défense

La figure 37 montre le résultat de 6 campagnes d'injection de fautes de types différents, de 28 expériences chacune, concernant l'exigence E3. `SetEvent(EventS)` réveille une tâche si elle attendait l'évènement `EventS`. `ClearEvent(EventA)` permet d'effacer l'évènement `EventA` du masque d'évènements d'une tâche. Les services `DisableAllInterrupts()` et `SuspendAllInterrupts()` désactivent toutes les interruptions quand le matériel le permet. `SuspendOSInterrupts()` ne désactive que les interruptions autorisées à faire des appels système.

Les résultats d'injection de fautes sur l'application non protégée sont intéressants. Ils peuvent être utilisés comme mesure de robustesse de l'application.

Pour les services `SetEvent(EventS)`, `ClearEvent(EventA)` et `ClearEvent(EventS)`, l'injection de fautes a relativement peu d'impact. Avec `SetEvent(EventS)`, on a 6 cas de défaillances sur 28. Avec `ClearEvent(EventA)`, on a 2 cas de défaillances sur 28. Avec `ClearEvent(EventS)`, on a 1 cas de défaillance sur 28. Cela fait en tout 9 cas de défaillances applicatives sur 84 expériences, trois séries de 28 expériences, soit 11% de défaillances applicatives et 89% de non-observation. Les commentaires des résultats de la figure 35, section VI.6.2, peuvent être repris.

L'injection de fautes à l'aide des services `DisableAllInterrupts()`, `SuspendAllInterrupts()`et `SuspendOSInterrupts()` provoque des erreurs de segmentation. Ces services sont en effets utilisés par paire avec un service qui ré-autorise les interruptions. Pendant la phase de conception et d'intégration du logiciel, une utilisation malencontreuse d'une désactivation des interruptions sera vraisemblablement rapidement détectée puisqu'elle provoque un *crash* de l'application. La question qui se pose est l'intérêt de n'insérer qu'un seul appel système pour chaque expérience d'injection de fautes. Dans le cas considéré, il faudrait insérer pour chaque expérience deux appels systèmes : la désactivation et la réactivation des interruptions. Le protocole d'injection de fautes consisterait alors à insérer une ou plusieurs lignes de code, selon la perturbation que nous voulons provoquer.

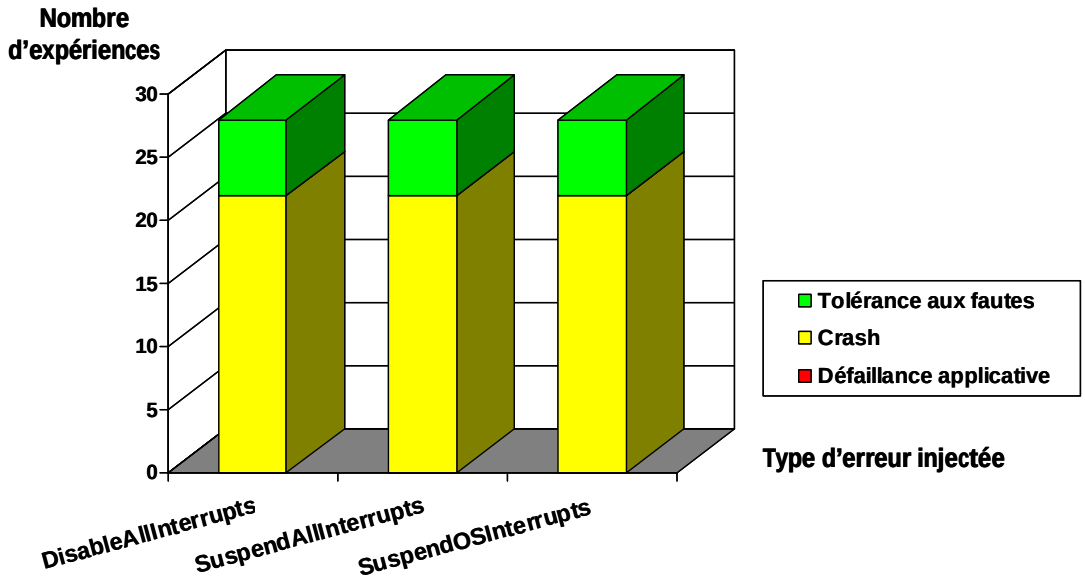


Figure 38 : Injection de fautes ciblant E3 avec nouvelle version d'OS

En rapportant malgré tout le problème de crash à l'équipe de développement de l'OS Trampoline que nous utilisons, une amélioration de l'OS a été faite au niveau de la terminaison des tâches sans l'utilisateur, de la détection des interruptions bloquées, et la remise en service. Nous avons alors effectué trois nouvelles campagnes d'injection de fautes (cf. Fig.38), avec la nouvelle version d'OS, en utilisant les trois types d'injection de fautes qui provoquaient les *crashs* (`DisableAllInterrupts`, `SuspendAllInterrupts` et `SuspendOSInterrupts`). La figure 38 montre qu'il y a 21 cas de crash (en jaune) et 7 cas de tolérance aux fautes (en vert) par type d'injection de fautes. Par comparaison, avec l'ancienne version d'OS (cf. Fig.37), les trois derniers bâtons représentent les injections de type `DisableAllInterrupts`, `SuspendAllInterrupts` et `SuspendOSInterrupts`. Le résultat était de 26 cas de crash (en jaune) et 2 cas de tolérance aux fautes (en vert) par type d'injection de fautes. La nouvelle version permet donc à l'OS de détecter automatiquement l'erreur injectée (en tout 15 cas de tolérance aux fautes supplémentaires) et inhiber le code inséré en l'ignorant, donc tolérer l'erreur. Ces tests ont permis d'améliorer la robustesse de l'OS.

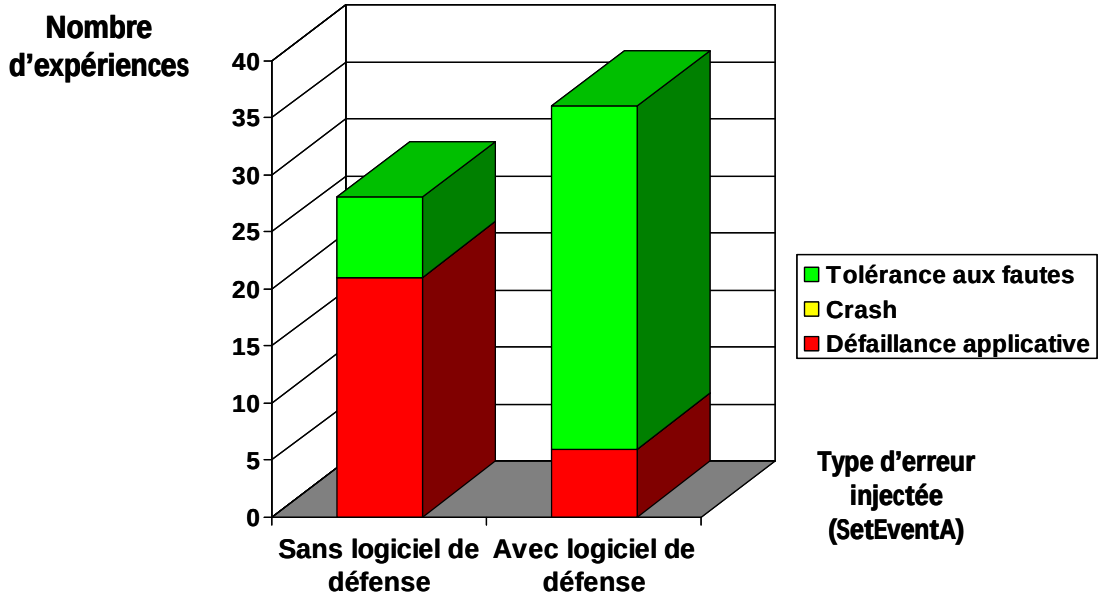


Figure 39 : Injection de fautes en simulation sans et avec logiciel de défense pour E4

Les deux campagnes d'injection de fautes de la figure 39 utilisent `SetEvent(EventA)` pour perturber l'application vérifient l'exigence E4. La première campagne cible l'application sans logiciel de défense, et la deuxième cible l'application avec logiciel de défense dédié à l'exigence E4.

L'augmentation du nombre d'expériences de 28 (sans logiciel de défense) à 36 (avec logiciel de défense) est due à l'augmentation du nombre de lignes dans les *taskbodies*, à cause de l'introduction de *hooks* pour la protection de l'application. La série de 36 expériences correspond à une application protégée par nos mécanismes de tolérance aux fautes. Nous avons voulu traiter ici l'ajout de ces *hooks* comme des lignes de codes fonctionnelles supplémentaires à tester, ce que nous n'avons pas fait pour les expériences sur microcontrôleur.

On constate sur la figure 39 que les injections de fautes avec `SetEvent(EventA)` n'ont pas provoqué de *crash*.

Sans mécanismes de tolérance aux fautes, il y a eu 21 cas de défaillance applicative ; tandis que la présence de mécanismes a permis de faire diminuer le nombre de cas de défaillance à 6. Ces expériences faites en simulation sont comparables aux injections de fautes effectuées précédemment sur microcontrôleur présentés en section VI.6, figure 35, pour l'exigence E4 également. L'amélioration de la robustesse de l'application en simulation est plus importante ici que pour l'application sur microcontrôleur. Ceci montre que le résultat d'amélioration pour une exigence de sûreté donnée dépend fortement de l'implémentation. S'il y avait un choix d'implémentation à faire, il serait difficile dans notre cas. L'utilisation des évènements (selon nos tests) est a priori moins robuste que les services d'activation de tâche : 21 défaillances contre 6 défaillances pour une application non protégée. La protection semble plus efficace sur la gestion des évènements. Malgré tout il reste 6 cas de défaillances à revoir, contre 3 cas seulement avec les services d'activation de tâche.

La conclusion, que nous pouvons tirer de ces expériences d'injection de fautes, est que notre technique peut être utilisée à la fois pour la vérification de notre logiciel de défense, et pour la robustesse. Elle donne au moins un indicateur de performance sous forme de distribution de modes de défaillances, spécifique à une application donnée. La détermination des erreurs à injecter peut être approfondie. Nous avons seulement voulu proposer une approche de tests simple et systématique, qui finalise l'illustration de la démarche proposée dans la thèse.

Conclusion du Chapitre VI

Une méthode de vérification de l'efficacité du logiciel de défense est réalisée par injection de fautes, en s'appuyant sur la matrice de traçabilité des exigences. Le principe consiste à provoquer les défaillances applicatives, ciblé par le logiciel de défense, par des insertions de code. Plus précisément, des appels systèmes sont ajoutés dans le code glue entre les modules applicatifs et le logiciel de base, pour corrompre les flots de données et de contrôle. Des techniques d'automatisation ont commencé à être étudiées en simulation et sur carte.

Des campagnes d'injection de fautes ont été réalisés manuellement en simulation et sur carte, sur des applications avec ou sans logiciel de défense. Elles montrent globalement l'amélioration de robustesse apportée par le logiciel de défense, pour les prototypes réalisés. L'évaluation quantitative de cette amélioration dépend fortement de l'implémentation de la cible et du logiciel de défense : sur l'ensemble des campagnes réalisées pendant la thèse, un logiciel de défense permet d'obtenir 40 à 70% de tests supplémentaires qui tolèrent les fautes, pour une exigence de sûreté donnée, par rapport à une cible non protégée. Des voies d'amélioration du protocole d'injection d'erreurs sont discutées sur la manière de saboter le code, à partir de résultats d'injection de fautes obtenus.

Dans l'état actuel de la technique d'injection de fautes utilisée, la première interprétation des campagnes d'injection est l'identification des services du logiciel de base qui sont le plus susceptible de représenter des erreurs de programmation à l'intégration (en particulier configuration d'outils et défauts du générateur de code automatique).

D'autres conclusions peuvent être tirées au cas par cas, en fonction du type et du moment d'insertion de code. Les expériences pour lesquelles une défaillance est observée doivent être analysées en détail. Cette analyse peut révéler une faute de développement du logiciel cible, ou bien une faiblesse du logiciel de défense. Dans ce dernier cas, il faut reprendre l'analyse des propriétés et donc la démarche globale dans son intégralité.

Conclusion générale

Conclusion générale

La **robustesse du logiciel embarqué automobile**, face à l'occurrence inévitable de fautes physiques et logicielles, est une problématique accentuée par : la complexité croissante du logiciel (architecture multicouche), l'utilisation de composants sur étagère, et la cohabitation d'applications de criticités différentes sur le même support d'exécution.

Pour améliorer cette robustesse, nous avons proposé et montré l'intérêt d'une **conception « réflexive » du logiciel**. Le comportement de référence du système est déterminé par les exigences de sûreté des applications critiques, traduites en assertions exécutables. La stratégie de tolérance aux fautes utilisée repose principalement sur la vérification de ces assertions. L'architecture réflexive en elle-même est une organisation permettant d'externaliser les services de traitement d'erreur et donc faciliter leur développement et leur évolution.

Les services d'enregistrement des traces d'exécution, les services de détection d'erreur (vérification d'assertion) et les services de commande de recouvrement d'erreur constitue le « **logiciel de défense** ». Il est idéalement sécurisé par des mécanismes complémentaires de protection matérielle (MPU, redondance matérielle, etc.). Le logiciel de défense est séparé du logiciel fonctionnel par une interface d'instrumentation qui lui permet d'interagir sur les flots de contrôle et de données à l'exécution. Cette **instrumentation** est constituée de services fonctionnels du logiciel de base (exécutif temps réel et communication), jouant le rôle de sondes et de capteurs logiciels. L'exécution des services du logiciel de défense est déclenchée sélectivement par des fonctions *hooks*, appelées à des instants prédéfinis (ex : début de tâche, après une écriture de donnée), selon les assertions à vérifier. Enfin, l'interfaçage (sondes et actionneurs logiciels) avec le support d'exécution favorise la réutilisation et le portage des mécanismes formant le logiciel de défense sur des plateformes logicielles provenant de différentes sources.

Contribution scientifique de la thèse

La contribution scientifique du travail est d'avoir regroupé des principes et mécanismes classiques pour la tolérance aux fautes, dans une architecture cohérente et adaptée au contexte automobile.

Les méthodes de conception diversifiée (à partir des exigences fonctionnelles d'une part et des exigences de sûreté d'autre part) et d'unité de contrôle externe (logiciel de défense) sont combinées pour traiter en particulier les fautes de conception logicielles. Les fautes physiques et les fautes logicielles de programmation peuvent également être détectées par les vérifications d'assertion.

L'organisation du logiciel en plusieurs couches d'abstraction limite les moyens de détection et de recouvrement d'erreurs dans chaque couche respective. En effet, les applications possèdent des stratégies élaborées de gestion de défaillances (inhibition

de fonctions applicatives), mais leurs actions sur le flot d'exécution sont généralement limitées (terminaison de tâche, réinitialisation). Inversement, un exécutif maîtrise complètement le flot d'exécution mais n'a pas la connaissance des stratégies de recouvrement spécifique. Notre solution exploite donc conjointement les avantages de chaque couche logicielle, pour une meilleure gestion de la tolérance aux fautes, c'est-à-dire assurer si possible la continuité du service critique, ou une dégradation acceptable, en cas d'erreur.

Notre approche favorise l'amélioration de l'observabilité et de la commandabilité des composants sur étagère à surveiller. De manière générale, les éléments des flots critiques de contrôle (transitions et séquences de traitements critiques) et de données (variables critiques) doivent être accessibles, autrement aucun contrôle n'est possible selon notre méthode. Pour que l'instrumentation nécessaire au logiciel de défense ne soit pas intrusive, il est préférable que le composant considéré possède une interface vers la sonde ou l'actionneur logiciel requis.

Le logiciel de défense est flexible, de manière à répondre à la diversité des applications automobiles. La granularité choisie pour étudier de manière générique les applications considère les échanges de données et les actions de contrôle à l'exécution. Ainsi, il est possible de définir, pour chaque type de défaillances applicatives, des stratégies génériques de détection, de recouvrement d'erreur, et de vérification de fonctionnement du logiciel de défense. L'utilisation d'une matrice de traçabilité des exigences de sûreté a le mérite de permettre le suivi du développement de chaque partie du logiciel de défense, spécifique à chaque exigence.

Contribution industrielle de la thèse

La contribution industrielle du travail est principalement l'analyse des avantages et des dangers liés à l'adoption des nouveaux standards AUTOSAR et ISO26262, ainsi que la proposition d'une solution technique et méthodologique pour la robustesse logicielle.

Le standard d'architecture logicielle AUTOSAR favorise la conception d'applications complexes. Il véhicule l'image du logiciel embarqué comme un assemblage de composants sur étagère aisément interopérables. Du point de vue de la sûreté de fonctionnement, les questions se posent autour des conséquences : de l'hétérogénéité de robustesse et de criticité des composants, de la fiabilité des premières générations d'outils non-certifiés de configuration et de génération de code AUTOSAR, ainsi que le rôle primordial de l'intégration logicielle. Pour répondre à ces risques, la phase « AUTOSAR 4.0 » en particulier renforce l'introduction de concepts et mécanismes de protection. Cependant, cette richesse de moyens qu'AUTOSAR met à disposition de l'utilisateur pour gérer la détection et le recouvrement d'erreurs peut également être problématique. Sans règles ou stratégies de conception, certains mécanismes peuvent être peu ou mal employés. L'architecture de tolérance aux fautes de notre étude de cas montre une utilisation cohérente de certains mécanismes AUTOSAR : les « OS-applications », les « *trusted functions* » (pour la protection mémoire), et les « *hooks* » (classiquement utilisés pour le débogage).

Le standard de sécurité fonctionnelle ISO26262 recommande des méthodes de détection et de recouvrement d'erreurs en fonction du niveau de criticité de l'application. De même que pour les mécanismes de protection AUTOSAR, le concepteur possède une certaine liberté de choisir la combinaison de mécanismes de sûreté qu'il va introduire dans le système. Nous avons choisi une combinaison de mécanismes pour les applications critiques (ASIL C et D), permettant de traiter à la fois les fautes physiques et les fautes logicielles.

En termes de qualité, la solution technique proposée permet d'améliorer la robustesse du logiciel et donc la qualité du produit. Concernant les indicateurs industriels de coût et de délai, même si la sécurité n'a pas de prix, elle a toujours un coût. La tolérance aux fautes peut avoir un impact plus ou moins fort sur les performances mémoire et/ou temporelles du système, selon le nombre d'exigences de sûreté à surveiller. Cependant, dans notre étude, l'exploitation des standards permet de réduire les coûts et délais de développement. Notre méthodologie revient à donner des règles de conception d'un logiciel robuste, à la fois compatible à ISO26262 et AUTOSAR.

La mise en œuvre industrielle de notre solution serait le fruit d'une collaboration entre :

- Les intégrateurs logiciels, qui seraient responsable de l'implémentation du logiciel de défense ;
- Les concepteurs d'application, qui donnent les exigences de sûreté de haut niveau ;
- Les fournisseurs de composants logiciels applicatifs ou du support d'exécution, qui doivent communiquer les informations d'implémentation des flots critiques de contrôle.

L'approche proposée ne signifie pas que les fournisseurs doivent nécessairement fournir leurs composants en « boîte blanche ». L'intérêt de la méthode est de se satisfaire d'une « boite grise » par l'introduction de capteurs et actionneurs logiciels à la demande. Le constructeur automobile coordonnerait alors les étapes de construction de la matrice de traçabilité des exigences de sûreté, et serait éventuellement responsable d'une ou plusieurs d'entre elles : analyse, conception et/ou validation.

Le message de la thèse est donc que la révolution architecturale des systèmes embarqués automobiles implique un changement de stratégie dans la production de systèmes sûrs. L'implication de l'intégrateur du logiciel d'un calculateur (constructeur ou équipementier de premier rang) est à ce titre fondamentale. Cette tendance architecturale à la cohabitation d'applications de criticité non-homogène sur une même plateforme matérielle est incontournable. L'adjonction au cas par cas d'un logiciel de défense externe spécialisable est une réponse à l'intégration de systèmes complexes ayant des objectifs en matière de sûreté de fonctionnement. En s'appuyant sur des concepts théoriques bien établis aujourd'hui, notre proposition va dans ce sens en profitant et en étendant les mécanismes d'observabilité et de commande disponibles dans les supports d'exécution.

Bibliographie

Bibliographie

[Ait-Ameur *et al.* 2002] Y. Ait-Ameur, B. d'Ausbourg, F. Boniol, R. Delmas, V. Wiels, *A component based methodology for description of complex systems, an application to avionics systems*, in the 3rd European Systems Engineering Conference (EuSEC'2002), Toulouse, France, May 2002.

[Anderson & Lee 1981] T. Anderson, P.A. Lee, *Fault Tolerance, Principles and practice*. Prentice Hall, 1981.

[Anderson & Metze 1974] D.A. Anderson, G. Metze, *Design of totally self-checking check circuits for M-out-of-N codes*. IEEE Transactions on Computers, vol. 22 (3): 263-269, 1974.

[Arlat *et al.* 1990] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, D. Powell, *Fault Injection for Dependability Validation, A methodology and some Applications.* IEEE Transactions on Software Engineering. 16 (2): 166-182, 1990.

[Arlat *et al.* 2000] J. Arlat (Editeur), J.P.Blanquart, T.Boyer, Y.Crouzet, M.H.Durand, J.C.Fabre, M.Founau, M.Kaaniche, K.Kanoun, P.Le Meur, C.Mazet, D.Powell, F.Scheerens, P.Thevenod-Fosse, H.Waeselynck, *Composants logiciels et sûreté de fonctionnement, integration de COTS*. Hermes, 2000.

[Arlat *et al.* 2002] J.Arlat, J-C.Fabre, M.Rodriguez, F.Salles. *Dependability of COTS Microkernel-Based Systems*. IEEE Transactions on Computers, 51(2): 138-163, Feb 2002.

[AUTOSAR] AUTomotive Open Standard ARchitecture, http://www.autosar.org

[AUTOSAR FIM] AUTOSAR. *Specification of Function Inhibition Manager, V1.2.0*, R3.0, Technical Report, Dec 2007.

[AUTOSAR OS] AUTOSAR. *Specification of Operating System, V3.0.0, R3.0*, Technical Report, Dec 2007.

[AUTOSAR RTE] AUTOSAR. *Specification of RTE, V2.0.0, R3.0*, Technical Report, Dec 2007.

[AUTOSAR SWC] AUTOSAR. *Software Component Template, V3.0.0, R3.0*, Technical Report, Nov 2007.

[AUTOSAR VFB] AUTOSAR. *Specification of the Virtual Functional Bus, V1.0.0, R3.0*, Technical Report, Nov 2007.

[AUTOSAR Watchdog] AUTOSAR. *Specification of Watchdog Manager, V1.2.0, R3.0*, Technical Report, Dec 2007.

[Avizienis & *al.* 2004] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr., *Basic Concepts and Taxonomy of Dependable and Secure Computing*, IEEE Trans. Dependable Secure Computing, vol. 1, pp. 11--33, 2004.

[Avizienis *& al.* 1985] A. Avizienis, P. Gunningberg, J.P.J. Kelly, L. Strigini, P.J. Traverse, K.S. Tso, U. Voges, *The UCLA DEDIX System: a distributed Testbed for multiple-version software*, in the 15th IEEE International Symposium on Fault Tolerant Computing (FTCS'1985), Ann Arbor, USA, pp. 126-134, Jun. 1985.

[Banâtre *& al.* 1987] J.P. Banâtre, M. Bañarte, G. Muller, F. Ployette, *Quelques aspects du système GOTHIC*. Tech. et Sc. Informatiques, vol. 6 (2): 170-174, 1987.

[Bechennec *& al.* 2006] J.L. Béchennec, M. Briday, S. Faucou, Y. Trinquet, *Trampoline : An Open Source Implementation of the OSEK/VDX RTOS Specification*, in Proc. of the Int. IEEE Conf. on Emerging Technologies & Factory Automation (ETFA'2006), Prague, Czech Republic, pp.62--69, 2006.
– see : http://trampoline.rts-software.org –

[Bruneton *& al.* 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: *The Fractal Component Model and Its Support in Java. Software Practice and Experience,* Softw. Pract. Exper. vol. 36 (11-12): 1257-1284, 2006.

[Burns & Wellings 1995] A. Burns, A.J. Wellings. *Engineering a hard real-time system: from theory to practice*. Softw. Pract. Exper. vol. 25 (7): 705-726, 1995.

[Carreira & al. 1998] J. Carreira, H. Madeira, J.G. Silva. *Xception : a technique for the experimental evaluation of dependability in modern computers*. IEEE Transaction on Software Engineering, 24(2): 125-136, 1998.

[Chen & al. 2007] X. Chen, J. Feng, M. Hiller, V. Lauer, *Application of Software Watchdog as a Dependability Software Service for Automotive Safety Relevant Systems*. in Proc. of the Int. IEEE Conf. on Dependable Systems and Networks (DSN'2007), Edinburgh, UK, pp. 618-624, Jun. 2007.

[Chen & Hsiao 1984] C.L. Chen, M.Y. Hsiao, *Error correcting codes for semiconductor memory applications: a state of the art review*. IBM J. Research and Development, vol. 28 (2):124-134, 1984.

[Chérèque & al. 1992] M. Chérèque, D. Powell, P. Reynier, J.L. Richier, J. Voiron, *Active replication in Delta-4*, in the 22th IEEE International Symposium on Fault Tolerant Computing (FTCS'1992), Boston, USA, pp. 28-37, Jul. 1992.

[Chillarege *& al.* 1992] R. Chillarege, IS. Bhandari, JK. Chaar, MJ. Halliday, DS. Moebus, BK. Ray, and MY. Wong, "Orthogonal defect classification-a concept for in-process measurements", IEEE Transaction on Software Engineering, 18(11):943–956, 1992.

[CHORUS 1997] Chorus Systems. *CHORUS/ClassiX release 3, Technical Overview*. Technical report, no. CS/TR-96-119.12, 1997

[Cooling 2003] J. Cooling. *Software engineering for Real-time Systems.* Addison Wesley, 2003.

[Costa & al. 2000] D. Costa, T. Rilho, H. Madeira, *Joint evaluation of performance and robustness of a COTS DBMS through fault-injection,* in Proc. of the Int. IEEE Conf. on Dependable Systems and Networks (DSN'2000), New York, USA, pp 251--260, Jun. 2000.

[Coulson *& al.* 2004] Coulson, G., Grace, P., Blair, G.S., Mathy, L., Duce, D., Cooper, C., Yeung, W.K., Cai, W.: *Towards a Component-based Middleware Architecture for Flexible and Reconfigurable Grid Computing.* in Workshop on Emerging Technologies for Next generation Grid (ETNGRID), Italy, Jun. 2004.

[Deswarte & Lavictoire 1975] Y. Deswarte, J. Lavictoire, *MARIGNAN : an intermittent failure correction method,* in the 5th IEEE International Symposium on Fault Tolerant Computing (FTCS'1975), Paris, France, pp. 191-195, Apr. 1975.

[EASIS] Electronic Architecture and System engineering for Integrated Safety systems, http://www.easis-online.org

[EASIS FMF] EASIS, J.M. Dressler, *Fault Management Framework.* Technical Report, 2006.

[EASIS FT] J.Böhm, M.Menzel, X.Chen, J-M.Dressler, T.Eymann, M.Hiller, T.Kimmeskamp, V.Quenda. *Description of Fault Types for EASIS V2.0.* Technical report, Jun 2005.

[EGAS 2004] EGAS. *Standardized E-Gas monitoring concept for engine management systems of gasoline and diesel engines V2.0.* Technical report, 2004.

[EMS2010 2005] A.Belaid, A.Guillemin, J.Regnard, S.Rombauts. *EMS2010 Architecture Diagnostics & Modes Dégradés, l'indicateur de validité.* Technical report, Jun. 2005.

[EMS201 2006] RENAULT. *EMS2010 Coding Rules 1.6.1,* Technical report, 2006.

[Feldmeier 1995] Feldmeier, D. C. 1995. *Fast software implementation of error detection codes. IEEE/ACM Trans. Netw.* pp 640--651, 1995.

[Garlan & Shaw 1994] D. Garlan, M. Shaw, *An introduction to software architecture*. CMU Software Engineering Institute, Technical Report, CMU/SEI-94-TR-21, ESC-TR-94-21, 1994.

[Horning *& al.* 1974] J.J. Horning, H.C. Lauer, P.M. Meliar-Smith, B. Randell, *A program structure for error detection and recovery.* Lectures Notes in Computer Science, vol.16, pp. 172--187, Springer-Verlag, 1974.

[IEC 1998] International Electrotechnical Commission. *Functional safety of electrical/electronic/programmable electronic safety related systems.* Technical report, 1998.

[IEC 2002] International Electrotechnical Commission. *Functional Safety and IEC 61508, a basic guide*. Technical report, 2002.

[ISO26262] ISO/DIS 26262-6: *Road vehicles, Functional safety, Part 6: Product development: software level.* Technical report, 2009.

[ISO26262-1] ISO/DIS 26262-1: *Road vehicles, Functional safety, Part 1: Vocabulary.* Technical report, 2009.

[ISO26262-5] ISO/DIS 26262-5: *Road vehicles, Functional safety, Part 5: Product development: hardware level*. Technical report, 2009.

[Kantz & Koza 1995] H. Kantz, C. Koza, *The ELEKTRA railway Signaling-System: Field Experience with an Actively Replicated System with Diversity*, in the 25th IEEE International Symposium on Fault Tolerant Computing (FTCS'1995), Pasadena, California, pp. 453--458, Jun. 1995.

[Kanawati *& al.* 1992] G.A. Kawanati, N.A. Kawanati, J.A. Abraham. *FERRARI : a tool for the validation of system dependability properties*, in the 22th IEEE International Symposium on Fault Tolerant Computing (FTCS'1992), Boston, USA, pp 336--344, Jul. 1992.

[Karlsson *& al.* 1995] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger. *Application of three physical fault injection techniques to the experimental assessment of the MARS architecture*, in Proc. of the 6th Int. IFIP Conf. on Dependable Computing for Critical Applications (DCCA'1995), Urbana-Champaign, USA, pp 267--287, Sept. 1995.

[Kiczales *& al.* 1991] G. Kiczales, J. Rivières, D.G. Bobrow. *The art of the Metaobject Protocol*. MIT Press, ISBN-13: 978-0-262-61074-2, 1991.

[Kiczales *& al.* 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, J. Irwin. *Aspect-Oriented Programming*. in Proc. of the 11th European Conf. on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, pp 220--242, Jun. 1997.

[Koo & Toueg 1987] R. Koo, S. Toueg, *Checkpointing and rollback recovery for distributed systems*, IEEE Trans. on Software Engineering, vol. SE-13, no. 1, pp. 23-31, 1987.

[Kon *& al.* 2000] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L.C. Magalhaes, R.H. Campbell, *Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB*, in the Int. IFIP/ACM Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), New York, USA, pp 121--143, Apr. 2000.

[Kon *& al.* 2002] F. Kon, D. Costa, G. Blair, R.H. Campbell. *The case for reflective middleware*. Communications of the ACM, 45: 33-38, Jun 2002.

[Koopman & DeVale 1999] P.J. Koopman, J. DeVale. *Comparing the robustness of POSIX operating systems*, in the 29th IEEE International Symposium on Fault Tolerant Computing (FTCS'1999), Los Alamitos, USA, pp 30--37, 1999.

[Landwehr *& al.* 1993] C.E.Landwehr, A.R.Bull, J.P.MCDermott, W.S.Choi. A *Taxonomy of Computer Program Security Flaws, with Examples.* NRL Report 9591, Naval Research Laboratory, Nov. 1993.

[Laprie 1996] J-C. Laprie. *Guide de la sûreté de fonctionnement.* Cépaduès,1996.

[Leaphart *& al.* 2005] E.G. Leaphart, B.J. Czerny, J.G. D'Ambrosio, C.L. Denliger, D. Littlejohn. *Survey of software Failsafe Techniques for Safety-Critical Automotive Applications*, in SAE World Congress, Detroit Michigan, Delphi Corp., 2005.

[Lopes & Hursch 1995] C. Lopes, W. Hursch, *Separation of Concerns*, Technical Report, College of Computer Science, Northeastern University, Boston, USA, Feb 1995.

[Lu *& al.* 2009 ETFA] C. Lu, J.C. Fabre, M.O. Killijian, *Robustness of modular multilayered software in the automotive domain: a wrapping-based approach*, in Proc. of the 14th Int. IEEE Conf. on Emergent Technology and Factory Automation (ETFA'09), Palma-de-Mallorca, Spain, Sept. 2009.

[Lu *& al.* 2009 RTNS] C. Lu, J.C. Fabre, M.O. Killijian, *An approach for improving Fault-Tolerance in Automotive Modular Embedded Software*, in Proc. of the 17th Int. IEEE Conf. on Real-Time and Network Systems (RTNS'09), Paris, France, Oct. 2009.

[Madeira *& al.* 2000] H. Madeira, D. Costa, M. Vieira. *On the emulation of software faults by software fault injection*, in Proc. of the Int. IEEE Conf. on Dependable Systems and Networks (DSN'2000), New York, USA, pp. 417--426, Jun. 2000.

[Maes 1987] P. Maes, *Concepts and Experiments in Computational Reflection. Conference on Object-Oriented Programming Systems, Languages, and Applications*, in Proc. of the Int. IEEE Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'1987), Orlando, Florida. pp. 147--155, Dec. 1987.

[Mariani 2003] L.Mariani. *A Fault Taxonomy for Component-based Software*. Electronic Notes in Theoretical Computer Science 82 No.6, 2003.

[Marques 2006] R.S. Marques. *Méthodologie de développement des services de communication temps-réel d'un intergiciel embarqué dans l'automobile*, PhD Thesis, LORIA, Nancy, France, 2006.

[Marsden 2004] E. Marsden. *Caractérisation de la Sûreté de Fonctionnement de Systèmes à base d'Intergiciel*. Phd Thesis, LAAS-CNRS, Toulouse, France, 2004.

[Marsden *& al.* 2002] E. Marsden, J-C. Fabre, J. Arlat. *Dependability of CORBA Systems : Service Characterization by Fault Injection.* in the 21th IEEE International Symposium on Reliable Distributed Systems (SRDS'2002), Suita, Japan, pp. 276--285, Oct. 2002.

[MATHIX] MATHIX. *Sûreté de fonctionnement des logiciels*. Training Course, Nov. 2006.

[MISRA 1998] MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Technical Report, Apr 1998.

[NASA 1998] B.L.D. Vito. *A formal model of partitioning for Integrated Modular Avionics*. Technical Report NASA-98-cr208703, 1998.

[NASA 2004] NASA, *NASA Software Safety Guidebook*. NASA technical standard, NASA-GB-8719.13, 2004.

[OSEK] OSEK group, *OSEK/VDX Operating system Version 2.2.3*, Technical report, 2005.

[Pan *& al*. 2001] J. Pan, P. Koopman, D. Siewiorek, Y. Huang, R. Gruber, M.L. Jiang. *Robustness testing and hardening of CORBA ORB implementations*, in Proc. of the Int. IEEE Conf. on Dependable Systems and Networks (DSN'2001), Göteborg, Sweden, pp. 141--150, Jul. 2001.

[Parlavantzas & Coulson 2000] N. Parlavantzas, G. Coulson. *Towards a reflective component-based middleware architecture*. in Workshop on Reflection and Metalevel Architectures (ECOOP'2000), Sophia Antipolis, France, Jun. 2000.

[Pernet & Sorel 2005] N. Pernet, Y. Sorel. *Transformations de spécifications incluant du contrôle en spécification flot de données pour implantation distribuée*. In Actes de la Conférence Modélisation des Systèmes Réactifs (MSR'05), Grenoble, France, Oct. 2005.

[Racu & al. 2006] R.Racu, R.Ernst, K.Richter, *The need of a timing model for the AUTOSAR software standard*, in Workshop on Models and Analysis Methods for Automotive Systems (RTSS Conference), Rio de Janeiro, Brazil, Dec. 2006.

[Randell 1975] B. Randell, *System structure for software fault-tolerance*. IEEE Trans. on Software Engineering, vol.SE-1, no.2, pp.220-232, 1975.

[Rebaudengo *& al*. 1999] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, M. Violante, *Soft-error detection through software fault-tolerance techniques*, in IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 210--218, 1999.

[RENAULT SdF] RENAULT. *Sensibilisation à la Sécurité Fonctionnelle des Systèmes Electriques & Electroniques*. Training Course, Aug 2006.

[RENAULT spec.] RENAULT. *System and IS software safety requirements for the supplier justification needs*. Technical Report, 2007.

[RESEDA] SILICOMP/LIPN/RENAULT/TRIALOG. *Rapport Final Projet RESEDA*. Technical Report, Dec 2004.

[Rimén *& al.* 1994] M. Rimén, J. Ohlsson, J. Torin. *On microprocessor error behaviour modeling*, in the 24th IEEE International Symposium on Fault Tolerant Computing (FTCS'1994), Austin, Texas, pp. 76--85, Jun. 1994.

[Rodriguez 2002] M. Rodriguez. *Technologie d'empaquetage pour la sûreté de fonctionnement des systèmes temps-réel*, Phd Thesis, LAAS-CNRS, Toulouse, France, 2002.

[Rodriguez *& al.* 2002 EDCC] M. Rodriguez, J.C. Fabre, J. Arlat, *Wrapping real-time systems from temporal logic specifications*, in Proc. of the 4th Int. IEEE Conf. on European Dependable Computing Conference (EDCC'2002), Toulouse, France, pp. 253--270, Oct. 2002.

[Rodriguez *& al.* 2002 DSN] M. Rodriguez, A. Albinet, J. Arlat. *MAFALDA-RT: A tool for dependability assessment of Real-Time Systems*, in Proc. of the Int. IEEE Conf. on Dependable Systems and Networks (DSN 2002), Washington, USA, pp. 267--272, Jun. 2002.

[Salkham 2005] A.M. Salkham, *Fault Detection, Isolation and Recovery (FDIR) in On-Board Software*, Master's Thesis, Chalmers University of Technology, Göteborg, Sweden, 2005.

[Simulink] http://www.mathworks.com/products/simulink/

[Smith 1983] B.C. Smith. *Reflection and Semantics in Lisp*. in the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages Conference, Salt Lake City, USA, pp 23--35, 1983.

[Stateflow] http://www.mathworks.com/products/stateflow/

[Strom & Yemini 1985] R.E. Strom, S. Yemini, *Optimistic recovery in distributed systems, ACM Trans. on Computer Systems*, vol. 3, no. 3, pp. 204-226, 1985.

[Sullivan & Chillarege 1991] M. Sullivan, R. Chillarege. *Software Defects and their Impact on System Availibility, a study of Field Failures in Operating Systems.* , in the 21th IEEE International Symposium on Fault Tolerant Computing (FTCS'1991), Montreal, Canada, pp 2--9, Jun. 1991.

[SUN 1996] Sun Microsystems. *Java Core Reflection API and Specification.* Technical Report, 1996.

[Taiani *& al.* 2003] F. Taïani, M-O. Killijian, J-C. Fabre, *Towards Implementing Multi-Layer Reflection for Fault-Tolerance*, in Proc. of the Int. IEEE International Conference on Dependable Systems and Networks (DSN'2003), San Francisco, USA, pp. 435--444, Jun. 2003.

[Taïani 2004] F. Taïani, *La Réflexivité dans les architectures multi-niveaux : application aux systèmes tolérant les fautes*, Phd Thesis, LAAS-CNRS, Toulouse, France, 2004.

[Taiani *& al.* 2005] F. Taïani, M-O. Killijian, J-C. Fabre, *A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures*, in Proc. of the Int. IEEE International Conference on Dependable Systems and Networks (DSN'2005), Yokohama , Japan, pp. 270--279, Jun. 2005.

[Taïani 2006] F. Taïani, M-O. Killijian, J-C. Fabre, *Intergiciels pour la tolerance aux fautes, état de l'art et défis*. RSTI-TSI, 25 (5): 599-630, 2006.

[Tindell *& al.* 2003] K. Tindell, H. Kopetz, F. Wolf, R. Ernst, *Safe Automotive Software Development*, in Proc. of the Int. IEEE International Conference on Design, Automation and Test in Europe (DATE'2003), Washington, USA, pp 10616, Mar. 2003.

[Traverse *& al.* 2004] P. Traverse, I. Lacaze, J. Souyris, *Airbus fly-by-wire: A total approach to dependability*. in IFIP World Computer Congress, WCC 2004, Toulouse, France, Aug. 2004.

[Totel 1998] E. Totel, *Politique d'intégrité multiniveau pour la protection en ligne des tâches critiques*, Phd Thesis, LAAS-CNRS, Toulouse, France, 1998.

[Voas 1997] J. Voas, *A Defensive Approach to Certifying COTS Software*. Reliable Software Technologies Corporation, Technical Report: RSTR-002-97-002.01, 1997.

[Wei 2006] Y.Wei. *A study of Software Input Failure Propagation Mechanisms.* PhD Thesis, University of Maryland, USA, 2006.

[Workflow Patterns] Workflow Patterns Initiative, http://www.workflowpatterns.com

[Yau & Chen 1980] S. Yau, F. Chen, *An Approach to Concurrent Control Flow Checking*, in IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, pp. 126--137, 1980.

[Yokote 1992] Y. Yokote, *The Apertos Reflective Operating Systems : The concept and its implementation*, in Proc. of the Int. IEEE Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'1992), Vancouver, Canada, pp. 414--434, Oct. 1992.

[Zenha Rela *& al.* 1996] M. Zenha Rela, H. Madeira, J. G. Silva, *Experimental Evaluation of the Fail-Silent Behavior in Programs with Consistency Checks*, in the 26th IEEE International Symposium on Fault Tolerant Computing (FTCS'1996), Sendai, Japan, pp. 394--403, Jun. 1996.