Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations

AC	Caractéristiques Applicatives, page 14
ADAS	Advanced Driver Assistance Systems, page 25
AFT	Adaptive Fault Tolerance, page 4
AOP	Programmation Orientée Aspect, page 17
API	Application Programming Interface, page 60
ARXML	AUTOSAR XML, page 152
ASIL	Automotive Safety Integrity Level, page 28
AUTOSAR	AUTomotive Open System ARchitecture, page 5
AV	Acceptance Voting, page 89
BPA	Before - Proceed - After, page 16
BSW	Basic SoftWare, page 32
CBSE	Component-Based Software Engineering, page 18
CD	Crash Detector, page 54
COTS	Components Off-The-Shelf, page 31
D4A	Design for Adaptation, page 12
DDS	Data Distribution Service, page 151
E/E	Électriques/Électroniques, page 27
E2E	End-to-End protection, page 50
ECU	Electronic Control Unit, page 32
EM	Execution Management, page 153
FIFO	First In - First Out, page 50
FMECA	Failure Mode, Effects and Criticality Analysis, page 28
FT	Modèle de faute, page 14
FTA	Fault Tree Analysis, page 28
FTM	Fault Tolerant Mechanism, page 4
GRAFCET	Graphe Fonctionnel de Commande des Étapes et Transitions, page 80
HmD	Homogenous Duplex, page 88

HtD	Heterogenous Duplex, page 88
IPC	Inter-Process Communication, page 151
LFR	Leader Follower Replication, page 15
MISRA	Motor Industry Software Reliability Association, page 27
MooN	M-out-of-N, page 89
MSM	Machine State Management, page 153
NREC	National Robotics Engineering Center, page 25
NSCP	N-Self Checking Programming, page 89
NVP	N-Version Programming, page 89
OTA	Over-The-Air, page 13
P-BPA	Protocol - Before - Proceed - After, page 45
PBR	Primary Backup Replication, page 15
PHM	Platform Health Management, page 157
POC	Programmation Orientée Composant, page 17
RB	Recovery Block, page 89
ROS	Robot Operating System, page 5
RS	Besoins en ressource, page 14
RTE	Run-Time Environment, page 32
SdF	Sûreté de Fonctionnement, page 6
SnC	Sanity Check, page 89
SOC	Service Oriented Communication, page 151
SoC	Separation of Concern, page 12
SOME/IP	Scalable service-Oriented MiddlewarE over IP, page 151
SWC	SoftWare Component, page 32
TaF	Tolérance aux fautes, page 47
TMR	Triple Modular Redundancy, page 92
TR	Time Redundancy, page 15
UCM	Update Configuration Management, page 153
UML	Unified Modeling Language, page 18
VFB	Virtual Functional Bus, page 33
XML	Extensible Markup Language, page 60

Introduction & Objectifs

Au cours des dernières années, l'utilisation des véhicules automobiles a évolué. Précédemment symbole de liberté permettant de se rendre à n'importe quel endroit, la voiture est devenue, au fil de sa popularisation, un calvaire au quotidien, synonyme de temps perdu dans le trafic. Un français passant en moyenne 4 ans de sa vie dans sa voiture, ces dernières proposent à leurs utilisateurs de plus en plus de logiciels embarqués, que cela soit des aides à la conduite (aide au parking, régulateur de vitesse) ou de l'*infotainment* (Application mobile, réseau 4G) avec pour objectif le véhicule autonome. Ainsi, à l'instar du téléphone mobile se transformant en *smartphone*, la voiture n'est plus un simple outil de déplacement mais une véritable *smartcar*.

Ce changement d'utilisation de la voiture a entraîné une explosion des calculateurs embarqués appelé ECU (*Electronic Control Unit*) poussant les constructeurs automobile à former un consortium appelé AUTOSAR [AUTOSAR]. Ce consortium à pour but de standardiser le support d'exécution logiciel des ECU. L'avantage de cette standardisation est de permettre une meilleure réutilisation du code, une standardisation des interfaces pour une meilleure communication entre les différents calculateurs, et une abstraction des couches basses (protocol de communication, drivers, système d'exploitation) avec la couche applicative. De plus, l'Organisation Internationale de Normalisation (ISO) a définit la norme ISO 26262 [Sinha 2011] permettant de garantir la sécurité fonctionnelle des systèmes électriques/électroniques dans les véhicules automobiles.

Avec l'arrivée de Tesla sur le marché automobile, la voiture est de plus en plus personnalisable au niveau du logiciel qui peut être modifié à n'importe quel moment de la vie du véhicule. De ce fait, la capacité de mettre à jour à distance les systèmes embarqués automobiles est devenue un enjeu technologique majeur pour les constructeurs. Cette volonté de mettre à jour les systèmes à distance est avant tout économique. D'une part, cela permet de diminuer les coûts de maintenance du logiciel car le rappel au garage n'est, dans ce cas, plus nécessaire. D'autre part, cela donne une image de marque pour attirer de nouveaux clients en proposant d'ajouter des options après l'achat du véhicule.

Cependant, qui dit plus de systèmes embarqués, dit accroissement du nombre de bugs pouvant entraîner la défaillance des dits systèmes. Des techniques de sûreté de fonctionnement sont donc mises en place pour assurer des propriétés telles que la fiabilité, la disponibilité ou encore la maintenabilité des systèmes. Parmi ces techniques se trouvent les mécanismes de tolérance aux fautes permettant de détecter et de corriger une erreur pouvant survenir. Avec la mise à jour à distance du logiciel, il faut également assurer la pertinence des mécanismes mis en place et les modifier si nécessaire. De plus, le logiciel peut être soumis à des perturbations (évènement non prévu) pouvant également entraîner la défaillance des systèmes. Afin d'augmenter la disponibilité (leur durée de fonctionnement), il faut être capable d'adapter en

ligne les mécanismes avec le moins impact possible sur le système.

Le standard actuel *Classic* AUTOSAR, n'est pas suffisant pour suivre l'évolution rapide des systèmes. La possibilité de mettre à jour les systèmes de manière différentielle est limitée. Le consortium développe un nouveau standard qui vise à fournir un environnement avec une puissance de calcul plus élevée, le déploiement dynamique de nouvelles fonctionnalités, l'interaction avec des applications non-AUTOSAR, et même des mises à jour du système à distance [Fürst 2016]. Cela a encouragé le développement indépendant d'*Adaptive* AUTOSAR dans le but d'avoir toujours les deux plates-formes coexistant et fonctionnant ensemble sur le même réseau sans compromettre la stabilité de l'architecture *Classic* AUTOSAR existante.

Les recherches effectuées dans cette thèse s'articulent autour de la mise à jour des mécanismes de tolérance aux fautes sur les supports d'exécution à composant. Nous souhaitons définir une approche satisfaisante pour adapter et mettre à jour ces mécanismes pour répondre aux problématiques suivantes :

Comment concevoir une architecture logicielle pour adapter et mettre à jour des mécanismes de tolérance aux fautes ?
Comment projeter cette architecture au contexte automobile ?

Ce mémoire présente notre travail suivant 5 chapitres. Le Chapitre 1 définit les différentes notions importantes à la compréhension de cette thèse. Il présente également le contexte et l'avancée des recherches dans ce domaine. Il définit également quatre sous-problématiques auxquelles nous répondrons dans les chapitres suivants.

Le Chapitre 2 présente un *framework* pour la conception de mécanismes de tolérances aux fautes adaptables. Il met en avant trois conditions, la *Separation of Concern*, le *Design for Adaptation* et le *Fine-grain Update*, pour pouvoir adapter les mécanismes sans impacter l'application à laquelle ils sont attachés. De plus, nous effectuons une implémentation et une évaluation de notre approche sur ROS, un intergiciel permettant la conception d'architecture à composant.

Le Chapitre 3 mettra en avant une première amélioration concernant cette approche en affinant la granularité des composants constituant nos mécanismes et en proposant une vision non plus basée sur des composants mais orientée action de sûreté de fonctionnement.

Le Chapitre 4 proposera, quant à lui, un autre *framework* destiné à améliorer les consommation de ressources en concevant des mécanismes comme des graphes d'objets dynamiques. Une implémentation et une évaluation seront également effectuées sur ROS pour valider cette seconde approche.

Enfin, le Chapitre 5 aura pour objectif de projeter nos deux approches sur les deux plate-formes automobiles *Classic* AUTOSAR et *Adaptive* AUTOSAR. La seconde plate-forme étant pour le moment incomplète, nous proposerons une analyse critique des concepts proposés par le consortium.

En conclusion, nous tirerons les leçons de cette étude en soulignant les contraintes auxquelles nous avons du obéir pour concevoir nos approches et les projeter sur les plate-formes automobiles. De plus, nous énumérerons les possibles perspectives vouées à améliorer nos travaux de recherche.

Chapitre 1

Contexte & Démarche scientifique

Sommaire

Introduction

A travers ce premier chapitre, nous allons définir les concepts clés pour la compréhension de nos travaux de recherche, les objectifs scientifiques et la démarche pour les obtenir.

Cette thèse a pour objectif de définir une architecture logicielle pour permettre la mise à jour simple dans les systèmes embarqués automobiles de mécanismes de tolérance aux fautes ou ***F**ault **T**olerant **M**echanisms* . L'acronyme FTM sera utilisé par la suite pour y faire référence.

Cette définition de l'objectif autour duquel nous avons fondé nos recherches a pour but d'assurer le bon fonctionnement des systèmes informatiques embarqués dans les véhicules automobiles. En effet, l'évolution croissante du désir d'une voiture connectée ayant de plus en plus d'aide à la conduite implique pour le constructeur une nécessité de mettre à jour les systèmes informatiques embarqués à bord.

En fonction de la criticité des systèmes, c'est à dire l'impact sur la sûreté qu'aurait un mauvais fonctionnement de ces systèmes, des FTM peuvent y être rattachés. Ces FTM ont pour but, par exemple, de vérifier que les valeurs transmises sont correctes, que le système fonctionne correctement ou qu'il est capable de continuer à fonctionner en cas de problèmes (p.ex. en cas de capteur défaillant, utilisation d'un capteur secondaire). Ces mécanismes sont définis en fonction du système et de la façon dont il pourrait défaillir. La mise à jour d'un système pourrait entraîner un changement de comportement et donc nécessiterait une évolution des FTM qui y sont rattachés.

Comme le montre le contexte de nos travaux de recherche, de nombreux domaines intrinsèquement liés se croisent, s'enchevêtrent et donc nécessitent de plus amples explications. Il nous faudra donc expliquer les bases de la sûreté de fonctionnement où sont définies les techniques assurant le bon fonctionnement des systèmes informatiques, les architectures logicielles permettant une évolution rapide et facile des systèmes ainsi que l'environnement des systèmes embarqués automobiles et leurs caractéristiques.

Dans un premier temps nous définirons les principes de sûreté de fonctionnement, à savoir assurer des propriétés permettant à l'utilisateur d'avoir confiance dans un système, comprenant les notions de tolérance aux fautes, et le principe de résilience, à savoir la capacité de s'adapter à son environnement.

Dans un deuxième temps, nous présenterons le concept de la Tolérance aux Fautes Adaptative (en anglais *Adaptive Fault Tolerance* ou AFT [1]), c.-à-d., la capacité d'adapter les mécanismes de tolérance aux fautes. Nous en déduirons nos besoins en terme de support d'exécution logiciel pour la conception et l'implémentation de nos FTM évolutifs.

Dans un troisième temps, nous introduirons les notions d'architecture à composants permettant de créer des applications logicielles par assemblage de blocs interagissant les uns avec les autres au travers de messages. Cette seconde partie

1. cet acronyme sera utilisé dans le reste du document pour parler de tolérance aux fautes adaptative.

nous permettra de définir les caractéristiques attendues d'un support d'exécution pour implémenter l'AFT.

Dans un quatrième temps, nous présenterons les principales caractéristiques de ROS (*Robot Operating System*), un support d'exécution lié à la robotique, qui permet de créer des architectures logicielles à composants et qui est un candidat intéressant pour nos premières implémentations de l'AFT.

Dans un cinquième temps, nous examinerons le contexte industriel automobile et les standards définissant le niveau de criticité des applications embarqués (ISO 26262) ainsi que le support d'exécution utilisé (*Classic* AUTOSAR). Nous évoquerons également l'évolution de ce support pour permettre la mise à jour des systèmes embarqués (*Adaptive* AUTOSAR).

Dans un sixième et dernier temps, nous expliciterons notre démarche scientifique en présentant les problèmes généraux auxquels nous avons dû faire face et la manière dont nous nous y sommes confrontés. Nous évoquerons également les points qui ne seront pas abordés dans cette thèse.

1 Sûreté de fonctionnement et Résilience

Dans cette première section, nous allons définir les principes qui sous-tendent l'*Adaptive Fault Tolerance*, à savoir la Sûreté de fonctionnement des systèmes informatiques et plus précisément la tolérance aux fautes et la résilience. De plus nous synthétiserons les besoins tant du point de vue de l'architecture que de l'implémentation de l'AFT, pour préciser les technologies nécessaires à sa mise en place.

1.1 Sûreté de Fonctionnement Informatique

La sûreté de fonctionnement d'un système informatique (SdF) est *la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre, le service étant le comportement du système perçu par un utilisateur, cet utilisateur étant un système (informatique, humain, environnemental) qui interagit avec le premier.* [Laprie 1995] C'est donc la capacité d'un système informatique de répondre de manière correcte, conformément aux spécifications fonctionnelles, à une requête d'un autre système.

La sûreté de fonctionnement est définie en fonction de trois notions principales : *les attributs* qui définissent les propriétés assurées, *les entraves* qui caractérisent les circonstances indésirables mais prévues et *les moyens* qui précisent les techniques permettant au système de fournir son service.

Selon les services souhaités par l'utilisateur, ce dernier peut vouloir accentuer certaines propriétés pour assurer le bon fonctionnement du système. Ainsi la sûreté de fonctionnement englobe les attributs suivants :

- **La disponibilité :** la capacité d'être prêt à délivrer le service correct ;
- **La fiabilité :** l'assurance de continuité d'un service correct ;
- **La sécurité-innocuité :** l'assurance de non-propagation de conséquences catastrophiques à l'utilisateur ou l'environnement ;
- **L'intégrité :** l'assurance de non-altération du système ;
- **La maintenabilité :** l'aptitude à la réparation et à l'évolution du système.

Ces attributs permettent d'une part d'exprimer les propriétés devant être respectées par le système, et d'autre part d'évaluer la qualité du service délivré vis-à-vis de ces propriétés. Les aspects de sécurité, au sens de la confidentialité et des attaques, à savoir, la disponibilité des actions autorisées, l'intégrité du système face à des actions irrégulières ainsi que la confidentialité, c.-à-d., la non-occurrence de divulgation non-autorisée d'informations, ne seront pas abordés dans cette thèse.

Les entraves à la sûreté de fonctionnement sont les défaillances, les erreurs et les fautes. **Une défaillance** est une transition d'un service correct vers un service incorrect. Un service est considéré incorrect s'il n'est pas conforme à la spécification ou si la spécification ne décrit pas avec précision la fonction du système. Étant donné qu'un service consiste en une séquence d'états externes du système (observés par l'utilisateur), la survenue d'une défaillance signifie qu'au moins un des états externes s'écarte de l'état correct du service. La déviation est liée à **une erreur**, qui

représente la partie de l'état interne du système pouvant entraîner une défaillance, dans le cas où elle atteint l'interface du service du système. La cause déterminée ou présumée d'une erreur est appelée **une faute**. La figure 1.1 représente ce lien de cause à effet.

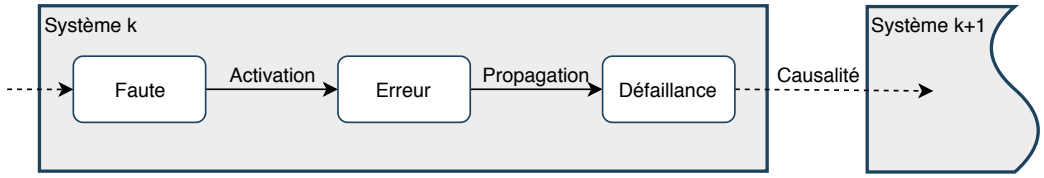


FIGURE 1.1 – Chaînes de causalité des entraves à la sûreté de fonctionnement

Pour minimiser l'impact de ces entraves sur les attributs retenus d'un système, la sûreté de fonctionnement dispose de méthodes et techniques qui permettent de conforter les utilisateurs quant au bon accomplissement des fonctions du système. Le développement d'un système sûr de fonctionnement passe donc par l'utilisation combinée de ces méthodes, appelés moyens, pouvant être classées en quatre types :

- **Prévision des fautes :** estimation de la présence, de la création et des conséquences des fautes (p. ex. Analyse FMEA) ;
- **Prévention des fautes :** entrave à l'occurrence ou l'introduction de fautes (p. ex. outil de génie logiciel) ;
- **Élimination des fautes :** réduction du nombre et de la sévérité des fautes (p. ex. test, injection de fautes) ;
- **Tolérance aux fautes :** capacité de fournir un service, optimal ou dégradé, en présence de fautes (p. ex. techniques de redondance).

La prévention et la tolérance aux fautes visent à fournir la capacité de délivrer un service correct, tandis que l'élimination et la prévision des fautes visent à susciter la confiance en cette capacité en justifiant que les spécifications fonctionnelles, de sûreté de fonctionnement et de sécurité sont adéquates et que le système est conforme. Toutes ces techniques ne servent pas à délivrer un service applicatif, mais à garantir des propriétés de sûreté de fonctionnement issues des spécifications non-fonctionnelles. Ces aspects non-fonctionnels comprennent non seulement la Sûreté de Fonctionnement mais également des critères tels l'ergonomie ou l'audit.

Après avoir énoncé les principes clés de la sûreté de fonctionnement, nous allons nous intéresser plus précisément à la tolérance aux fautes.

1.2 Tolérance aux fautes

Les fautes auxquelles un système doit faire face sont nombreuses et peuvent ne pas avoir d'impact sur celui-ci tant qu'un ou plusieurs évènements ne se sont pas produits. On les appelle alors des fautes dormantes. Une fois activées, ces fautes peuvent avoir un impact catastrophique sur le système. D'origines diverses et variées, certaines fautes sont dues à l'environnement, au matériel, ou encore à l'être humain.

Elle peuvent être involontaires suite au vieillissement du matériel, aux conditions climatiques, à une erreur de conception, ou malignes comme des portes dérobée, virus, erreurs volontaires de programmation et sont alors dites malveillantes [Laprie 1995]. Elles sont classées en fonction huit critères présentés en figure 1.2.

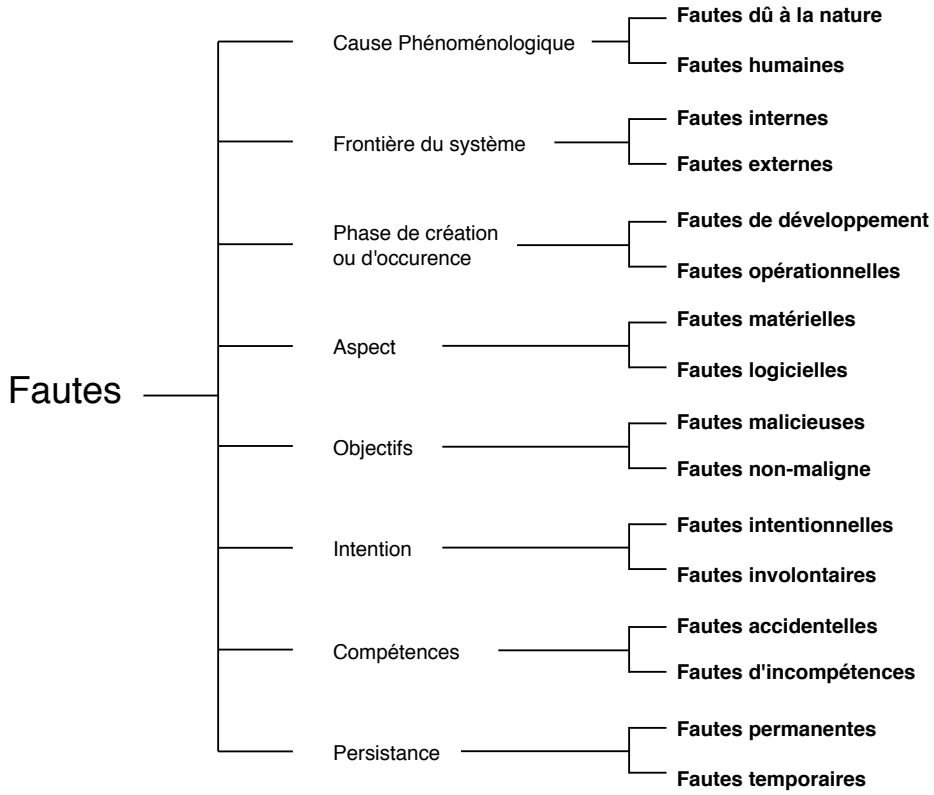


FIGURE 1.2 – Classes élémentaires de fautes

Chaque faute peut provoquer une ou des erreurs différentes pouvant entraîner la défaillance du système. Malgré l'application des techniques de prévention et d'élimination des fautes, certaines subsistent et sont à même d'être activées.

Un système tolérant aux fautes doit pouvoir assurer à l'utilisateur un service correct en dépit des fautes pouvant altérer ses composants, durant sa conception ou son interaction avec d'autres systèmes [Avizienis 2004]. La Tolérance aux fautes est mise en œuvre grâce aux moyens de **détection d'erreurs**, c.-à-d., l'identification des déviations du service correct, et de **recouvrement**, c.-à-d., les techniques permettant en cas d'erreur détectée de passer d'un état de système fautif à un état assurant un service nominal ou dégradé.

La détection d'erreur peut être soit **concurrente** et se déroulant pendant l'exécution du système soit **anticipée** en vérifiant les paramètres du système lors de la suspension de son exécution. Une fois cette erreur détectée, les techniques de recouvrement peuvent être employées, d'une part pour assurer le service désiré et éviter la propagation de l'erreur (traitement des erreurs) et d'autre part pour isoler

le composant fautif, diagnostiquer l'erreur, trouver et déterminer la faute originelle pour assurer une opération de maintenance (traitement des fautes).

Les techniques de détections et de recouvrement sont nombreuses et sont regroupés dans des mécanismes de tolérance aux fautes associés à un ou plusieurs types de fautes. Il n'y a à l'heure actuelle aucun mécanisme générique pouvant pallier n'importe quel type de fautes ou d'erreurs. Certaines stratégies de tolérance aux fautes ont été classifiées en fonction du type de fautes auxquelles elles répondaient [Stoicescu 2012a]. Que cela soit de la redondance matérielle, logicielle, temporelle, de la diversité dans l'implémentation ou l'architecture, les techniques sont nombreuses et souvent propres à chaque domaine et au budget alloué à la tolérance aux fautes.

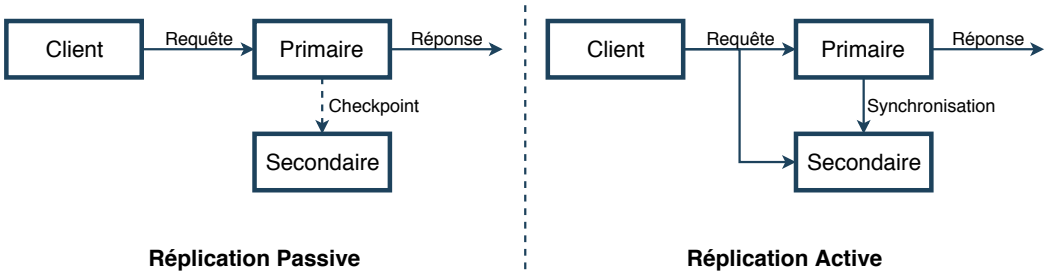


FIGURE 1.3 – Mécanismes de tolérance aux fautes matérielles permanentes

La figure 1.3 montre deux techniques de redondance permettant la tolérance de fautes matérielles permanente. La première technique repose sur la sauvegarde et la transmission de l'état d'un serveur primaire vers un secondaire alors que l'autre repose sur l'exécution parallèle des deux serveurs. Dans les deux cas, si le serveur primaire tombe (arrêt sur défaillance), le serveur secondaire prend le relais pour fournir le service.

Les techniques dépendent du modèle de faute mais également du contexte applicatif. Notre mécanisme de réplication passive requiert un accès à l'état pour que le primaire puisse le capturer et l'envoyer par un message de checkpoint au secondaire. Par cet échange de message, le secondaire peut mettre à jour son état et être dans un état identique au primaire. Notre mécanisme de réplication active exige de l'application du déterminisme dans son exécution. Comme les deux répliques s'exécutent en parallèle, il faut qu'elles fournissent le même résultat. Ajoutons que la livraison des messages de requête doit se faire de manière identique sur les deux répliques (diffusion atomique).

Ces mécanismes sont conçus après analyse des fautes statiques (qui n'évoluent pas) déduites durant la phase de prévision des fautes et sont développés en parallèle du système. Ils accompagnent le système durant toute sa vie opérationnelle.

Nous avons vu rapidement dans ce paragraphe le rôle de la tolérance aux fautes dans la sûreté de fonctionnement. La tolérance aux fautes est le dernier rempart une fois le système déployé pour assurer l'exécution correct du service. Néanmoins, comme dit dans le paragraphe 1.1, les entraves sont prévues. Nous allons maintenant

nous intéresser au cadre des circonstances indésirables et non-prévues.

1.3 Résilience

La résilience désigne la capacité d'un corps, d'un organisme, d'une espèce, d'un système à surmonter une altération de son environnement. C'est un concept qui apparaît dans de nombreux domaines, tels que la psychologie, la science des matériaux, le commerce, l'écologie et, enfin, l'informatique. Dans chacun de ces domaines, les définitions restent proches les unes des autres et les notions de perturbations et de la capacité à s'en remettre sont au coeur de ce concept.

Pour les systèmes informatiques, elle est définie dans [Laprie 2008] comme la persistance de la sûreté de fonctionnement en dépit de perturbations. On peut donc étendre la définition de résilience comme étant *la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service en dépit de perturbations parfois imprévues.*

Tout comme les fautes définies dans la section 1.2, ces perturbations sont variées et peuvent avoir des conséquences catastrophiques sur le système. Ces perturbations sont classifiées selon trois critères :

- **La nature :** perturbation environnementale, fonctionnelle ou technologique ;
- **L'échéance :**
  - Court terme : de l'ordre de la secondes à heures, p. ex., système dynamique ;
  - Moyen terme : de l'ordre de l'heure au mois, p. ex, nouvelle version logicielle ;
  - Long terme : du mois à l'année, p. ex., évolution du hardware ;
- **La prévisibilité :**
  - Prévue : due à une nouvelle version
  - Prévisible : due au vieillissement matériel
  - Imprévisible : due au changement drastique d'environnement

La capacité du système à absorber ces perturbations et donc l'impact qu'elles vont avoir sur celui-ci est directement lié à ces trois critères. Une perturbation fonctionnelle, prévisible à moyen terme, peut être un changement de contexte applicatif dû à une mise à jour de la version du logiciel entraînant la perte de déterminisme d'une application critique. Une perturbation environnementale, non prévisible à court terme, peut être l'apparition de fautes transitoires en valeur émanant d'une tempête de grêle subite aveuglant les capteurs de freinage d'urgence d'une voiture. Dans le premier cas, si seule l'application est mise à jour et non les FTM, le système n'est plus résilient. Dans le second cas, si aucun FTM n'a été prévu pour tolérer ce genre de fautes, le système n'est pas résilient également.

Un système résilient doit donc être capable d'évoluer et de s'adapter aux changements d'hypothèses, de contexte, d'architecture. Des travaux d'implémentation de la résilience au niveau du hardware et du software sont toujours au coeur des tra-

vaux de plusieurs équipes de recherche et reste un véritable challenge [Marin 2001] [Li 2008].

Des notions très fortes sont rattachées à la résilience telles que l'**évolutivité**, l'**évaluabilité**, la **diversité** et l'**accessibilité** [Laprie 2008]. Nous tournant vers la tolérance aux fautes, nous sommes surtout concernés par les notions d'accessibilité pour la mise à jour, de diversité pour éviter les fautes dues au matériel ou logiciel identique, d'évolutivité pour absorber les perturbations mais surtout d'adaptabilité, une sous-catégorie de l'évolutivité, pour absorber ces perturbations pendant l'exécution du système.

Nous avons vu dans cette section que la résilience complétait la sûreté de fonctionnement avec pour objectif de faire face aux fautes introduites par des changements. Nous allons maintenant voir comment la résilience s'intègre à la tolérance aux fautes.

2 Tolérance aux Fautes Adaptative

Les premiers travaux de la Tolérance aux Fautes Adaptative ou ***A**daptive **F**ault **T**olérance* remontent à [Kim 1990], qui la définissent comme : "*une approche permettant de répondre à l'exigence de tolérance aux fautes de façon dynamique et très variée en utilisant de manière efficace et adaptative un nombre limité et en constante évolution de ressources disponibles*".

Considérée dans un premier temps comme une branche de la tolérance aux fautes encore sous sa forme naissante, l'AFT a suscité dans un second temps un intérêt croissant, devenant une méthode à part entière. L'AFT est à la tolérance aux fautes ce que la résilience est à la sûreté de fonctionnement. C'est un moyen d'améliorer la résilience d'un système, prenant en compte l'état courant de ce système au niveau des ressources, de leur état et de l'environnement source de perturbations.

2.1 Mécanismes de tolérance aux fautes adaptatifs

L'AFT se veut une solution pour répondre à une question simple : Les mécanismes de tolérance aux fautes ne sont valides que sous certaines hypothèses. Que faire si une ou plusieurs hypothèses changent au cours de la vie opérationnelle du système ?

Le principe de l'AFT va donc nécessiter trois exigences primordiales. Il faut que les FTM mis en place **(1)** puissent évoluer indépendamment de l'application fonctionnelle, **(2)** aient une architecture permettant d'être modifiée avec un impact minimum sur la disponibilité de l'application et **(3)** puissent être mis à jour à distance [Stoicescu 2011].

Pour répondre à ces trois exigences, nous utilisons les concepts de *Separation of Concern* (*SoC*) et de *Design For Adaptation* (*D4A*). Nous fonderons nos travaux sur ces même concepts.

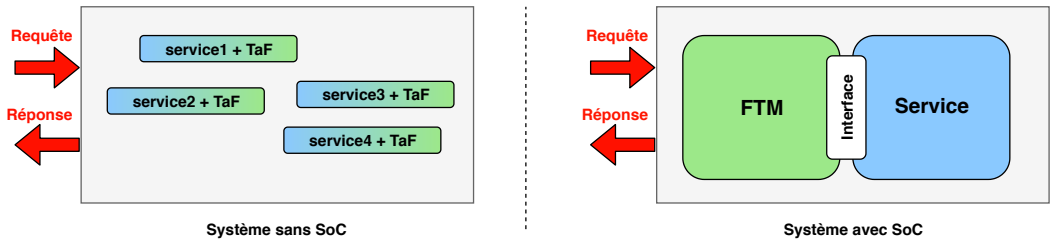


FIGURE 1.4 – Mise en application de la *Separation of Concern*

La *Separation of Concern*, présentée dans [Dijkstra 1982], signifie concentrer son attention sur un certain aspect en oubliant temporairement ceux qui ne sont pas pertinents au sujet. Pour l'AFT, cela signifie séparer les parties fonctionnelles d'un système, qui délivrent le service, avec les parties non-fonctionnelles, qui assurent la délivrance d'un service correct par des techniques de tolérance aux fautes comme l'illustre la figure 1.4. Cela permet donc de développer les FTM en parallèle du système, de les rattacher au système et de les modifier indépendemment de ce

dernier. Seules les interfaces sont communes pour l'interaction entre le FTM et le service.

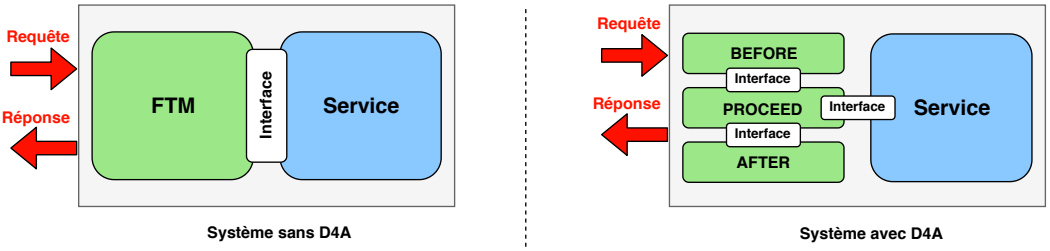


FIGURE 1.5 – Mise en application du *Design For Adaptation*

Le *Design For Adaptation* va être la capacité de créer les mécanismes pour permettre une modification rapide ayant un impact minime sur la délivrance du système. Dans le cas d'un intercepteur, un mécanisme qui intercepte les requêtes vers une application, on distingue les fonctions exécutées avant le service (p. ex. sauvegarde de l'état initial...), celle pendant (p. ex. vérification d'ordre d'exécution de tâche) et enfin celle après le service (p. ex. sauvegarde d'un point de reprise). Ce modèle de conception (*Design Pattern*) appelé *Before - Proceed - After* illustré sur la figure 1.5, est mis en avant dans [Stoicescu 2013].

Ce découpage en ordre d'exécution permet de compartimenter les différences entre les mécanismes et donc de définir les changements nécessaires au passage d'un mécanisme à un autre. Finalement, cela permet de ne changer que ce qui diffère entre deux FTM. Si nous prenons, par exemple, le FTM de réplication active de la section 1.3, nous avons deux intercepteurs entre le *Client* et le primaire et le secondaire. Nous pouvons concevoir le mécanisme avec le *Before* qui contiendrait la synchronisation avant l'exécution avec le partage de la requête du primaire vers le secondaire, le *Proceed* qui effectuerait l'opération requise sur le service et l'*After* qui contiendrait la synchronisation post-exécution.

La mise à jour à distance (*Over-The-Air updates*), quant à elle, permet de suivre l'évolution logicielle qui interviendra au cours de la vie opérationnelle du système sans avoir un besoin d'accès à la plate-forme matérielle. Que cela soit pour des raisons financières (rappel de véhicule coûteux) ou techniques (intervention manuelle sur satellite impossible), la mise à jour à distance est nécessaire pour réagir aux perturbations.

L'AFT présente donc de nombreux besoins (SoC, D4A, OTA *updates*) et de nombreuses contraintes (Manipulation en ligne des FTM, Manipulation des communications, Intégrité de l'envoi des mises à jour à distance) pour son implémentation. D'une part, la *Separation of Concern* va imposer des règles de conception, des choix de langages et de techniques de programmation pour permettre le développement des parties fonctionnelles et non-fonctionnelles de manière détachée. D'autre part, le *Design For Adaptation* va demander une architecture logicielle et des solutions techniques permettant la manipulation dynamique des fonctions de tolérance aux fautes.

L'idée repose sur une analyse fine des mécanismes de tolérance aux fautes qui conduit à concevoir chacun d'eux sous la forme de briques élémentaires (des composants logiciels dans notre cas). Plusieurs mécanismes partagent plusieurs de ces composants, d'autres sont plus spécifiques au mécanisme considéré. L'AFT exigeant une évolution de l'architecture interne des mécanismes, elle astreint à une adaptation des communications internes (inter-composant) et externes (entre le client, le mécanisme et le serveur).

On en déduit qu'un des moyens pour réaliser l'AFT est de créer des mécanismes de tolérance aux fautes comme un jeu de construction où chaque pièce mise dans un certains ordre permet de faire une structure particulière pouvant être réutilisée ailleurs. L'association "SoC + D4A + OTA updates" semble être une façon efficace de réaliser l'AFT et donc d'améliorer la résilience du système.

2.2 Mécanismes de tolérance aux fautes à composants

Comme nous l'avons défini en section 2.1, l'AFT a pour but d'assurer la sûreté de fonctionnement en cas de perturbation survenant durant la vie opérationnelle. Les FTM sont choisis pour répondre à certaines contraintes et sous certaines hypothèses. Une fois qu'une perturbation s'active, elle peut rendre une hypothèse obsolète et invalider l'utilisation du mécanisme.

Les travaux [Excoffon 2016], [Stoicescu 2012b] et [Stoicescu 2013] mettent en avant la définition d'hypothèses selon trois classes qui sont le modèle de faute (FT), définissant le type et l'impact d'une faute en cas de défaillance (P.ex. Fautes matérielles permanentes ⇒ Réplication matérielle), les caractéristiques de l'application (AC) qui vont imposer des choix de stratégies de tolérance aux fautes (Pas d'accès à l'état ⇒ Pas de réplication passive) et enfin, les ressources nécessaires (RS) au bon fonctionnement de l'application (Bande passante limité ⇒ Pas d'envoi de message de checkpoint) et des FTM qui lui sont rattachée en terme de mémoire, de bande passante, de nombre de CPU alloué etc... L'AFT servant à adapter les FTM en cas de perturbation, il est nécessaire d'assurer la transition d'un mécanisme à un autre.

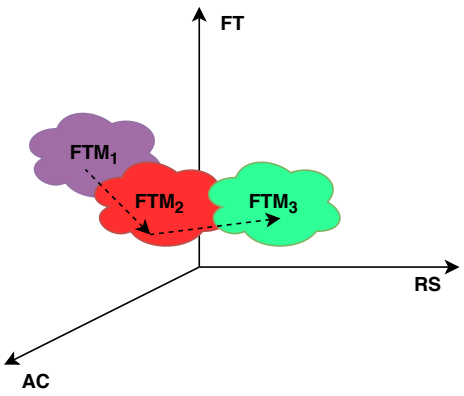


FIGURE 1.6 – Mis en application du *Design For Adaptation*

La figure 1.6 illustre les zones de validité des FTM en fonction des trois hypothèses FT, AC et RS ainsi que les transitions à assurer dans le cas d'un changement d'une ou de plusieurs de ces hypothèses. Dans le référentiel FT, AC, RS, un nuage représente un sous-espace dans lequel un FTM est valide selon ces trois axes.

Certains mécanismes peuvent avoir des hypothèses communes et ainsi tolérer le même type de fautes tout en ayant des besoins ou en ressource ou des caractéristiques applicatives différentes. Les réplications active et passive permettent de tolérer les fautes matérielles par crash, cependant la réplication active à besoin de déterminisme pour assurer la cohérence des résultats de chaque réplique alors que la redondance passive à besoin d'accéder à l'état pour le capturer et le sauvegarder. Les travaux cités précédemment s'appuient sur 3 mécanismes pour mettre en avant cette décomposition : La redondance passive (***P**rimary **B**ackup **R**eplication* ou *PBR*), la redondance active (***L**eader **F**ollower **R**eplication* ou *LFR*) et enfin la redondance temporelle (***T**ime **R**edundancy* ou *TR*). En analysant selon les 3 classes d'hypothèses, on obtient le tableau 1.1.

Hypothèse / FTM		PBR	LFR	TR
Modèle de Faute	crash	✓	✓	
(FT)	transitoire			✓
Caractéristique	accès à l'état	✓		✓
(AC)	déterminisme		✓	
Ressources	Bande Passante	High	Low	Nulle
(RS)	CPU	#2	#2	#1

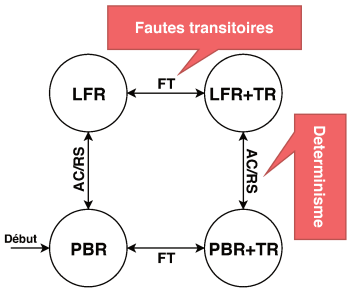


Table 1.1 – Hypothèses et transitions entre trois mécanismes

On remarque deux choses à partir de ce tableau. La première est que les hypothèses peuvent être communes entre les différents FTM. La deuxième est qu'on peut composer les mécanismes pour répondre à des modèles de fautes complexes. Un avantage du *Design for Adaptation* est donc la composition des mécanismes qui est supporté par la programmation orientée composant.

Pour permettre la transition d'un mécanisme à un autre, on décompose les mécanismes selon le principe du *D4A*, c.-à-d., définir pour chaque mécanisme les actions de Sûreté de Fonctionnement correspondant au *Before*, celles correspondant au *Proceed* et celle relatives à l'*After*. On pourra ainsi identifier les actions qui sont communes et qui donc ne nécessitent pas d'être échangées lors d'une transition. La Programmation Orientée Composant n'est pas la seule solution existante. Il est en soit possible d'avoir une approche à grain fin et d'utiliser la Programmation Orientée Objet pour décomposer les actions de SdF en objet et de manipuler ces objets pour faire évoluer les mécanismes. Mais l'avantage de la première est duale : elle permet **(1)** une implémentation en accord avec le *D4A* où chaque partie du mécanisme va être modélisée en un composant et **(2)** une intégration d'autres composants dans le système et donc l'ajout de nouvelles fonctions de tolérance aux fautes par la composition.

FTM		Before	Proceed	After
PBR	(Primaire)		Compute	Sauvegarde d'état + checkpoint
	(Secondaire)			Restoration d'état
LFR	(Primaire)	Envoi Requête	Compute	Envoi Synchronisation
	(Secondaire)	Gestion Requête	Compute	Gestion Synchronisation
TR		Sauvegarde/ Restoration d'état	Compute	Comparaison

TABLE 1.2 – Décomposition des mécanismes en *Before* - *Proceed* - *After*

Le tableau 1.2 nous montre la décomposition de ces trois mécanismes en suivant le *D4A*. On remarque que pour passer d'un *PBR* à un *LFR* par exemple, il faut ajouter le composant *Before* du *LFR* et également remplacer l'*After* du *PBR* par celui du *LFR*. L'avantage ici est le fait de laisser en place le *Proceed* et donc de ne pas interagir avec les communications entre le serveur et le FTM. De plus, les blocs sont triviaux et demande moins d'effort à remplacer que le mécanisme complet. Les travaux [Stoicescu 2013] montrent une implémentation de ces mécanismes sous FraSCAti, une plate-forme permettant la programmation et la gestion d'un système à composants [Seinturier 2009].

Nous avons à présent défini les caractéristiques que nous recherchons dans un support d'exécution permettant de mettre en œuvre l'AFT. La phase de conception implique pour le support d'exécution une capacité à développer des applications sous forme d'assemblage de composants logiciels. Il faut donc que celui-ci permette la création d'unité d'exécution communiquant entre elles et sur lesquelles nous pourrons implémenter nos briques élémentaires définies dans la section 2.1. De plus, l'évolutivité des FTM impose au support la capacité de gérer l'état de ces unités, à savoir, la capacité de les arrêter, de les relancer, de les supprimer ou encore de les créer de manière dynamique, sans oublier la gestion des communications qui accompagnent ces évolutions. Nous utiliserons dans nos premières expérimentations le schéma de conception *Before* - *Proceed* - *After* (BPA) pour explorer des moyens de mise en œuvre de l'AFT.

3 Programmation Orientée Composant

Nous avons synthétisé dans la section précédente les notions de sûreté de fonctionnement et de résilience reposant sur la tolérance aux fautes. Nous avons aussi exprimé les besoins de conception et d'implémentation qui sont nécessaires à la Tolérance aux Fautes Adaptative.

Nous allons maintenant aborder les premières réponses à ces besoins à travers la Programmation Orientée Composant (POC). Nous nous intéresserons tout d'abord à la Programmation Orientée Aspect qui a proposé en premier la possibilité d'une architecture logicielle décomposée sous forme de module que l'on peut développer indépendemment du code principal (fonctionnel) et que l'on peut insérer (activer) facilement à n'importe quel endroit de celui-ci.

3.1 De l'Aspect au Composant

La programmation orientée aspect (AOP) a été proposée comme technique permettant d'améliorer la *Separation of Concern*. L'AOP s'appuie sur des technologies antérieures, notamment la programmation orientée objet, qui a déjà considérablement amélioré la modularité des logiciels. Un vaste corpus de recherche existe sur le sujet de la programmation orientée aspect dans de nombreux travaux tels que [Kiczales 1997] et [Irwin 1997] ou encore [Elrad 2001].

L'idée de l'AOP repose sur des concepts différents de la programmation orientée objet. Un aspect correspond à l'implémentation d'une spécification non-fonctionnelle qui est synchronisée avec le code applicatif. Cette synchronisation s'effectue grâce à divers moyens tels des *pointcuts* ou des intercepteurs sophistiqués s'appuyant sur des éléments structurels du langages (p. ex. les appels de fonction).

Avec l'AOP, un système peut être décomposé en module. Chacun de ces modules peut gérer une spécification non-fonctionnelle. Ces modules comportent des greffons et des points d'activation. Les greffons sont le code qui sera exécuté lorsque l'application passera par le point d'activation. L'AOP est la capacité d'activer des modules (des morceaux de code) à des moments spécifiques comme avant ou après l'exécution d'une application. Concernant notre schéma de conception BPA, nous pourrions avoir deux modules *Before* et *After* qui sont activés avant et après l'exécution de l'application.

Le problème avec l'AOP est que les aspects sont enchevêtrés au code et ne permettent pas de réaliser correctement le niveau de *SoC* que nous attendons pour l'AFT. L'AOP reste une technique de programmation statique avec une définition a priori des modules. Il existe des possibilités de rendre la modification des modules dynamiques mais qui reste fastidieuse et liée à des outils spécifiques.

Nous allons donc nous tourner vers la Programmation Orientée Composant. Cette technique de programmation consiste à utiliser une approche modulaire de l'architecture d'un projet informatique. Les concepteurs, au lieu de créer un exécutable unique, se servent de briques réutilisables pour constituer leur application.

3.2 Architectures logicielles à composant

L'ingénierie logicielle à base de composants ou *Component-based Software Engineering* (CBSE) [Parnas 1972] est une branche de l'ingénierie logicielle qui, comme l'AOP, met l'accent sur la *Separation of Concern* concernant les fonctionnalités étendues disponibles dans un système logiciel donné. Il s'agit d'une approche basée sur la réutilisation pour la conception, la mise en œuvre et la composition de composants indépendants faiblement couplés dans des systèmes. Cette pratique vise à apporter autant d'avantages, à court et à long terme, au logiciel lui-même et aux organisations qui l'utilise [Szyperski 1999].

L'idée du composant logiciel remonte aussi loin qu'en 1968 dans la présentation de Douglais Mcllroy «*Mass Produced Software Components*» [McIlroy, 68]. Depuis lors, un grand nombre de modèles de composants, tant commerciaux qu'académiques, a été introduit en utilisant différentes technologies, visant différents objectifs et basés sur des principes différents. *Enterprise JavaBeans* [Rubinger 2010] développé par Sun Microsystems, *Component Object Model* (COM) [Sherlock 2000] de Microsoft, *CORBA Component Model* (CCM) [Geib 1997], OpenCOM [Blair 2004], Fractal [Blair 2009] ou encore ROS [Quigley 2009] font partis des exemples les plus connus.

Étant donné les objectifs et technologies si différents les uns des autres, il n'y a à ce jour aucun standard ou langage de description d'architecture commun contrairement à la Programmation Orientée Objet (UML pour la description des interactions et des caractéristiques des objets). Pourtant, la définition de la notion de composant logiciel est communément admise : "*Un composant logiciel est une unité de composition comportant uniquement des interfaces spécifiées par contrat et des contextes explicites. Un composant logiciel peut être déployé indépendamment et est sujet à une composition par un tiers*" [Szyperski 2002].

Cependant, il existe d'autres définitions soulignant l'importance des modèles de composants auxquels les composants doivent se conformer, comme indiqué dans [Crnkovic 2011]. Néanmoins, l'objectif de nos travaux n'est pas de fournir une étude des modèles de composants existants que l'on peut trouver dans les travaux de [Szyperski 2002], [Heineman 2001], [Crnkovic 2011] ainsi que dans les différentes implémentations de systèmes à composant énumérées plus tôt.

Les principes fondamentaux des techniques CBSE semblent être excellents pour l'AFT car ils mettent en place la notion d'unité d'exécution pouvant être couplée à d'autres unités composant un système. Nous allons maintenant analyser des travaux effectués pour l'implémentation de FTM en utilisant la Programmation Orientée Composant

Nous nous intéresserons dans la partie suivante au support d'exécution ROS qui semble prometteur et qui est utilisé autant dans la recherche pour la robotique que dans l'industrie pour le prototypage de systèmes allant de robots gérant le déplacement de marchandise dans des hangars à la voiture autonome.

4 Robot Operating System

Dans les parties précédentes, nous avons analysé les caractéristiques de la programmation orientée composants, quelques exemples d'architectures ainsi que les besoins spécifiques pour implémenter des mécanismes de tolérance aux fautes adaptatifs. Nous allons maintenant nous intéresser à un intergiciel nommé ROS qui permet la conception et l'implémentation d'architecture logicielle à composants pour la robotique et nous nous intéresserons particulièrement à ses capacités d'adaptation.

Nous expliquerons, dans la suite de cette section, les caractéristiques techniques du fonctionnement de ROS qui sont pré-requises pour la compréhension de l'implémentation de nos diverses approches.

4.1 Pourquoi ROS ?

En tant que support d'exécution pour la résilience des systèmes informatiques, les systèmes d'exploitation classiques basés sur Linux ou POSIX ne sont pas entièrement satisfaisant pour l'adaptation pendant l'exécution d'applications. La manipulation des unités exécutables (Processus, tâches...) et des communications (*socket*) est fastidieuse à réaliser. Nous nous sommes donc tourné vers un intergiciel, ROS, qui permet la réalisation d'application sous forme d'assemblage de composants et la modification de cet assemblage pendant l'exécution.

ROS est un intergiciel (ou *middleware* en anglais) s'exécutant sur un système d'exploitation Unix (généralement Linux) [Welsh 1999]. Il est décrit comme : "*un système de méta-exploitation open source pour votre robot. Il fournit les services que vous pouvez attendre d'un système d'exploitation, notamment l'abstraction matérielle, le contrôle de périphérique de bas niveau, la mise en œuvre des fonctionnalités couramment utilisées, la transmission de messages entre processus et la gestion des packages. Il fournit également des outils et des bibliothèques permettant d'obtenir, de générer, d'écrire et d'exécuter du code sur plusieurs ordinateurs*" [Quigley 2009].

L'avantage de ROS est la facilité de conception des applications en permettant au développeur de se concentrer sur une conception haut niveau sans se préoccuper du matériel sous-jacent. Par exemple, il suffit, pour créer une communication entre plusieurs composants, de définir qui écrit à qui et comment s'appelle cette communication. Il n'y a pas besoin de savoir quelle est l'adresse, sur quelle machine sont ces composants, ROS s'occupe de projeter cette description haut niveau sur un canal de communication. ROS est une plate-forme de développement intéressante pour tester les concepts de l'Adaptive Fault Tolerance.

4.2 ROS : Une architecture logicielle en LEGO

L'objectif principal de ROS est de permettre la conception d'applications modulaires pour la robotique : une application ROS est un ensemble de programmes, appelés nœuds, qui n'interagissent que par transfert de messages au travers de canaux de communication qui peuvent être des *topics* ou des *services* [Quigley 2009].

Ces nœuds sont des unités d'exécution élémentaires isolées les unes des autres temporellement (synchronisation par message) et spatialement (espace d'adressage réservé). Le développement d'une application implique l'assemblage de nœuds, ce qui s'apparente à des approches basées sur des composants. Un tel assemblage est appelé le graphe d'exécution de l'application.

En somme, une application ROS est un assemblage de petites briques élémentaires remplissant une ou plusieurs fonctions et communiquant entre elles par l'intermédiaire de messages.

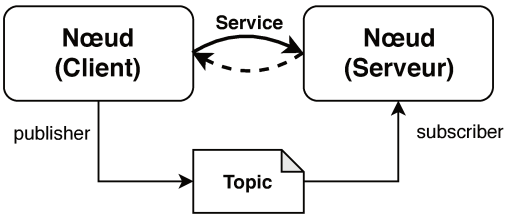


FIGURE 1.7 – Types de communication ROS

Sur la Figure 1.7, les deux modèles de communication disponibles dans ROS sont représentés : un modèle *publish/subscribe* et un modèle *client/serveur*. Le modèle *publish/subscribe* définit des communications unidirectionnelles, plusieurs-à-plusieurs, asynchrones via le concept de *Topic*. Le modèle *client/serveur* repose sur des communications synchrones bi-directionnelles via le concept de *service*. Ces modèles de communications de haut niveau introduisent la modularité et la flexibilité dans les systèmes logiciels. Chaque *topic* et *service* sont définis par leur nom, leur type de données, et la taille de la file d'attente de message pour le premier. Les adresses des nœuds ne sont donc pas explicitées lorsque deux nœuds communiquent.

Pour créer ces communications, le nœud doit définir un port de communication qui peut être soit un *publisher* ou un *subscriber* pour les *topics* et un *client* ou *server* pour les *services*. Une fois ces ports définis, le nœud utilise une méthode pour envoyer un message ou pour réceptionner un message et définir ce qu'il doit faire sur réception.

Pour fournir ce niveau d'abstraction, chaque application ROS comprend un processus spécial, pouvant être commun à plusieurs applications, appelée *ROS Master*. C'est une sorte d'annuaire contenant l'adresse des nœuds, leur types de communications, leur paramètres. C'est le seul processus qui dispose d'une vue complète du graphe d'exécution. Lorsqu'un nœud veut créer une communication de n'importe quel type, il s'enregistre au près du *ROS Master* en déclarant le nom du *topic* ou du *service*. Les nœuds qui veulent envoyer ou recevoir des messages à travers ce nouveau canal de communication s'enregistrent également en employant le même nom et le *ROS Master* les met en relation.

La figure 1.8 montre la création d'une communication de type *publish / subscribe* entre deux nœuds ROS. Premièrement, l'Émetteur crée le port de communication *publisher* puis, avec la méthode `advertise`, s'enregistre auprès du *ROS Master* comme *publisher* sur le *topic* "Data". Deuxièmement, le Récepteur crée le port de

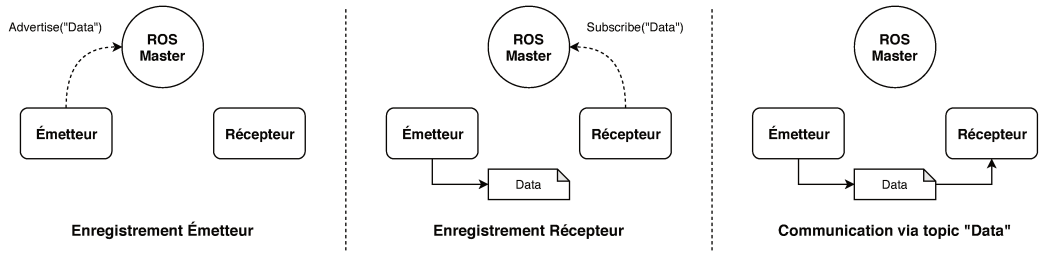


FIGURE 1.8 – Établissement d'un topic entre deux nœuds

communication *subscriber* puis, avec la méthode `subscribe`, s'enregistre au près du *ROS Master* comme *subscriber* sur le topic "Data" et définit l'action à exécuter sur réception de message. Une fois la connexion établie entre les deux nœuds, ils communiquent directement dans le cas d'un *topic*. Dans le cas d'un service, le nœud contenant le client demande avant d'envoyer chaque requête l'emplacement du serveur au *ROS Master*. La communication par *service* est donc dépendante du *ROS Master* non seulement à l'initialisation de la communication mais également à l'envoi de chaque requête.

La programmation de ROS s'effectue en C++ ou en Python. Les nœuds sont projetés à l'exécution sur des processus Unix et l'ordonnancement est géré par le système d'exploitation. D'un point de vue utilisateur, ROS est une collection de bibliothèques permettant de programmer des composants (les nœuds) pouvant contenir plusieurs objets ou fonctions. En bref, ROS offre la possibilité de créer une architecture distribuée complexe tout en restant simple à réaliser.

4.3 Conception d'une application sous ROS

ROS semble compatible avec l'AFT car il offre une notion de composant que l'on peut manipuler. Mais tout cela est très théorique et peu représentatif de ce qu'est réellement une application sous ROS. Nous allons, dans cette partie, illustrer le cas de deux nœuds communiquant à travers un *topic*. Nous verrons donc comment le nœud "Émetteur" enverra un nombre entier à le nœud "Récepteur".

4.3.1 Communication sous ROS

Avant de rentrer dans le cœur de l'exemple, nous allons d'abord voir comment créer les deux types de messages (synchrones et asynchrones).

Les messages sont définis dans des fichiers textes. Les messages asynchrones sont définis dans des fichiers avec l'extension `.msg` alors que les messages synchrones sont définis dans des fichiers avec l'extension `.srv`.

```
1 Int64 data
```

Listing 1.1 – Contenu d'un message asynchrone message.msg

```
1 Int64 requete
2 ---
3 Int64 reponse
```

Listing 1.2 – Contenu d'un message synchrone message.srv

Les deux listing 1.1 et 1.2 montrent la différence fondamentale entre ces deux types de messages, à savoir, la définition d'une donnée de retour pour les messages synchrones. Le champ de retour (ligne 3) du message synchrone peut être vide si le concepteur souhaite juste informer le *client* de la réussite de l'action du serveur. De plus, il existe dans ROS un certain nombre de messages standards que l'on peut utiliser.

Nous allons maintenant décrire l'exemple au travers duquel nous expliquerons la conception d'une application ROS.

4.3.2 Description de l'exemple

Pour rappel, un *topic* est un canal de communication uni-directionnel au travers duquel communique des nœuds par messages asynchrones. Il est caractérisé par les éléments suivants :

- **Le Nom** : Le canal de communication est nominatif ;
- **Un émetteur** : Objet permettant l'enregistrement au près du *Master* du *Publisher* et l'envoi de messages ;
- **Un récepteur** : Objet permettant l'enregistrement au près du *Master* du *Subscriber* et la réception de messages ;
- **Le type de donnée** : Un type de donnée unique par *topic* spécifiant le contenu des messages.

Dans notre exemple, nos deux nœuds communiquerons via un *topic* que nous appellerons "Topic_heure", au travers duquel elles s'enverront un message "msg" contenant un entier. Le nœud "Émetteur" générera un entier aléatoire entre 1 et 10 et le nœud "Récepteur" affichera le message "Il est $data heure du matin et tout va bien".

La librairie de messages standards de ROS *std_msgs* peut être utilisée pour spécifier les données de messages prédéfinis. La liste de ces messages est très complète mais la plupart sont triviaux (une donnée ou des tableaux d'un seul type). Des messages plus complexes peuvent être employés et sont personnalisables. Pour une question de praticité, nous utiliserons un message standard contenant un entier de type *Int64*.

4.3.3 Création du Nœud "Émetteur"

Voici, dans un premier temps, le code complet du nœud que nous expliquerons au fur et à mesure :

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"
#include <cstdlib>
#include <time.h>

int main(int argc, char **argv){
  ros::init(argc, argv, "Emetteur");
  ros::NodeHandle n;
  ros::Publisher pub = n.advertise<std_msgs::Int64>("Topic_heure", 10);

  std_msgs::Int64 msg;
  srand(time(0));

  while (ros::ok()){
    msg.data = (rand() % 10) + 1;
    pub.publish(msg);
    sleep(5);
  }
  return 0;
}
```

Listing 1.3 – Code du nœud Émetteur

On peut voir dans ce code deux parties distinctes entre l'initialisation du code et son exécution :

`ros::init(argc, argv, "Emetteur")`

Cette ligne permet de définir le nom du nœud mais également les arguments *argv* pour l'adaptation. En effet, on remarque que ROS ne peut modifier les paramètres d'un nœud qu'à son initialisation.

`ros::NodeHandle n`
`ros::Publisher pub = n.advertise<std_msgs::Int64>("Topic_heure", 10)`
L'objet *NodeHandle* est le point d'accès principal de toute les communications ROS. Nous créons ici notre *publisher* "pub" et nous spécifions le fait qu'il écrira des messages de type *Int64* sur "Topic_heure" avec une file de messages pouvant stocker 10 messages.

Nous créons ensuite notre message "msg" (`std_msgs::Int64 msg`) que nous envoyons avec `pub.publish(msg)` toutes les 5 secondes tant que le nœud n'a pas été supprimée (`while(ros::ok())`).

Nous voyons qu'il est facile d'écrire un nœud en utilisant simplement les librairies fournies par ROS. La façon dont ROS gère les communications est très opaque et demande une analyse en profondeur des caractéristique de l'intergiciel.

4.3.4 Création du Nœud "Recepteur"

Comme précédemment, voici le code complet du nœud que nous expliquerons au fur et à mesure :

```
#include "ros/ros.h"
#include "std_msgs/Int64.h"
#include <stdlib.h>

void afficherHeure(const std_msgs::Int64::ConstPtr& msg){
  std::cout<<"Il est "<<msg->data<<"h du matin et tout va bien"<<std::endl;
}

int main(int argc, char **argv){
  ros::init(argc, argv, "Recepteur");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("Topic_heure", 10, afficherHeure);

  ros::spin();
  return 0;
}
```

Listing 1.4 – Code du nœud Recepteur

Les principales différences entre Récepteur et Émetteur est que Récepteur doit définir une action à effectuer (ici afficher un texte contenant la valeur du message reçu) sur réception d'un message. Cela va se traduire par :

`ros::Subscriber sub = n.subscribe("Topic_heure", 10, afficherHeure);`
`ros::spin();`
Nous avons créé ici un *subscriber* "sub" qui va écouter sur le canal "Topic_heure" et qui activera la méthode *afficherHeure*. L'utilisation de `spin()` est le coeur du fonctionnement des activations de fonctions sur réception de message. En arrivant à ce point, le nœud va entrer dans une boucle qui va regarder à intervalle régulier l'apparition d'un nouveau message (mode écoute). A chaque nouveau message, il appellera la méthode définie dans la méthode `subscribe` puis repassera en mode écoute. Il ne sortira de cette boucle que lors de la suppression du nœud.

Comme pour Émetteur, on remarque que l'on peut faire un nœud certes très simple mais fonctionnel en peu de lignes. L'autre point important est l'utilisation de *spin()* qui implique une activité constante des nœuds. Des ressources seront toujours utilisées pour écouter le canal à une fréquence fixée par le développeur.

Nous allons maintenant parcourir quelques exemples d'utilisation de ROS dans des systèmes concrets.

4.4 Exemples d'utilisation de ROS

Donc, ROS permet de faire de la programmation orientée composant, mais est-ce vraiment utilisé et pas uniquement dans des cas expérimentaux obscurs ? La réponse à cette question est évidemment oui. Nous allons énumérer plusieurs domaines dans lequel il est utilisé avec son utilisation ainsi que les systèmes sur lesquels des applications ROS sont déployées.

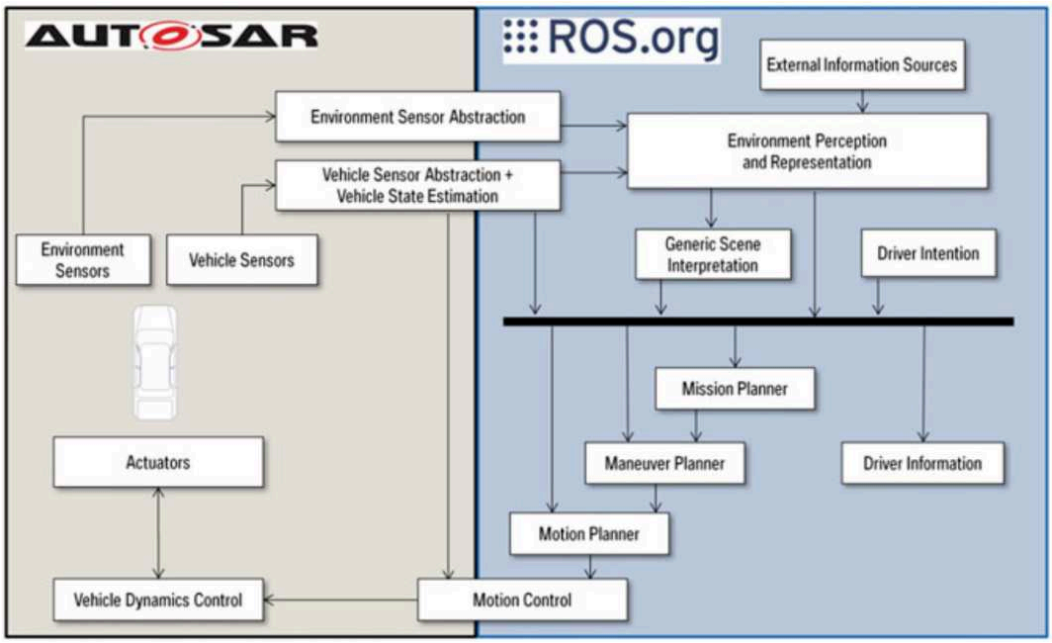


FIGURE 1.9 – Architecture de la conduite autonome avec ROS par BMW

Avant d'être utilisé par certains constructeurs automobiles pour le prototypage d'application de conduite autonome, ROS fut utilisé par le ***N**ational **R**obotics **E**ngineering **C**enter* (NREC) qui fait partie du département de robotique de l'université de Carnegie Mellon, à Pittsburg. Le NREC commença à utiliser ROS comme solution pour ses tanks (plate-forme de char d'assaut) autonomes appelés *Crusher* [NREC].

BMW fût le premier constructeur automobile à s'intéresser à ROS pour faire d'une part des prototypes d'ADAS (***A**dvanced **D**river-**A**ssitance **S**ystems*) [Guettier 2016] et d'autre part les premières architectures logicielles pour les voitures autonomes [Aeberhard 2015] comme le montre à figure 1.9.

De plus, de nombreux systèmes divers utilisent des applications déployée sous ROS :

Personal Robot 2 : PR2, développé par Willow Garage, est l'un des premiers robots conçus pour exécuter ROS [Quigley 2015]. Il est également l'un des robots les plus avancés et les plus performants avec ROS à l'heure actuelle. PR2 est conçu pour la recherche et le développement d'applications de robots de service. [Marder-Eppstein 2010] décrit comment le PR2 a utilisé la pile de navigation pour parcourir de manière autonome 42 km (26,2 miles).

TurtleBot : TurtleBot est une alternative moins chère, compatible ROS, à PR2. Il se compose d'une base mobile avec différent plateaux s'installant dessus et permettant de personnaliser l'architecture matérielle en ajoutant des capteurs. C'est un robot de support pédagogique très intéressant pour appréhender les concepts de ROS et déployer ses premières applications.

Robonaut 2 : Robonaut 2 est un robot humanoïde habile, qui réside actuelle-

ment dans la Station spatiale internationale à environ 400 km au-dessus de la surface de la Terre [ROS b]. Robonaut est conçu pour la recherche sur la manière dont les robots peuvent aider l'équipage à entretenir et à exploiter la station spatiale. Son but est donc d'aider le travail des astronautes dans des situations délicates, notamment lors de sorties extra-véhiculaires. Il permet aussi d'effectuer des tâches répétitives et ainsi d'économiser le précieux temps des astronautes.

Industrial Hardware : Le programme ROS-industrial [ROS c] fournit des interfaces matérielles à divers équipements industriels. L'IRB-2400 d'ABB en est un exemple. ROS-industrial fournit un ensemble de logiciels de planification de mouvement (MoveIt !) et de téléchargement de trajectoires [ROS a].

Jusqu'à présent, nous avons étudié les caractéristiques et les besoins de l'AFT, nous en avons déduit que la Programmation Orientée Composant est une technique intéressante pour son implémentation et nous avons pu confirmer que ROS est un candidat sérieux pour servir de support d'exécution. Il permet le développement d'applications sous forme d'association de composants logiciels, a des capacités d'adaptation et est en plus utilisé dans l'automobile pour le prototypage d'*ADAS* et de systèmes permettant la conduite autonome.

Nous allons maintenant nous intéresser plus précisément au domaine automobile, en quoi l'AFT est utile pour ce domaine et comment la mettre en œuvre.

5 Contexte Automobile

Dans ce chapitre, nous avons présenté précédemment les notions essentielles de sûreté de fonctionnement, énoncé les besoins de faire de la tolérance aux fautes adaptative, synthétisé des techniques basées sur l'ingénierie logicielle à composant pour la conception et l'adaptation de FTM, ainsi que présenté succinctement un candidat sérieux pour l'implémentation de l'AFT.

Nous allons maintenant nous intéresser au contexte automobile. Les constructeurs automobiles se dirigent de plus en plus vers une voiture connectée avec du logiciel embarqué qui peut être mis à jour à distance. Ces mises à jour peuvent impliquer un changement de contexte applicatif conduisant à l'apparition de nouvelles fautes. L'AFT semble être une méthode intéressante pour d'une part développer les FTM parallèlement aux applications et d'autre part rendre le système résilient face à ces modifications. En sus, nous nous préoccuperons des caractéristiques du contexte automobile à respecter que ça soit du point de vue de la conception et de l'implémentation, que des règles de sûreté de fonctionnement.

5.1 Un standard pour développer les applications critiques

Avant 2011, les principaux guides de sûreté de fonctionnement pour le logiciel automobile étaient l'IEC 61508 et le standard MISRA, qui furent utilisés lors du développement de systèmes embarqués. Encore aujourd'hui, ces guides établissent des règles de conception et d'implémentation qui garantissent le développement de systèmes électriques/électroniques (E/E) sûrs de fonctionnement.

L'IEC 61508 est un standard européen générique pour un nombre très large de secteurs d'activités [Bell 2006]. 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 de ce guide sont difficilement applicables à des projets très contraints (délais courts, coûts de mise en œuvre). Elles permettent néanmoins de fournir des recommandations quant aux taux de fiabilité des composants E/E (P.ex. la fiabilité des plate-formes matérielles embarquées appelée ECU - *Electronic Control Unit*) à utiliser en fonction du niveau de criticité établi.

Le standard MISRA (Motor Industry Software Reliability Association), quant à lui, est issu d'une collaboration de constructeurs automobiles, et donne des règles pour guider le développement logiciel dans les véhicules, de manière à éviter certaines erreurs de conception [Hatton 2007]. Ce standard apporte des directives de développement pour les logiciels embarqués sur les véhicules visant à atteindre des objectifs tels qu'apporter robustesse et fiabilité au sein du logiciel automobile ou s'assurer que la sécurité des personnes doit être prépondérante quand elle est en conflit avec la sécurité du véhicule.

Depuis 2011, une première version de la norme ISO 26262 (« Véhicules routiers - Sécurité fonctionnelle») a été publiée à la suite des travaux du groupe de travail « ISO TC22/SC3/WG16 ». Ici, le terme "sécurité fonctionnelle" qui est utilisé dans

la norme est ce qui est décrit comme sûreté de fonctionnement dans ce manuscrit. C'est 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. La norme ISO26262 marque une dépendance forte entre la sûreté de fonctionnement et le processus de développement d'un produit. C'est finalement une norme de développement ISO pour les systèmes de sécurité dans les véhicules routiers à moteur. Elle définit un cadre et un modèle d'application, ainsi que les activités, les méthodes à utiliser lors des phases de développement, et les données de sortie attendues. La mise en œuvre de cette norme permet de garantir la sûreté de fonctionnement des systèmes électrique/électronique dans les véhicules automobiles. En effet à chaque étape du processus de développement, 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 recommandées.

L'ISO 26262 se compose de dix parties normatives qui couvrent les thèmes suivants :

1. Le Vocabulaire
2. Gestion de la sécurité fonctionnelle
3. Phase de projet (phase de conception)
4. Développement du produit au niveau du système
5. Développement du produit au niveau du matériel
6. Développement du produit au niveau du logiciel
7. Production et utilisation (maintenance et recyclage)
8. Processus d'appui
9. Analyses liées aux niveaux d'intégrité de sûreté automobile (ASIL)
10. Lignes directrices relatives à l'ISO 26262

Les parties de l'ISO 26262 qui vont nous concerner sont principalement les parties 3, 4 et 6. En effet, ces parties contiennent les exigences concernant la réalisation d'une Analyse des risques déterminant les niveaux de criticité ainsi que les moyens à mettre en place au niveau conceptuel, matériel et logiciel (choix des ECU, FTM...). Les analyses peuvent être diverses allant de l'analyse des modes de défaillance, de leurs effets et de leur criticité (FMECA - *Failure Mode, Effects, and Criticality Analysis*) aux arbres de fautes (FTA - *Fault Tree Analysis*). Cette thèse se concentre sur l'adaptation des mécanismes rattachés aux applications. Ces analyses permettant de choisir quel FTM appliquer ne seront donc pas détaillés dans cette thèse.

L'ISO 26262 adapte les SIL de l'IEC 61508 en Automotive SIL ou ASIL. Chaque situation dangereuse sera cotée selon un niveau d'exigence en matière de sûreté de ASIL A (peu critique) à ASIL D (très critique), sinon comme non influent sur la sûreté (quality management - QM).

Le tableau 1.3 montre la définition d'un niveau ASIL en fonction des critères de sévérité des conséquences, de probabilité d'apparition d'une défaillance et de contrôlabilité en cas de défaillance. Prenons deux exemples extrêmes, la climatisation et le système de freinage et imaginons que la probabilité d'une défaillance soit importante dans les deux cas (vieillissement de l'hélice de soufflerie et usure des câbles de commande de freinage) (E4). Dans le premier cas, les conséquences d'une défaillance sont inexistantes (S1) et c'est tout à fait contrôlable par le conducteur (C1). Le système sera donc qualifié de QM. Dans le second cas, les conséquences en cas de défaillance du système de freinage peut avoir des conséquences graves, voire fatales (S3) et le conducteur n'a aucun contrôle sur le système (C3). Le système sera qualifié ASIL D.

		Probability	Contrôlabilté		
			C1	C2	C3
Sévérité	S1	E1	QM	QM	QM
		E2	QM	QM	QM
		E3	QM	QM	A
		E4	QM	A	B
	S2	E1	QM	QM	QM
		E2	QM	QM	A
		E3	QM	A	B
		E4	A	B	C
	S3	E1	QM	QM	A
		E2	QM	A	B
		E3	A	B	C
		E4	B	C	D

Table 1.3 – Niveau de Criticité Automobile ASIL

Ces niveaux de certification permettent autant aux constructeurs qu'aux fournisseurs automobiles de s'entendre sur le niveau d'exigence lors de la réalisation d'un système. Certains fournisseurs vendant des systèmes complets certifiés ASIL D doivent prouver que leur produit répond à ces exigences. Les exigences peuvent être accompagnées de tableaux, listant des méthodes ou techniques recommandées pour respecter l'exigence. Ces recommandations sont plus ou moins fortes. Il y a également trois niveaux de recommandation :

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

\+ : 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 1.4, extrait de l'ISO26262 part 6 expose les principes de conception d'architecture du logiciel qui doivent être appliqués pour assurer les propriétés de modularité, d'encapsulation, et de complexité faible. Lors de la conception de nos approches théoriques, il est important de garder ces exigences en tête pour la conception de nos mécanismes. Nous essayerons donc de garder des

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 1.4 – ISO26262 – Principes de conception pour l'architecture logicielle

composants de faibles tailles (exigence 1b) avec des interfaces de communications limitées et réduites à une ou deux interactions (1c et 1e) qui seront ordonnancés par messages (1f).

Bien qu'il soit important de garder en tête le contexte automobile et de comprendre la vision de son industrie concernant la sûreté de fonctionnement, celui-ci ne doit pas nous imposer ses contraintes lors de la recherche d'une solution mais doit rester présente pour que la solution proposée puisse répondre aux problèmes automobiles.

5.2 AUTOSAR

Nous allons maintenant nous intéresser au contexte et au support logiciel pour l'implémentation des systèmes automobiles et plus précisément au standard logiciel utilisé par les constructeurs pour implémenter leurs systèmes embarqués.

L'évolution croissante des applications pour véhicules conduit les systèmes automobiles à un niveau de complexité élevé, si bien qu'il est maintenant possible de trouver plus d'une centaine de calculateurs embarqués dans certains modèles de voitures. Une quantité croissante de fonctionnalités est à présent réalisée en utilisant du logiciel et de nombreuses innovations actuelles (jusqu'à 90% [Broy 2005]) sont possibles grâce à celui-ci.

En outre, les véhicules sont maintenant de plus en plus connectés (les communications *car-to-car* [Baldessari 2007] ou *car-to-infrastructure* [Harding 2014] sont des sujets d'actualité auxquels les industriels s'intéressent de près. L'augmentation massive du logiciel a créé de nouveaux problèmes. En effet, des fonctionnalités totalement décorrélées peuvent désormais se retrouver sur un même calculateur et interférer les unes avec les autres [Broy 2006].

Historiquement, les calculateurs embarqués étaient développés de manière ad-hoc, c'est-à-dire que le logiciel associé à chaque calculateur était développé spécifiquement chaque fois. Ceci était possible car peu de logiciel était embarqué dans les véhicules. Cependant ce procédé rendait très difficile la réutilisation du logiciel,

ainsi que sa modification, sa maintenance ou encore l'intégration de logiciels issus de différents fournisseurs.

Afin de gérer ces problèmes, un partenariat de développement réunissant constructeurs, fournisseurs et vendeurs d'outils a été mis en place pour développer l'***AUT****omotive* ***O****pen* ***S****ystem* ***AR****chitecture* [Fürst 2009]. .

5.2.1 Architecture globale

L'objectif d'AUTOSAR est de créer et d'établir une architecture logicielle automobile ouverte et normalisée. Les objectifs principaux d'AUTOSAR sont :

- La personnalisation du logiciel en fonction des véhicules et des plates-formes diverses ;
- La prise en charge de différents domaines fonctionnels ;
- L'utilisation de composants standards (*Commercial Off-The-Shelf* - COTS) ;
- L'intégration de modules venant de différents fournisseurs ;
- La prise en charge des normes internationales de l'automobile ;
- La mise en œuvre de mécanismes de sécurité ;
- L'assurance de la fiabilité.

Pour améliorer la rentabilité et la réutilisabilité, AUTOSAR sépare le logiciel du matériel associé. Pour atteindre les objectifs techniques de modularité, d'évolutivité, de transférabilité et de réutilisabilité des fonctions, AUTOSAR fournit une infrastructure logicielle commune pour les systèmes automobiles de tous les domaines du véhicule, reposant sur des interfaces normalisées pour différentes couches présentées sur la figure 1.10.

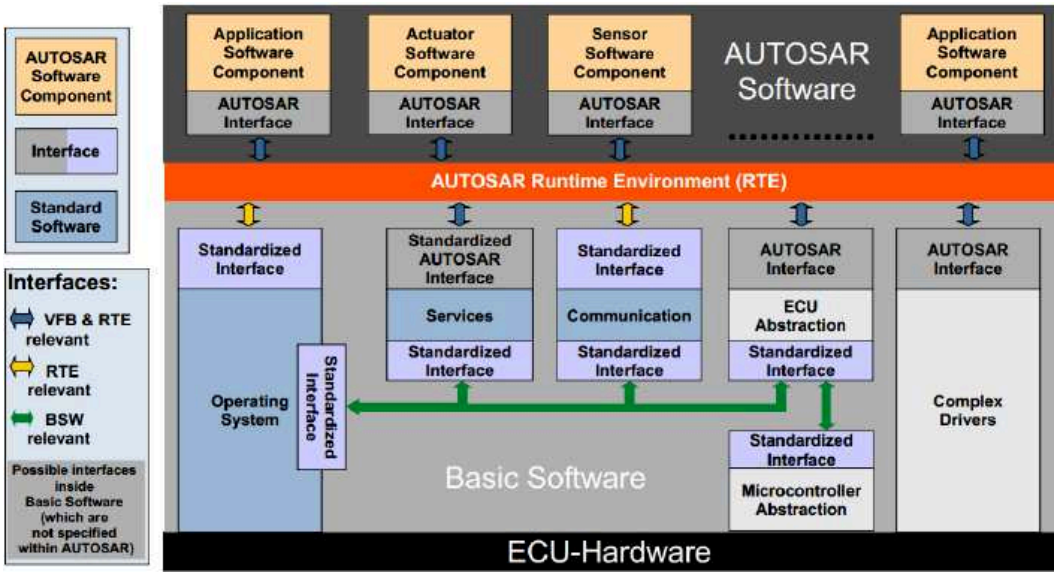


FIGURE 1.10 – Architecture logicielle d'AUTOSAR

AUTOSAR présente donc une architecture en couche qui permet d'abstraire le matériel. Il s'agit d'une architecture modulaire facilitant la construction du logiciel à partir de briques élémentaires. Ces composants applicatifs peuvent être réutilisés facilement et être portés d'une plate-forme à une autre. Il est également possible d'utiliser des COTS (*Commercial Off-The-Shelf*) pour réaliser des fonctions génériques communes à différents véhicules automobiles. Dans le standard AUTOSAR, un ECU (*Electronic Control Unit*) présente une architecture logicielle divisée en quatre couches, comme montré par la figure 1.10. La couche la plus basse (ECU-*Hardware*) correspond au matériel. Au dessus du matériel se trouve le logiciel de base, également appelé couches basses (BSW ou *Basic Software*) [Fürst 2010]. La couche la plus haute, quant à elle, est la couche applicative qui contient les différents composants logiciels (SWC) réalisant les fonctions [Bunzel 2011]. Enfin, entre la couche applicative et le logiciel de base se trouve l'intergiciel spécifique à AUTOSAR appelé ***R**un-**T**ime **E**nvironment* (RTE). Cette couche sert d'interface de communication entre les différents SWC [Long 2009].

5.2.2 Composant Logiciel

Les Software Components (SWC) correspondent au logiciel applicatif, c'est-à-dire à la partie de l'architecture qui se trouve au dessus du RTE. Un SWC est un ensemble de *runnables*, une unité d'exécution élémentaire sur AUTOSAR (P.ex. une fonction en C), de ports d'entrée/sortie qui permettent la communication avec les autres SWC via le RTE, et d'IRV (*Inter-Runnable Variables*) qui permettent aux *runnables* d'un même SWC de communiquer entre eux. Il définit également les évènements qui permettent aux *runnables* d'être déclenchés (*RTE Events*) ainsi que les *Exclusive Areas* (gestion d'accès concurrents de données).

Le SWC est un regroupement structurel qui n'a pas d'existence propre dans l'implémentation finale. Un SWC peut soit être atomique, c'est-à-dire qu'il ne peut pas être subdivisé en Software Components plus petits et doit par conséquent être assigné sur un seul ECU, soit être une composition, c'est-à-dire un regroupement de SWC connectés entre eux.

Un *runnable* peut être défini comme une séquence d'instruction qui peut être déclenché par le RTE. Ils doivent ensuite être alloués dans des tâches du systèmes d'exploitation afin d'être exécutés. La répartition des *runnables* dans les tâches du système d'exploitation est totalement indépendante de leur répartition dans les SWC. D'autre part, un *runnable* peut être soit périodique, soit non périodique. Un *runnable* peut soit s'exécuter lors de la réception d'un évènement soit attendre un évènement pour poursuivre son exécution

5.2.3 RTE & Virtual Functionnal Bus

Dans AUTOSAR, une application est modélisée par un assemblage de SWC [Bo 2010]. Le RTE est l'interface concrète qui permet aux différents SWC de communiquer non seulement entre eux mais également avec le Basic Software.

Le VFB (*Virtual Functional Bus*) permet la conception des fonctions indépendamment du matériel. Il permet l'abstraction des connexions entre les différents composants logiciels peu importe leur localisation. L'existence du VFB est purement conceptuelle. Le RTE est l'implémentation concrète du VFB au niveau du calculateur. La figure 1.11 illustre la différence entre le niveau VFB et le niveau RTE.

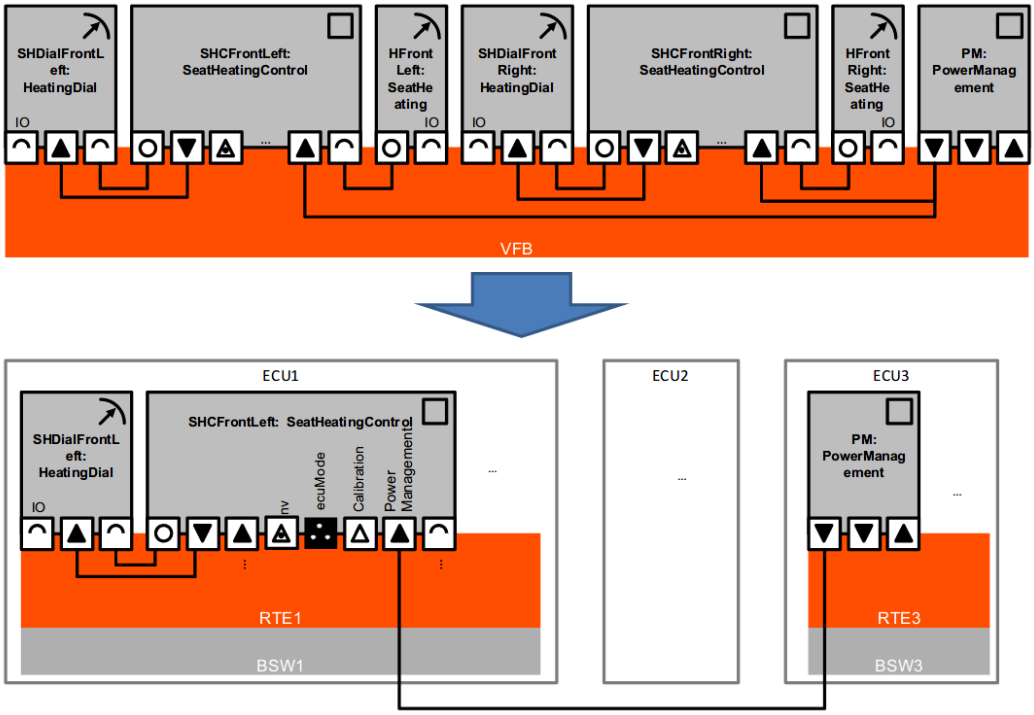


FIGURE 1.11 – Fonctionnement du RTE

La partie haute de cette figure montre différents composants qui communiquent via le VFB. C'est une vue architecturale qui montre les différents SWC présents dans le système. Il n'y a pas de prise en compte de la localisation de ces SWC. Dans la partie basse, les SWCs sont alloués sur les différents ECU disponibles au cours du processus de développement. Le RTE de chaque ECU doit ensuite être généré. Quel que soit l'emplacement physique des SWC, comme la communication a été prévue de manière conceptuelle, le RTE prend en compte, lors de sa génération, l'emplacement des SWC : du point de vue les SWC, ils communiquent indépendamment de leur allocation sur un calculateur.

AUTOSAR dispose d'un standard graphique (comme l'UML) pour définir les différentes communications. On peut distinguer sur cette figure que les deux principales communications utilisées sont les communications **(1)** de type *Sender/Receiver* équivalant à un *Publish/Subscribe* sur ROS et représentées par des flèches noires, **(2)** de type *Client/Server* et représentées par un demi-cercle blanc et un cercle blanc. De plus, le type de SWC est définie par un symbole en haut à droite :

le carré pour informer qu'il s'agit d'un SWC atomique (P.ex. le SWC *PowerManagement* à droite) et la flèche d'un capteur/actionneur (P.ex. le SWC SHDialFrontLeft à gauche).

Le RTE est souvent généré automatiquement en fonction des interactions entre SWC après la conception de l'application. Il implémente les besoins en communication des composants et les évènements qui déclenchent l'exécution des *runnables*. Il faut s'assurer, après génération, que le code fourni est bien conforme aux communications définies dans le VFB.

La fonction du RTE est de permettre la communication entre les différents SWCs mais aussi avec le Basic Software. Le RTE dépend du type d'applications ainsi que du BSW disponible installées sur l'ECU. Il n'est donc pas générique, que ce soit du point de vue fonctionnel ou du point de vue de la portabilité.

5.2.4 Basic Software & OS

Le Basic Software contient de nombreuses briques logicielles comprenant les drivers, les services d'accès à la mémoire ainsi que le système d'exploitation. Les applications automobiles devant répondre à des contraintes de temps-réel, AUTOSAR OS est basé sur le standard OSEK/VDX [Zahir 1998].

AUTOSAR OS applique un ordonnancement à priorité fixe (chaque tâche est paramétrée avec une priorité qui ne varie pas au cours de l'exécution), offre une protection contre un mauvais usage des services de l'OS et est capable de gérer les interruptions. Ces dernières doivent avoir des priorités supérieures à celles des tâches. Ces caractéristiques principales de l'OS associé au standard sont largement héritées de OSEK/VDX. Dans le modèle architectural proposé par AUTOSAR, l'implémentation d'un composant logiciel se fait par l'intermédiaire de l'allocation de chacun des *runnables* qui le composent à une ou plusieurs tâches de l'OS.

Nos FTM étant des mécanismes purement logiciels, nous ne détaillerons pas plus le basic software ainsi que le fonctionnement spécifique du système d'exploitation. Il existe des implémentations en "code source ouvert" (code source mis à la disposition du public) d'OSEK/VDX comme Trampoline [Bechennec 2006] et qui permettent d'appréhender les concepts de cet OS temps-réel.

5.3 Mise à jour à distance et Adaptation

Nous distinguons deux types de modification du logiciel. La mise à jour à distance qui correspond au remplacement du logiciel obsolète ou l'ajout de fonctionnalité. L'adaptation du logiciel qui modifie, pendant l'exécution le logiciel (changement du graphe d'exécution, du graphe de communication).

La mise à jour du logiciel à distance dans les systèmes embarqués automobiles correspond à une étape importante afin, d'une part d'améliorer l'assistance aux conducteurs et le confort des passagers, et d'autre part de faciliter les modifications logicielles pour le constructeur automobile. Cette mise à jour est donc cruciale pour le constructeur pour assurer la maintenance (correctifs de version, élimination de

fautes...) et l'évolution (nouvelles capacités) de ses véhicules sans le coût de retour au garage.

Les mises à jour à distance traitées ici se concentrent sur le logiciel applicatif des calculateurs ainsi que des mécanismes de tolérance aux fautes qui lui sont rattachés. Le problème est donc, premièrement, d'être en mesure de faire les mises à jour en ajoutant tous les mécanismes nécessaires, deuxièmement, de s'assurer que ces mises à jour n'impactent pas négativement le fonctionnement du système et, troisièmement, de faire des mises à jour partielle, c'est-à-dire, de ne mettre à jour que le code ciblé et non tout l'ECU. Ainsi, il s'agit d'un problème complexe qui influence de nombreux aspects du développement logiciel.

Concernant l'adaptation, comme montré dans [Martorell 2014], le standard AUTOSAR ne permet pas une modification simple du logiciel. Une pré-réservation des espaces est strictement nécessaire a priori que cela soit pour charger un SWC et une communication. Une nouvelle plate-forme logicielle, dites plate-forme adaptative ou Adaptive AUTOSAR, est actuellement en cours de développement pour permettre la mise à jour et l'adaptation des systèmes automobiles (P.ex. les ADAS) ainsi que les FTM.

Une partie très importante de notre travail sera donc l'analyse et la critique de cette nouvelle plate-forme et sa compatibilité avec l'AFT.

Nous avons jusqu'à maintenant abordé les notions nécessaires à la compréhension de cette thèse ainsi que les différents domaines sur laquelle elle s'appuie. Nous allons maintenant définir notre positionnement face au problème définit dans l'introduction et expliquer notre démarche scientifique pour répondre à ce problème.

6 Démarche Scientifique

L'enjeu des travaux proposés dans cette thèse est de pouvoir mettre à jour les systèmes informatiques, de les garder sûr de fonctionnement et d'avoir un impact minimum sur le système. Le contexte applicatif est celui des systèmes embarqués dans l'automobile.

Nous allons, dans cette section, affiner la problématique générale et en tirer les questions essentielles auxquelles cette thèse répondra. Nous expliquerons également la façon d'y répondre en explicitant notre démarche scientifique et enfin nous finirons par les points qui ne seront pas abordés dans cette thèse et qui sont autant de pistes de recherche ou bien qui sont aujourd'hui du ressort de l'ingénierie.

6.1 Problématiques

Tout au long de ce chapitre nous avons posé les bases pour répondre à deux questions : Comment concevoir une architecture logicielle pour adapter et mettre à jour des mécanismes de tolérance aux fautes ? Comment projeter cette architecture aux systèmes embarqués automobiles ?

Nous avons, dans la section 2 de ce chapitre, expliqué les concepts de l'AFT et nous en avons tiré des exigences pour pouvoir la mettre en place. Néanmoins, bien que nous ayons répondu à "qu'est-ce que l'AFT ?", nous n'avons pas répondu à la question "comment ?". Et de ce "comment" nous en tirons finalement plusieurs problématiques.

La première sous-problématique concerne l'implémentation sous ROS du concept *Before - Proceed - After* (BPA) proposé dans différents travaux et évoqué en section. Cela nous amène à nous poser cette question :

1 - Peut-on concevoir et implémenter des FTM adaptatifs sous ROS à partir du schéma de conception *Before - Proceed - After* ?

La deuxième sous-problématique dans cette thèse concerne la granularité de notre approche. Les composants du BPA peuvent contenir plusieurs actions qui peuvent être communes à différents FTM. La décomposition des mécanismes selon le schéma de conception BPA n'est pas optimale concernant la réutilisation du code et l'utilisation de COTS.

2 - Comment affiner la granularité pour permettre une meilleure réutilisation de code ?

La troisième sous-problématique concerne l'utilisation des ressources dans notre première approche. L'utilisation de composants sous ROS et leur projection sur des processus entraîne une surconsommation des ressources.

3 - Peut-on créer des FTM adaptatif avec une meilleure utilisation des ressources ?

Le dernier point et non le moindre est celui du contexte automobile. Pour l'instant ces questions ne prennent pas en compte le standard AUTOSAR ni la nouvelle plate-forme adaptative. Il nous faut donc intégrer les contraintes qui nous amène à une question essentielle :

4 - Nos approches d'implémentation de FTM adaptatifs sont-elles compatibles avec le contexte automobile ?

Ces quatre questions essentielles tracent les contours de notre étude et motivent les travaux que nous avons menés pour y répondre.

6.2 Déroulement des travaux de thèse

Les questions posées précédemment seront développées tout au long de cette thèse dans les différents chapitres. Il semble quand même important de détailler un minimum ce que nous vous présenterons.

Pour répondre à la question **1**, nous allons devoir reprendre le principe du *Before - Proceed - After* défini en section 2 de ce chapitre, le compléter avec les interfaces de communication et autres composants nécessaires, définir comment faire une adaptation et une composition, et également respecter les contraintes liées à ROS. Nous verrons donc dans notre Chapitre 2 comment nous avons développé une approche par composants substituables pour l'adaptation et la composition de FTM et comment nous avons implémenté sous ROS trois mécanismes permettant d'illustrer l'approche.

La question **2** amène un point important dans la réalisation de cette thèse, le changement de granularité dans notre approche. Nous ne nous intéressons plus aux blocs du BPA mais aux actions qu'ils remplissent. Ces actions élémentaires qui constituent les blocs peuvent être catégoriser indépendamment du FTM. Cela signifie que les mécanismes peuvent être définis comme une succession d'actions élémentaires, qui devront être spécifiées en fonction du mécanisme et de l'application à laquelle il est rattaché (type du mécanisme, type de message etc...). Le Chapitre 3 aura pour objectif de définir les catégories auxquelles nous pouvons rattacher ces actions élémentaires ainsi que les sous-catégories qui les composent en fonction des choix de conception et d'implémentation.

En ce qui concerne la problématique **3**, nous nous intéresserons à résoudre un problème de ressource présent dans l'approche par composants substituables. Avec ROS, chaque composant est un nœud ROS qui est projeté sur un processus UNIX à l'exécution. Avec une application contenant un modèle de fautes complexe, un mécanisme peut être constitué de 17 composants et autant de communications TCP/IP. Nous concevrons nos mécanismes non plus comme un graphe de composants mais comme un graphe d'actions élémentaires de SdF. Nous étudierons également les éléments permettant de manipuler et d'ordonnancer ce graphe. De cette façon, ce ne sont plus des composants mais des actions élémentaires que nous adapterons. À

l'instar du Chapitre 2, le Chapitre 4 s'intéressera à la conception et l'implémentation d'une approche par objets ordonnançables.

Enfin, c'est dans l'optique du changement du support d'exécution pour les systèmes embarqués automobile, avec l'apparition de cette nouvelle plate-forme *Adaptive* AUTOSAR, que s'articule notre dernier chapitre. La question **4** soulève deux travaux de recherche cruciaux : **(1)** faire une analyse critique de cette nouvelle plate-forme, **(2)** étudier la compatibilité de notre approche avec Adaptive AUTOSAR mais aussi AUTOSAR. En effet, il faut analyser si l'incompatibilité entre le support d'exécution et la mise à jour et l'adaptation du logiciel nous permet quand même d'implémenter nos mécanismes sous forme de COTS.

A la fin de cette thèse, nous aurons développé deux méthodes d'implémentation de l'AFT avec des niveaux de granularités différents, une catégorisation des fonctions de sûreté de fonctionnement ainsi que des exemples d'implémentation de mécanismes sur ROS, AUTOSAR et Adaptive AUTOSAR.

6.3 Ce que n'aborde pas cette thèse

Maintenant que nous avons examiné le contexte et les notions essentielles à la compréhension de cette thèse, il nous semble important de préciser ce qui ne sera pas abordé dans cette thèse.

Dans un premier temps, il existe des centaines de possibilités pour définir des mécanismes de sûreté de fonctionnement, et autant de façon différente de les implémenter. Nous avons choisi 3 mécanismes complets et complexes permettant une bonne représentation des différents travaux. Cette thèse ne fournit donc pas une liste de mécanismes exhaustive (ce qui est impossible) répondant à tous les cas imaginables mais quelques mécanismes qui permettent d'illustrer la méthode.

Dans un deuxième temps, toutes les fonctionnalités de ROS ne sont pas utilisés et/ou montrés dans cette thèse. Ce *middleware* en fournit un nombre assez conséquent et un manuscrit complet sur le sujet serait nécessaire.

Dans un troisième temps, l'étude de la classification des mécanismes par rapport à leur fonctions élémentaires n'est pas complète. C'est à partir de l'étude d'une dizaine de mécanismes vus dans différents travaux que nous avons pu l'établir mais elle n'est en aucun cas suffisante.

Dans un dernier temps, nous ne fournissons pas les détails d'implémentation de nos mécanismes sur la nouvelle plate-forme d'AUTOSAR, celle-ci étant toujours en cours de développement au moment où cette thèse est écrite. Vous trouverez néanmoins une analyse critique de la plate-forme et des possibilités, à cet instant, d'y appliquer nos approches de conceptions de FTM adaptatifs.

7 Synthèse

Ce premier chapitre nous a permis d'établir les fondements sur lesquels se repose cette thèse. Nous avons pu y avoir des notions diverses allant de la sûreté de fonctionnement à l'automobile, en passant par les approches à composant.

Ce qu'il faut retenir finalement de ce chapitre sont les caractéristiques et les exigences de l'AFT à savoir la capacité d'adapter les mécanismes de tolérance aux fautes pour assurer la confiance que peut avoir un utilisateur en un service donné peut importe le changement de contexte. De plus, lors d'une adaptation, la modification des mécanismes doit avoir un impact limité sur la disponibilité de l'application fonctionnelle.

Nous étudierons donc dans les prochains chapitres comment nous avons respecté la *Separation of Concern*, le *Design for Adaptation* ainsi que la nécessité d'une mise à jour des mécanismes au travers de deux méthodes, une à gros grain et une à grain fin. Nous examinerons également les détails d'implémentation de ces méthodes sous ROS ainsi que dans le contexte automobile.

Nous allons détailler dans le prochain chapitre la première méthode à gros grain s'appuyant sur le modèle B-P-A. Le Chapitre 2 se divisera en quatre parties majeures. La première sera dédiée à la conception théorique des FTM pour assurer l'adaptation et la composition. La seconde illustrera l'approche conçue par une implémentation de trois mécanismes choisit sous ROS. La troisième partie sera consacrée aux méthodes utilisées pour effectuer trois transformation, la reconfiguration d'un composant, l'adaptation d'un FTM vers un autre et la composition entre deux FTM. Enfin, la quatrième partie évaluera la conception et l'implémentation de l'approche et statuera sur le choix de ROS comme candidat possible pour mettre en œuvre l'AFT.

Chapitre 2

Adaptation par composant substituable

Sommaire

Introduction

Comme énoncé dans le chapitre précédent, le but de ce chapitre 2 est de trouver une manière de concevoir des mécanismes de tolérance aux fautes (FTM) et un support d'exécution qui soient compatibles avec l'AFT. Le principe de l'AFT est l'adaptation de FTM lorsque le contexte, pour lequel ils ont été choisis, change (p. ex. modification des hypothèses liées à la sûreté de fonctionnement, changement de la version de l'application).

Pour cela, nous allons partir des travaux précédents sur le *Design For Adaptation* (D4A), à savoir la décomposition des FTM en *Before - Proceed - After* (BPA), compléter ce schéma conceptuel et l'implémenter sous ROS. De plus, il nous faudra choisir des mécanismes auxquels nous appliquerons notre méthode pour illustrer la transition d'un FTM vers un autre et la composition de deux FTM.

Cela nous permet donc de distinguer trois parties : **(1)** la conception d'une approche à gros grain manipulant les blocs BPA, **(2)** l'implémentation des mécanismes sur ROS **(3)** l'implémentation des méthodes de transitions entre mécanismes sur ROS et **(4)** la validation de cette approche à travers un scénario impliquant un changement d'hypothèses amenant à une adaptation et une composition. Le chapitre 2 sera divisé en quatre sections principales.

La première section de ce chapitre se concentrera sur la conception des mécanismes pour pouvoir mettre en application le *Design for Adaptation* (D4A) qui ne se limite pas qu'au BPA. Il va donc falloir exprimer ce qui compose un FTM, ce que signifie une transition entre deux mécanismes et quelles sont les conditions pour la réaliser. De plus, nous illustrerons notre approche à travers trois FTM de notre choix.

La deuxième section présentera l'implémentation de ces mécanismes sur ROS en mettant en avant et en détaillant les techniques utilisées. Ainsi nous expliquerons les différentes caractéristiques de ROS pour mettre en œuvre nos FTM.

La troisième section se focalisera sur les capacités de ROS à être un support d'exécution pour l'AFT. Notre but va être d'implémenter non seulement les mécanismes mais également tous les composants nécessaires pour l'adaptation et la composition des FTM.

La quatrième section concernera l'établissement d'un scénario pour valider la méthode développée précédemment. Nous partirons d'un cas simple avec un Client et un Serveur auquel nous y aurons attaché un mécanisme de tolérance aux fautes. Nous décrirons comment nous avons mis en place l'adaptation et la composition de deux FTM. Enfin nous ferons une analyse critique du support d'exécution et analyserons ses capacités à mettre en œuvre l'AFT.

1 Conception théorique d'un FTM

Nous aborderons dans cette section le modèle de conception des mécanismes de tolérance aux fautes en s'appuyant sur des travaux précédent [Stoicescu 2013]. Pour commencer, nous étudierons la projection des mécanismes de tolérance aux fautes sur les composants *Before - Proceed - After* (BPA) ainsi que leur catégorisation en fonction des hypothèses issues des analyses de risques.

Nous définirons ensuite ce que signifie l'adaptation et la composition dans le cas de notre schéma de conception et nous ajouterons à notre mécanisme les composants nécessaires pour les réaliser. Nous sélectionnerons enfin les mécanismes pour l'implémentation sous ROS de l'AFT en définissant les actions qu'exécuteront les différents composants contenus dans les mécanismes.

1.1 Projection des mécanismes

Le but de nos mécanismes de tolérance aux fautes est d'assurer le bon fonctionnement de l'application à laquelle ils sont rattachés selon les spécifications non-fonctionnelles associées. Lors de la conception de l'application, des analyses de risques vont être mise en place pour déterminer les hypothèses servant à déterminer les mécanismes qui répondent aux exigences de sûreté de fonctionnement. Ces hypothèses sont définies dans la section 2.2 comme le modèle de faute (FT), les caractéristiques applicatives (AC) et les besoins en ressources(RS).

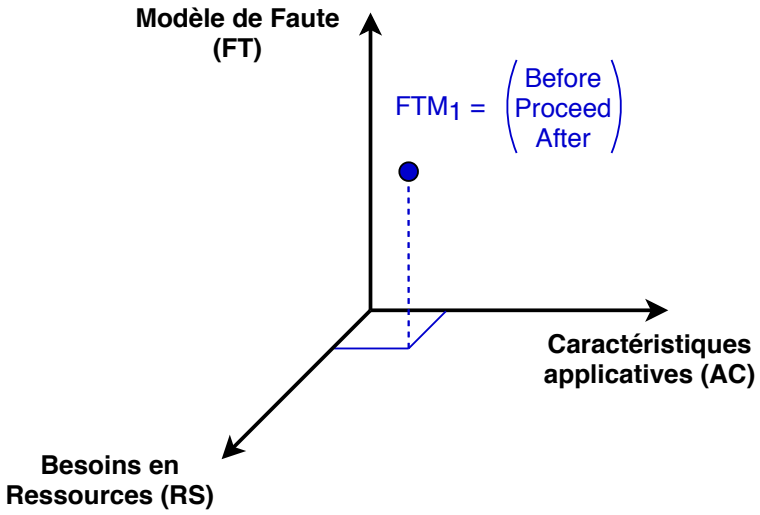


FIGURE 2.1 – Espace de définition d'un FTM

La figure 2.1 illustre l'espace de définition d'un mécanisme en fonction des hypothèses FT, AC et RS. C'est une représentation idéalisée qui permet d'illustrer la sélection d'un FTM à un instant t. Chaque FTM a un volume de validité et si le contexte applicatif change et sort de ce volume, nous devons utiliser un autre FTM. Ces volumes de validité peuvent se superposer, laissant le choix au concep-

teur de choisir parmi plusieurs FTM possibles. Une version plus réaliste sous forme matricielle est présentée dans les travaux [Excoffon 2016].

Toutes les hypothèses n'ont pas le même rôle et chacune d'entre elle affine la liste de mécanismes possibles. L'hypothèse FT permet de définir une liste de mécanismes capables de tolérer un type de faute (p.ex. faute par crash). Des caractéristiques applicatives AC dépend le choix et la conception des mécanismes (p.ex. impossible d'implémenter un FTM manipulant l'état s'il n'est pas accessible). Les ressources disponibles définissent le coût à payer pour tolérer le type de faute. Si nous souhaitons tolérer les fautes par crash sans perte de disponibilité de l'application, il faut de la redondance matérielle et donc plusieurs calculateurs. Sinon il faut redémarrer l'application dans un état stable, après réparation du support d'exécution.

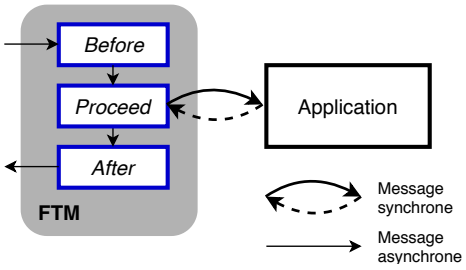


FIGURE 2.2 – Projection d'un FTM en *Before* - *Proceed* - *After*

L'avantage de l'approche BPA présentée sur la figure 2.2 est la capacité à projeter n'importe quel mécanisme de tolérance aux fautes sur trois composants. On considère toute application ou FTM comme un graphe de composants qui va non seulement définir les composants présents dans le système mais également les interactions entre les composants. On se place du côté de l'application à laquelle est rattaché le mécanisme et on définit les différents composants en fonction de leur ordre d'exécution :

- ***Before*** **:** Réunit les fonctions exécutées en amont de l'application ;
  ⟶ Réplication : *Before* = Synchronisation des répliques en entrée ;
  ⟶ Assertion : *Before* = Vérification des données d'entrée.
- ***Proceed*** **:** Coordonne l'exécution de l'application ;
  ⟶ Traitement des données : *Proceed* = Encapsulation des données d'entrée et de sortie de l'application.
- ***After*** **:** Réunit les fonctions exécutées en aval de l'application.
  ⟶ Réplication active : *After* = Synchronisation des répliques en sortie ;
  ⟶ Réplication passive : *After* = Sauvegarde et Restauration d'état.
  ⟶ Assertion : *After* = Vérification des données de sortie

On peut remarquer une différence fondamentale entre les composants. Le *Before* et l'*After* sont définis en fonction du mécanisme alors que le *Proceed* est défini en fonction de l'application. Pour que notre démarche soit transparente pour l'application, le *Proceed* du FTM est spécifié en fonction du type de données transmises à l'application. La mise en œuvre de l'approche BPA est soumise à plusieurs

problèmes : **(1)** l'interception transparente des requêtes émises par l'application (appelée Émetteur), **(2)** la coordination des composants BPA, et **(3)** l'indépendance des *Before* et *After* face aux données transmises à l'application réalisant la fonctionnalité (appelée Récepteur). En effet, nous souhaitons que les *Before* et *After* soient indépendants du type de données transmises pour en faire des COTS interchangeables.

Pour résoudre le premier problème **(1)** nous nous appuyons sur les architectures classiques de programmation distribuée. Dans celle-ci, l'Émetteur envoie ses requêtes à travers un *proxy* qui sert de passerelle à l'interface du Récepteur, donc de l'application. Dans le graphe de composant, nous introduisons le composant *Proxy* qui va assurer la transparence de la communication du point de vue de l'Émetteur. Concernant le deuxième problème **(2)**, nous intégrons un composant local, que nous nommons *Protocol*, en amont du BPA pour coordonner l'exécution du graphe BPA lors d'une requête de la part de l'Émetteur. Ce composant va servir de superviseur à chaque trio BPA. Il y a donc autant de *Protocol* que de BPA et nous pouvons considérer qu'un FTM n'est plus projeté sur trois mais quatre composants P-BPA . Le troisième problème **(3)** nécessite d'uniformiser les communications internes au mécanisme pour assurer la généricité des composants *Before* et *After*. Ceux-ci ne doivent dépendre que du FTM rattaché à l'application et non du type de messages échangés. Le *Proxy* devra effectuer une opération d'encapsulation des données en entrée pour les stocker dans un message générique et le *Proceed* devra effectuer l'opération inverse pour transmettre la requête à l'application.

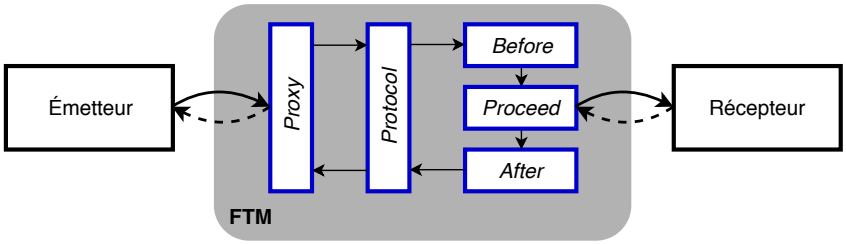


FIGURE 2.3 – graphe de composants d'un FTM

Comme le montre la figure 2.3 un FTM est un assemblage des composants *Proxy* et P-BPA. Néanmoins, il se base également sur des services communs de tolérance aux fautes qui ne font pas parties de ces composants mais qui sont présents pour assurer le bon fonctionnement du mécanisme (p.ex. mémoire locale, vote, consensus, *watchdog*). Prenons l'exemple du LFR, présenté sur la figure 2.4, pour illustrer le graphe et les fonctionnalités de chaque composant du calculateur Primaire.

- ***Proxy* :** Encapsule la requête dans un message, y ajoute un identifiant puis transfère le message au *Protocol* ;
- ***Protocol* primaire :** Vérifie avec l'identifiant si c'est une requête dupliquée : Si oui, il renvoie la réponse stockée, sinon il transfert le message à son *Before* ;
- ***Before* primaire :** Synchronise les répliques en transférant le message à chaque autre *Before* (consensus) puis le transmet à son *Proceed* ;

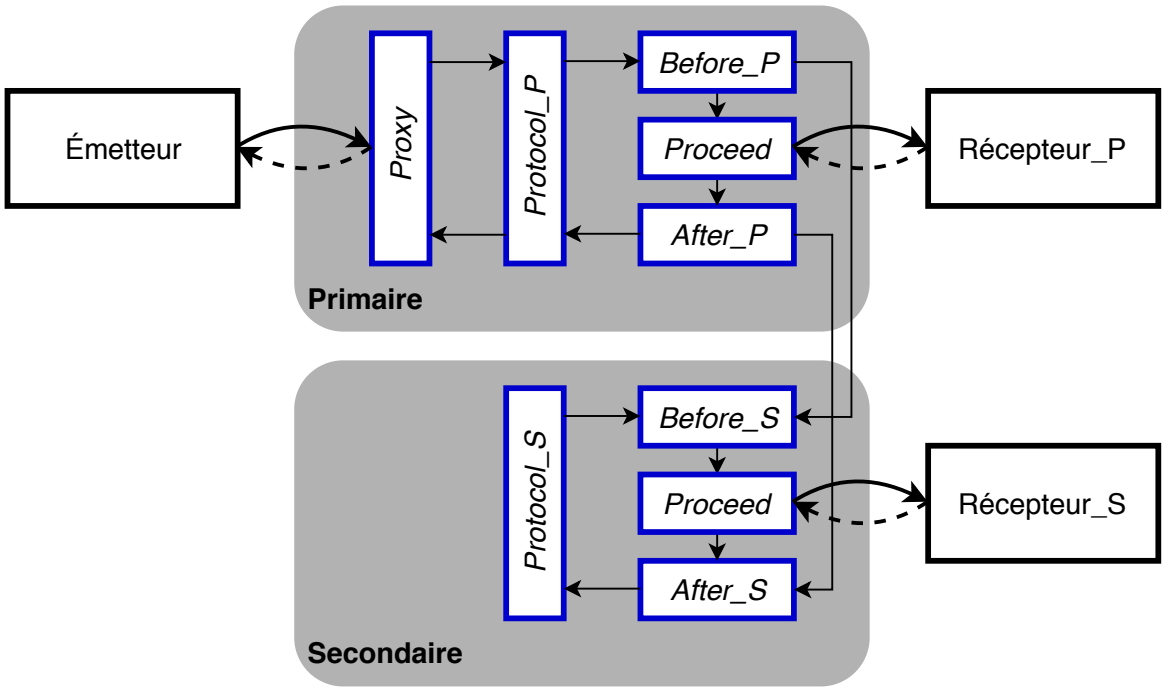


FIGURE 2.4 – Représentation du fonctionnement d'un LFR

- ***Before* secondaire :** Réceptionne le message contenant la requête puis l'envoie à son *Proceed* ;
- ***Proceed* :** Identique pour les répliques, il extrait la requête et l'envoie à son Récepteur. Sur retour de ce dernier, il encapsule la réponse dans un message et l'envoie à son *After* ;
- ***After* primaire :** Synchronise les répliques en envoyant un message de fin d'exécution à chaque autre *After* puis transfère la réponse à son *Protocol* qui stocke la réponse puis la renvoie à l'Émetteur.
- ***After* secondaire :** Réceptionne le message de synchronisation puis envoie sa réponse à son *Protocol* qui la stocke.
- **Services communs de TaF :** *Watchdog* pour chaque réplique, *Recovery* pour sélectionner une réplique de reprise, mémoire locale stable [1] pour stocker la réponse.

Maintenant, nous allons nous intéresser au cœur de l'AFT, l'adaptation et la composition des FTM. L'adaptation consiste à manipuler le graphe de composants des FTM alors que la composition consiste à agréger ces graphes.

1. Nous utilisons ici le terme mémoire local stable sans faire référence aux nombreux travaux et architectures spécifiques de mémoire stable. Nous y référons au sens de la capacité de cette mémoire à conserver, de manière externe et persistante, les informations d'un composant défaillant.

1.2 Transition entre les mécanismes

Comme nous avons pu le voir dans la partie précédente, les FTM sont définis en fonction de graphe de composants (*Proxy* + P-BPA) et de services communs de Tolérance aux Fautes (TaF). La transition entre mécanismes, à savoir l'adaptation et la composition de mécanismes, implique une modification du graphe de composants initial. Avant de réaliser l'une de ces transformations, une analyse en amont est effectuée pour déterminer les composants communs qui ne seront pas modifiés et ceux qu'il faut changer.

1.2.1 Architecture générale

La conception et l'implémentation des FTM, de leur adaptation et de leur composition va nécessiter des services supplémentaires. L'architecture va comporter quatre couches :

- **Couche applicative :** Composants des FTM et de l'application ;
- **Services communs de TaF :** *Watchdog*, Horloge commune... ;
- **Services pour l'adaptation :** Moniteur, *Adaptation Engine* ;
- **Support d'exécution :** OS (p.ex. Linux) / Intergiciel (p.ex. ROS).

La couche applicative contient les composants de l'application et les composants d'un FTM, à savoir le graphe Proxy + P-BPA. Le FTM interagissant directement avec l'application, nous avons décidé de le représenter au même niveau que celle-ci. Les Services Communs de TaF sont tous les services nécessaires à assurer le fonctionnement nominal du FTM et s'exécutent en parallèle de l'application. Les Services pour l'Adaptation sont les composants nécessaires à la réalisation de la transition entre un mécanisme et un autre. Ils vont avoir un rôle essentiel pour d'une part détecter l'évolution du système amenant à une invalidation des hypothèses initiales et donc rendant le FTM caduc et d'autre part, assurer la manipulation du graphe de composants avec un impact minimal sur le système. Le *Moniteur* a pour rôle la surveillance du système. Idéalement, il a pour tâche de surveiller son évolution, détecter les changements entraînant une adaptation ou une composition et surtout définir quels mécanismes il faut mettre en place pour garder le système sûr. De plus, nous ajoutons un organisme décisionnel pour prendre la décision de modifier le graphe de composants sur détection d'un changement de contexte applicatif du *Moniteur*. L'*Adaptation Engine*, sur ordre du *Moniteur*, va mettre en place l'adaptation ou la composition en ajoutant et retirant les composants nécessaires. Nous pouvons voir ces deux éléments sur la figure 2.5. Pour assurer l'intégrité du système lors de la transition entre deux mécanismes, l'*Adaptation Engine* chargé de modifier le graphe de composants doit connaître les transitions à effectuer pour le passage d'un mécanisme à un autre, par exemple d'un PBR à un LFR.

Le *Moniteur* ne rentre pas dans le cadre des travaux de cette thèse, celle-ci se focalisant sur la transition entre les mécanismes. Plusieurs travaux sur le *Moniteur* se concentre sur la détection de changement de contexte [Cui 2019].

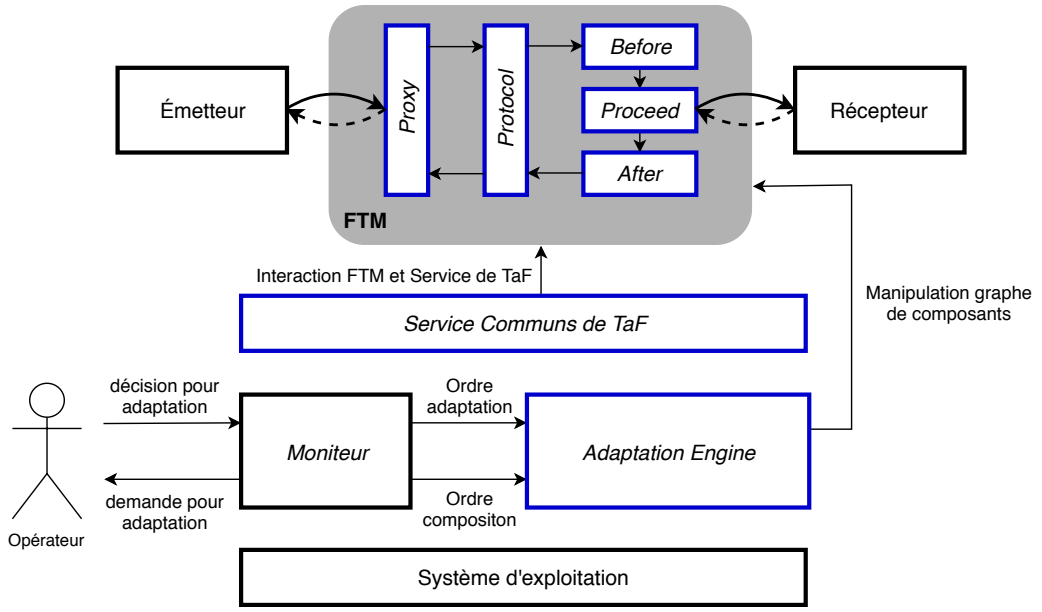


FIGURE 2.5 – Interaction pour une transformation

1.2.2 Adaptation entre deux FTM

L'adaptation consiste en la manipulation du graphe de composant. Elle répond à un changement d'une ou de plusieurs des hypothèses initiales. Certains composants sont communs à tous les mécanismes d'un certain type. Par exemple, les interfaces de communications, le *Protocol* et certains services communs de TaF sont partagés dans le cas de la réplication. De plus, les *Before* et *After* peuvent être identiques dans deux mécanismes différents. Il ne faut donc changer que les composants qui diffèrent et donc faire une modification différentielle du graphe de composant.

Pour illustrer l'adaptation, prenons l'exemple d'un LFR attaché au Récepteur. Ce dernier subit une modification permettant un accès à l'état de l'application mais occasionnant la perte du déterminisme. Par exemple, après une mise à jour de l'application, celle-ci prend en compte une variable aléatoire. Ainsi, les deux répliques peuvent donner des résultats différents avec la même requête. Le Moniteur est informé du changement des hypothèses et va donner l'ordre de mettre en place un PBR à la place du LFR. Le PBR n'a pas besoin de déterminisme car une seule application exécute la requête et l'état de celle-ci est transmis à la réplique par un message de *checkpoint*. L'*Adaptation Engine* connaît la transition entre ces deux mécanismes et va supprimer le *Before* et l'*After* du LFR de chaque réplique pour les substituer par le *Before* et l'*After* d'un PBR. A noter que l'adaptation n'est pas que le remplacement d'un FTM par un autre totalement différent. Elle peut aussi permettre la transition entre différentes implémentations d'une même stratégie de FTM. Autre exemple, nous décidons d'étoffer le PBR et d'ajouter un système de consensus permettant de choisir quelle réplique va exécuter la requête. Dans ce cas, seul les deux *Before* sont à substituer.

L'adaptation permet de conserver un système résilient face à une modification d'une des hypothèses invalidant le FTM rattaché à l'application.

1.2.3 Composition de deux FTM

La composition répond à une complexification du modèle de faute FT, c'est-à-dire, l'apparition d'un nouveau modèle de faute en plus de celui existant sans modification du reste des hypothèses. Au lieu de concevoir des composants *Before* et *After* répondant à ce modèle de faute complexe, il est plus efficace, concernant la réutilisation de code, de composer deux mécanismes répondant ensemble à celui-ci. Il faut donc agréger les deux graphes de composant. Cependant, agréger deux graphes n'est pas si simple. Il faut pouvoir, à l'exécution, ajouter un second FTM entre le premier et l'application. Cela implique de modifier les communications déjà existantes. Dans l'approche que nous proposons, nous substituons le *Proceed* par un ensemble P-BPA comme l'illustre la figure 2.6. Cette solution est rendue possible grâce à l'uniformisation des communications internes du FTM. Les communications entre le *Proxy* et le *Protocol* sont les mêmes qu'en entrée et en sortie du *Proceed*.

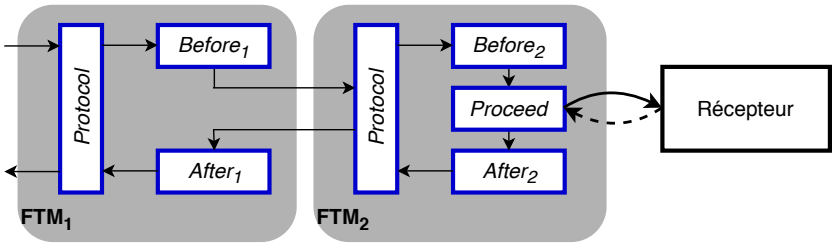


FIGURE 2.6 – Composition de deux FTM

Pour illustrer la composition, prenons cette fois ci l'exemple d'un PBR attaché au Récepteur. Le matériel vieillissant sur lequel l'application est implémentée entraîne l'apparition de fautes en valeur transitoires dues à des *bitflip* sur une RAM. Dans le cas où les *bit-flips* sont fréquents et surtout multiples, ils peuvent rendre les codes détecteur-correcteur d'erreur inefficaces (Code de Hamming). Nous savons aussi que les composants électroniques récents à forte intégration sont très sensibles aux rayonnements cosmiques dans un environnement particulier, ce qui entraîne des inversions de bits multiples sur des zones nécessaires [Rousselin 2017].

Le Moniteur ayant repéré ce problème, il interroge l'Opérateur pour valider la transition puis donne l'ordre à l'*Adaptive Engine* de composer le PBR avec un TR. Le PBR est toujours valide pour répondre aux fautes matérielles par crash mais il nous faut ajouter un TR pour répondre à ce nouveau type de faute. L'*Adaptive Engine* va donc supprimer le *Proceed* et le remplacer par un ensemble P-BPA du TR.

Il est intéressant de noter qu'il est également possible d'ajouter des fonctions liées à la sécurité pour assurer, par exemple, l'intégrité entre des communications entre machine. Comme l'illustre la figure 2.7, un *Before* en amont (côté Client) pour

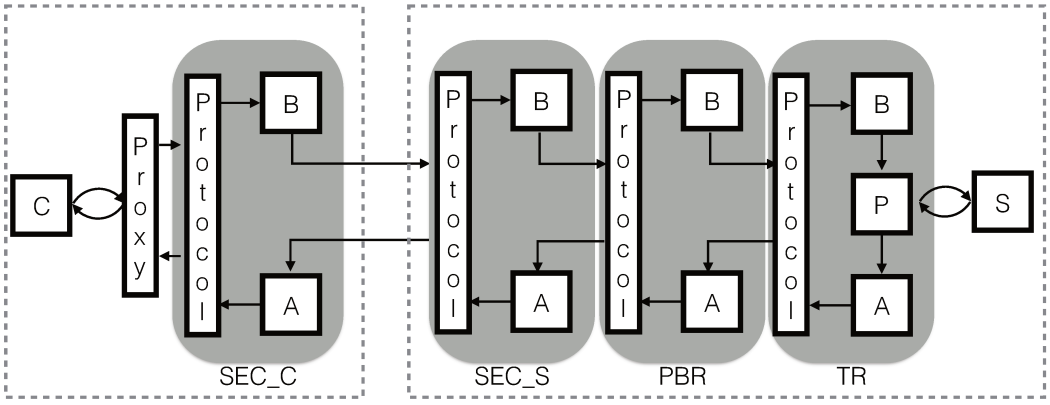


FIGURE 2.7 – Principe de composition pour ajouter du chiffrement de donnée

chiffrer la requête avec une clé de chiffrement, un *Before* en aval (côté Serveur) la déchiffrerait. Les *Afters* feraient de même pour la réponse. Plusieurs techniques de *End-to-End protection* (E2E) comme le *parity check* seraient compatibles à la place des fonctions de sécurité.

1.2.4 Contraintes de conception pour la transition

L'adaptation et la composition vont imposer plusieurs contraintes pour le choix du support d'exécution permettant la réalisation de l'AFT. Celui devra fournir : **(1)** la manipulation dynamique des composants (lancement, suppression, arrêt et redémarrage) et **(2)** la manipulation des communications entre les composants. En bref, il nous faut un contrôle total sur le graphe de composants pour pouvoir manipuler celui-ci.

De plus, pour assurer la transition entre deux FTM, il nous faut pouvoir arrêter l'exécution du graphe de composants et ainsi assurer une transition sans perte d'information ou d'état. Concernant le traitement des messages pendant une transition, plusieurs stratégies sont possibles. Nous pouvons soit bufferiser les requêter et les traiter de manière FIFO (*First In, First Out*) ou vérifier leur validité temporelle et écarter les requêtes périmées, soit ne garder que la dernière requête reçu et écarter les autres.

De surcroît, nos composants peuvent avoir des états qui peuvent être perdus lors de la perte d'une réplique. Nous pouvons périodiquement, pour ces composants, stocker leur état sur une mémoire locale et restaurer leur état lors de leur initialisation.

Enfin, l'adaptation et la composition nous imposent une standardisation des entrées-sorties des composants *Before* et *After* mais également du *Protocol* et du *Proceed*. Pour l'adaptation, il faut que tous les *Before* et les *After* aient au moins deux communications standards (Communication avec le *Protocol* et avec le *Proceed*). Pour la composition, il faut que les communications entre le *Proxy* et le *Protocol* soient identiques à celles entre le *Proceed* et le *Before* et l'*After* pour pou-

voir substituer ce dernier par un ensemble P-BPA.

Nous allons maintenant définir un ensemble de mécanismes permettant d'illustrer pleinement les concepts décrits plus tôt, à savoir la projection de mécanismes sur l'ensemble P-BPA, l'adaptation et la composition.

1.3 Ensemble de mécanismes pour la validation de l'architecture

Afin de valider les différents concepts que sont la projection sur l'ensemble P-BPA, l'adaptation et la composition, nous nous proposons d'implémenter un assortiment restreint, mais représentatif, de mécanismes. Celui-ci permet de montrer les transitions entre FTM en fonction de l'évolution des hypothèses FT, AC et RS.

En premier lieu, nous nous placerons dans l'hypothèse d'un modèle de fautes fixe (p.ex. faute par crash). Plusieurs mécanismes de réplication sont possibles et se distinguent par leur caractéristiques applicatives et leur besoin en ressource. En second lieu, nous nous placerons dans l'hypothèse d'un modèle de fautes qui évolue (p.ex. apparition de fautes transitoires en valeur). Dans ce cas de figure, le changement de FT peut conduire à la composition de deux mécanismes comme l'ajout d'une redondance temporelle (TR) à une réplication matérielle (PBR ou LFR).

La figure 2.8 nous montre un arbre de classification de FTM défini en fonction du modèle de faute, des caractéristiques applicatives et des besoins en ressource. Nous nous sommes concentrés sur les fautes matérielles car assez faciles à simuler et plus spécifiquement, les fautes permanentes par crash et transitoires en valeur.

Pour valider l'adaptation, nous avons choisi deux mécanismes de réplication, le PBR (réplication passive) et le LFR (réplication active) permettant de faire de la redondance matérielle et de tolérer les fautes permanente par crash. La réplication s'appuyant sur deux plate-formes matérielles nous permet d'illustrer l'exécution de l'AFT en cas de système distribué. Pour valider la composition, nous avons choisi un mécanisme simplexe, le Statefull Time Redundancy que l'on peut associer aux mécanismes de réplication et ainsi tolérer et les fautes matérielles permanentes en crash et transitoires en valeur. Leurs caractéristiques et décomposition ont été résumées dans le Chapitre 1 dans les tableaux 1.1 et 1.2 situés page 15.

Le but de ces exemples est de montrer que cette approche permet de facilement passer d'un mécanisme à un autre pour assurer la résilience du système tout en ayant un impact minimal sur le système. Nous devons prouver que la transition entre deux FTM est possible avec l'architecture et qu'elle permet d'accentuer la réutilisation de code lors de la conception de FTM, réduisant ainsi les temps et les coût de développement.

L'implémentation se focalisera sur la réalisation des FTM ainsi que de l'*Adaptation Engine*. Le Moniteur et l'Opérateur seront simulés à travers de fonctions *stub*. Nous allons maintenant étudier l'intergiciel ROS, qui permet donc la réalisation d'architecture à composants, et étudier son potentiel en tant que support d'exécution pour cette architecture.

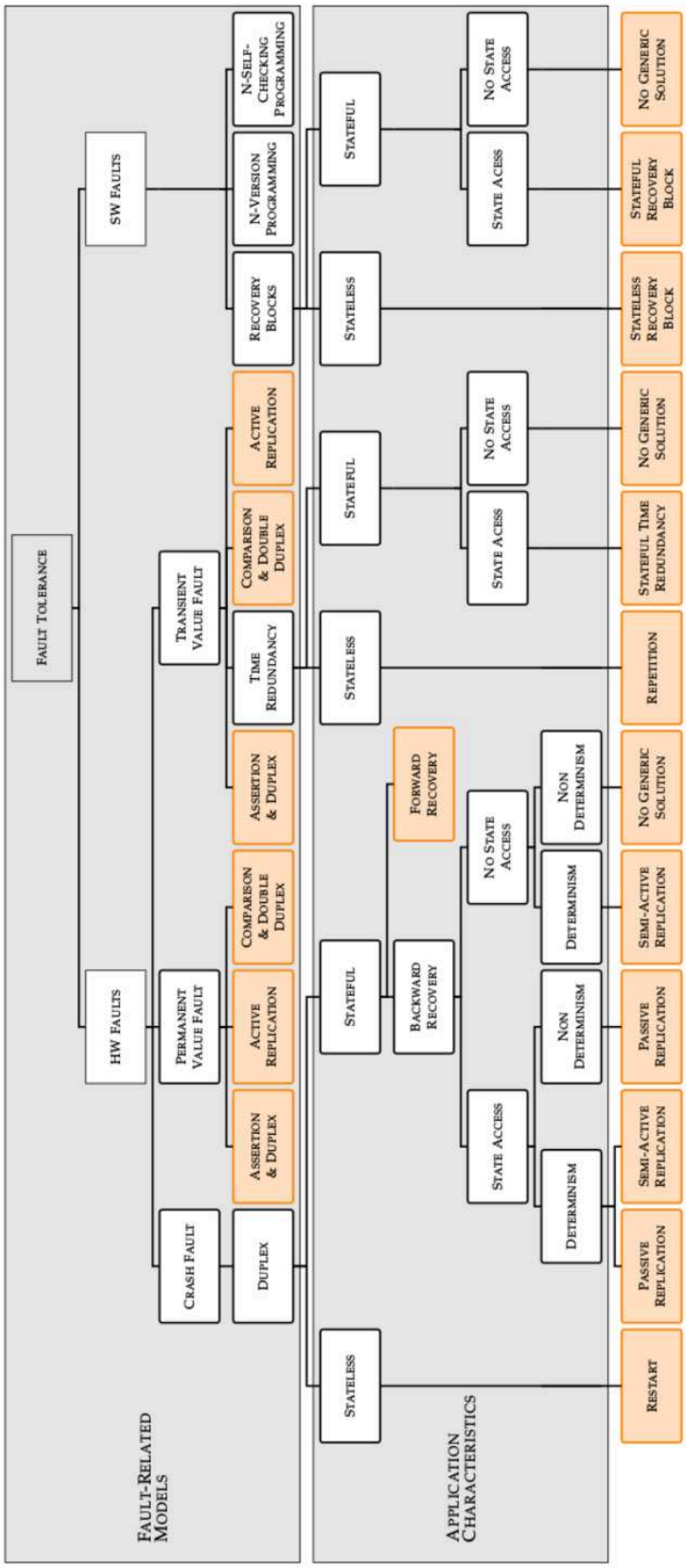


FIGURE 2.8 – Liste de FTM en fonction des hypothèses FT et AC

2 Implémentation de mécanismes sous ROS

ROS n'a pas été conçu pour faire fonctionner des systèmes critiques sûrs de fonctionnement, malgré la criticité de certains systèmes robotiques. L'objectif principal de ROS est de permettre la conception d'applications modulaires et de favoriser la réutilisation de composants. Le développement d'une application dans ROS implique la description d'un assemblage de nœuds en accord avec les architectures basées composants. A l'exécution, ces nœuds ROS sont projetés sur des processus (processus UNIX dans notre cas).

Nous avons sélectionné ROS comme support d'exécution car il possède plusieurs concepts intéressants pour la mise en place de l'AFT. Les applications sont une collection de nœuds, à savoir des objets actifs. De plus chaque nœud a son espace d'adressage propre ce qui permet d'assurer leur confinement. Il contient plusieurs moyens de communications, notamment par messages asynchrones, appelés *messages* et par messages synchrones, appelés *services*. Enfin, il est utilisé dans l'automobile par des constructeurs tels que BMW [Aeberhard 2015] pour faire du prototypage de systèmes d'aide à la conduite.

La faisabilité de l'implémentation de l'AFT sur ROS permet d'identifier les exigences nécessaires à cette mise en œuvre. Cela permet aussi d'identifier les manques du support d'exécution. Tout autre support, en particulier dans l'automobile, offrant les mêmes caractéristiques de base et qui répond aux manques identifiés, permettra une utilisation de cette technique dans l'automobile.

Nous allons, dans cette section, décrire le fonctionnement des mécanismes, leur implémentation sous ROS ainsi que leur comportement lors du passage d'un mécanisme à un autre.

2.1 Fonctionnement Nominal

Dans le Chapitre 1 avec le tableau 1.2 page 16, nous avons présenté la décomposition des mécanismes sélectionnés pour illustrer le fonctionnement de l'AFT. Nous avons illustré l'utilisation de ROS à partir d'un modèle *Publisher/Subscriber*(Émetteur/Récepteur) dans le Chapitre 1. Nous regardons maintenant comment mettre en place ces mécanismes sur ROS en partant cette fois-ci d'un modèle Client/Serveur.

Dans ROS, les deux principaux types de communication sont les *Topics*, pour les communications asynchrones, et les *Services*, pour les communications synchrones. Les communications externes au FTM seront des *Services* et celles internes au FTM des *Topics*. L'utilisation de *Topics* permet une synchronisation séquentielle des composants internes du FTM. L'approche est appliquée à une application communiquant par *services*. Une fois validée, l'approche est facilement transposable à une application communiquant par *topics*.

Tout d'abord, nous avons choisi une application assez simple mais représentative d'une demande de traitement de données avec retour du résultat et affichage de celui-ci. Le *Client* générera aléatoirement une donnée entre 0 et 255, attendra

le résultat du traitement fourni par le Serveur et l'affichera. Le Serveur fera la moyenne sur une fenêtre glissante de 10 valeurs, la 11ème valeur reçu remplaçant la 1ère etc... Nous avons donc une application déterministe (Nécessaire au LFR et au TR), le résultat retourné est censé être identique si deux composants dans le même état reçoivent la même valeur. Elle a également un état qui est un tableau à dix valeurs. De plus, nous instrumenterons notre Serveur avec des fonctions de type *setter* et *getter* pour avoir accès à cet état (Nécessaire au PBR et au TR). Notre Serveur remplit maintenant toutes les conditions pour qu'on puisse y attacher nos mécanismes.

Pour illustrer notre approche, nous prenons comme exemple le PBR qui est un mécanisme contenant des fonctions communes aux autres mécanismes. En effet, il partage une certaine similarité au niveau de la gestion d'état (sauvegarde et restauration) avec le TR et au niveau de la présence de répliques avec le LFR. À partir de la figure 2.9, nous décrivons maintenant le comportement des composants à travers l'échange d'un message entre le Client et le Serveur :

1. Le Client génère un nombre aléatoire et l'envoie au *Proxy* (*Service* clt2pxy) ;
2. Le *Proxy* ajoute un identifiant à la requête puis transfert le message encapsulant la fonction appelée et ses paramètres au *Protocol* (*Topic* /pxy2pro) ;
3. Le *Protocol_P* vérifie, à l'aide de l'identifiant, si c'est une requête dupliquée : Si oui, il renvoie la réponse stockée en mémoire (*Topic* pxy2pro), sinon il transfert la requête au *Before* (*Topic* pro2bfr) ;
4. Le *Before_P* transfert la requête au *Proceed* (*Topic* bfr2prd) ; dans le cas de cette implémentation du PBR, aucune action n'est associé à ce composant, ce qui n'est pas le cas pour le LFR par exemple ;
5. Le *Proceed* fait appel au *Service* fourni par le Serveur (*Service* prd2srv) après avoir désencapsulé le message et transfère la réponse à l'*After* (*Topic* prd2aft) ;
6. L'*After_P*, sur réception de la réponse, capture l'état du Serveur_P en faisant appel au *Service* de gestion d'état du Serveur (*Service* getState), construit un message de *checkpoint* qu'il envoie à l'*After* de la ou des répliques (*Topic* aft2aft) puis finalement renvoie la réponse au *Protocol* (*Topic* aft2pro) ;
7. L'*After_S*, sur réception du message de *checkpoint* met à jour l'état du Serveur_S en faisant appel au *Service* de gestion d'état du Serveur (*Service* setState) puis transfert la réponse au *Protocol_S* qui la stocke avec son identifiant (*Topic* aft2pro) ;
8. Le *Protocol_P* stocke également la réponse reçu de l'*After_P* puis la transfert au *Proxy* (*Topic* pro2pxy) ;
9. La réponse est ensuite transmise au Client par le *Proxy* (*Service* clt2pxy).

Nous pouvons remarquer la présence des composants *Recovery* et *Crash Detector* (CD) définis dans la section 1.1. Ces CD et le *Recovery* sont des nœuds présent

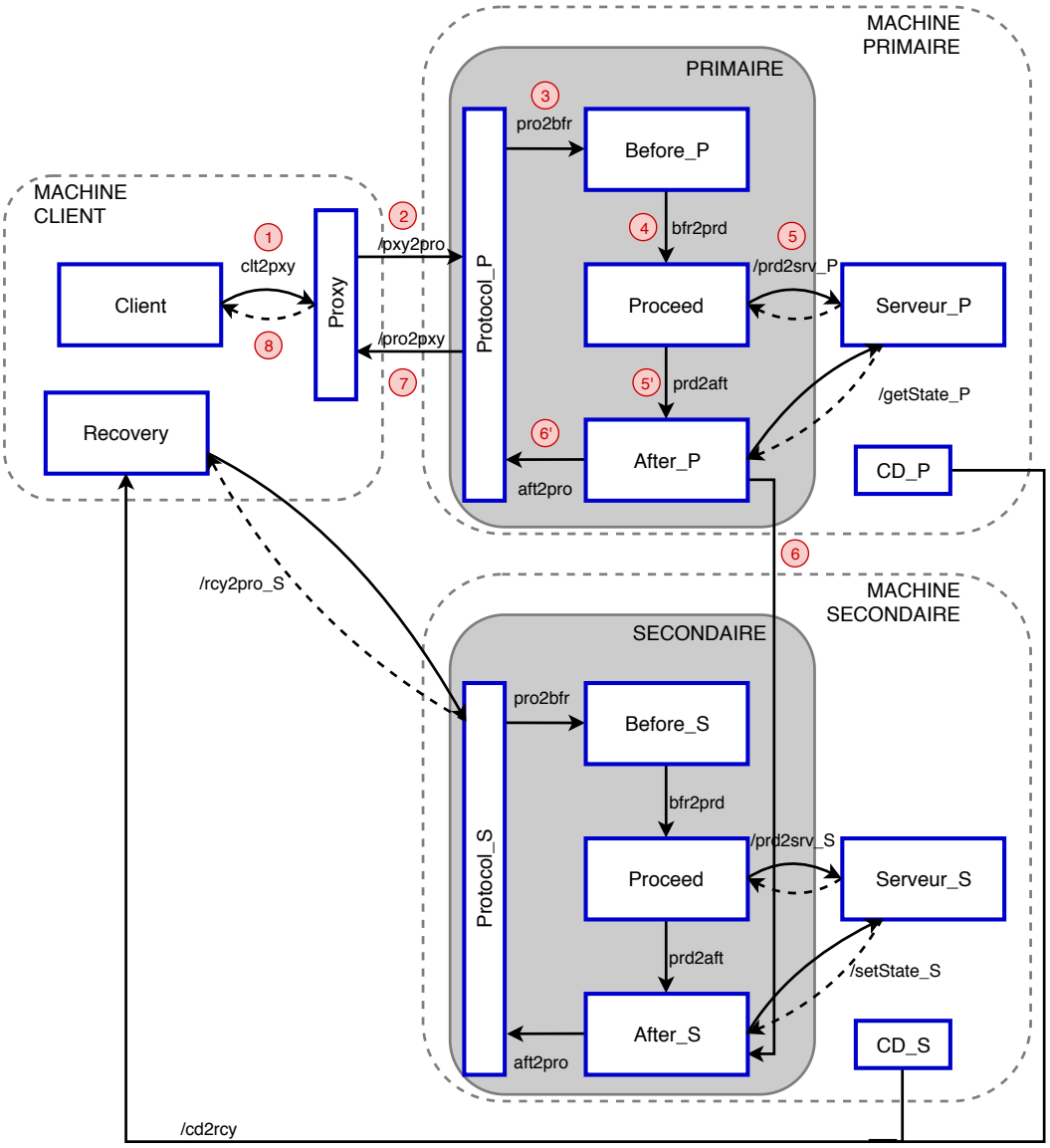


FIGURE 2.9 – Implémentation d'un PBR sous ROS

dans le cas de FTM Duplexe pour déterminer la vitalité d'un nœud ou de la plate-forme. Dans cette implémentation, les CD effectuent un contrôle de l'activité du nœud Serveur appartenant à l'autre machine (contrôle périodique effectué par un échange de message "*I'm alive*"). Si le Serveur_P ne répond pas, le CD_S envoie un *message* au *Recovery* qui appelle le *Service* /rcy2pro et établit la connexion entre le *Protocol_S* et le *Proxy*. Il est intéressant de signaler que l'exécution nominale est indépendante du format des messages applicatifs. Hormis le *Proxy* et le *Proceed* qui encapsulent les données, tout le reste est générique et peut être réutilisé pour d'autres applications.

De plus, il faut noter que nous avons mis le *Proxy* et le *Recovery* sur la même

machine que le Client. Pour le premier, ce choix est issu des architectures distribuées, le *Proxy* servant d'interface entre la Machine Client et les machines des répliques. Pour le second, c'est pour assurer l'indépendance du système de reprise en cas de crash d'une réplique avec les machines des répliques.

2.2 Implémentation d'un composant

Nous allons maintenant détailler l'implémentation d'un composant à partir d'un exemple. Nous entrons dans les détails du code que nous expliquerons au fur et à mesure. Nous avons choisi une approche inspirée des circuits intégrés en créant nos nœuds avec un boîtier (*main* en C++) qui se charge de l'initialisation et de la réception de messages, et une puce (*objet* en C++) qui contiendra les fonctionnalités du composant.

2.2.1 Détail du code source

```
1  #include "ros/ros.h"
2  #include "afterPBR.h"
3
4  int main (int argc, char **argv){
5    //Initialisation du nœud AfterPBR
6    ros::init(argc, argv, "AfterPBR");
7    // Récupération de l'argument rôle
8    int role = atoi(argv[1])
9    //Instanciation d'un objet contenant les actions de SdF
10   AfterPBR afterpbr(role);
11   //Fréquence de rafraîchissement du spinOnce
12   ros::Rate loop_rate(10000);
13   while(ros::ok()){
14         ros::spinOnce();
15         loop_rate.sleep();
16   }
17   return 0;
18 }
```

Listing 2.1 – Code du boîtier de l'*After* du PBR

La figure du listing 2.1 représente l'implémentation du boîtier de l'After du PBR. Ce code contient l'initialisation du nœud AfterPBR (ligne 6), l'instanciation de la puce (ligne 10) et la fréquence du contrôle de réception de message (fonction loop_rate & *spinOnce* lignes 12, 14 et 15). Le code du boîtier sera réutilisé presque tel quel pour chaque composant, ce qui nous permet de séparer les notions de création du nœud avec celle de la conception des fonctionnalités dépendantes du FTM. Seul le nom du nœud (ligne 6) et l'instanciation de la puce (ligne 10) sont différents pour chaque composant.

Le code de la puce étant assez conséquent, nous allons le diviser en sous-parties pour le présenter. Nous avons conçu un *package* par FTM (ici le *package* pbr) qui contient les codes sources et les *headers* des nœuds *Before* et *After*. Le *Proceed* étant commun à tous les FTM, il est dans un *package* différent. Nous avons également créé un *package* contenant toutes les communications qui ne sont pas dépendantes des FTM.

```
4  #include "ros/ros.h"
5  #include "pbr/Checkpoint.h"
6  #include "communication/Answer.h"
7  #include "communication/SrvGetState.h"
8  #include "communication/SrvSetState.h"
```

Dans un premier temps, nous avons créé deux *messages* et deux services. Comme le rappelle la figure 2.9, l'*After* du PBR doit pouvoir envoyer un message de *checkpoint* (ligne 5) et transférer la réponse au *Protocol* (ligne 6) à travers deux *Topics*, et capturer l'état du l'application (ligne 7) et mettre à jour l'état de la réplique (ligne 8). Seul le message de *checkpoint* est dépendant du PBR, les *services* de capture et de mise à jour de l'état étant commun avec le TR. Nous utiliserons les numéros des étapes de cette même figure pour faire référence aux communications.

```
10 class AfterPBR{
11   protected:
12     ros::NodeHandle nh;
13     // Création des ports de communication
14     ros::Publisher pub_pro, pub_aft;
15     ros::Subscriber sub_prd, sub_aft;
16     ros::ServiceClient clt_get, clt_set;
17
18     // Création des messages et services
19     communication::Answer msg_aft_pro;
20     pbr::Checkpoint checkpoint;
21     communication::SrvGetState get_state;
22     communication::SrvSetState set_state;
23     // Variables internes du composant
24     char answer[100],state[100];
```

Nous créons la classe *After* qui contient le *NodeHandle* qui est le point d'accès principal pour les communications (ligne 12). Nous instancions ensuite les *Publishers* (ligne 14) pour les communication (6' & 7') et (6"), les *Subscribers* (ligne 15) pour les communications (5') et (6'), et les *Clients* (ligne 16) pour les deux *services* de gestion d'état (6) et (7). Les lignes 19 à 22 détaillent l'instanciation des différents *messages* et *services* que nous utiliserons pour communiquer. Enfin, nous avons créé deux variables internes pour stocker la réponse et l'état de l'application. Pour des questions de simplification, nous ne les avons pas stockées dans une mémoire locale.

```
25  public:
26    AfterPBR(int role){
27      pub_pro = nh.advertise<communication::Answer>("aft2pro",10);
28      sub_prd = nh.subscribe("prd2aft",10,&AfterPBR::getState,this);
29      if (role == 1){
30        pub_aft = nh.advertise<pbr::Checkpoint>("/aft2aft",10);
31        clt_get =
          ↪  nh.serviceClient<communication::SrvGetState>("/getstate_P");
32      }
33      else if (role == 2){
34        sub_aft = nh.subscribe("/aft2aft",10,&AfterPBR::setState,this);
35        clt_set =
          ↪  nh.serviceClient<communication::SrvSetState>("/setstate_S");
36      }
37    };
```

Pour que les composants soient génériques, ils contiennent toutes les méthodes et les communications. Nous initialisons nos ports de communications (ligne 27 & 28) pour créer les deux *Topics* standards prd2aft (5') et aft2pro (6"). En fonction du rôle qu'aura le nœud à l'exécution, on initialise des ports particuliers. Si l'*After* est associé au Primaire (ligne 29 à 32), le nœud publie sur le *Topic* /aft2aft (6') pour envoyer le message de *checkpoint* et s'abonne au *Service* /getstate_P (6). S'il est associé au Secondaire (ligne 33 à 36), il souscrit au *Topic* /aft2aft (6') pour recevoir le message de *checkpoint* et s'abonne au *Service* /setstate_S (7).

```
40    void AfterPBR::getState(const communication::Answer msg_prd_aft){
41      std::memcpy(answer, &msg_prd_aft.answer, 100);
42      sendAnswer();
43      // Appel du service getState
44      if(clt_get.call(srv_get_state)){
45        // Création du checkpoint
46        std::memcpy(&checkpoint.state, &srv_get_state.response.state, 100);
47        std::memcpy(&checkpoint.answer, answer, 100);
48        checkpoint.id = msg_prd_aft.id;
49        // Envoi du checkpoint
50        pub_aft.publish(checkpoint);
51      }
52      // Action si échec de capture état
53      else{}
54    };
55
56    // Envoi de la réponse au Protocol
57    void AfterPBR::sendAnswer(){
58      std::memcpy(&msg_aft_pro.answer, answer, 100);
59      pub_pro.publish(msg_aft_pro);
60    };
```

Nous allons détailler les méthodes de l'*After* dans le cas d'un Primaire. Sur réception de la réponse du *Proceed* (5'), il copie celle-ci dans sa variable interne (ligne 41) puis envoie la réponse au *Protocol* (6") (ligne 42 & 57 à 60). L'avantage avec ROS est de pouvoir appeler un *service* et de tester si l'appel s'est effectué correctement (ligne 44 à 52). L'*After* capture l'état de l'application (ligne 44), puis crée le message de *checkpoint* en y insérant l'état de l'application, la réponse et l'identifiant de la requête.

```
62      // Stockage de l'état
63      void AfterPBR::storeState(const pbr::Checkpoint msg_aft_aft){
64        std::memcpy(answer, &msg_aft_aft.answer, 100);
65        std::memcpy(state, &msg_aft_aft.state, 100);
66        msg_aft_pro.id = checkpoint.id;
67        setState();
68      };
69
70      // Mise à jour de l'état du Secondaire
71      void AfterPBR::setState(){
72        std::memcpy(&set_state.request.state, state, 100);
73        if(clt_set.call(srv_set_state)){
74            sendAnswer();
75        }
76        // Action si echec mise à jour état
77        else{}
78      };
79  };
```

Dans le cas du Secondaire, sur réception du message de checkpoint (6'), le nœud va stocker l'état et la réponse dans ses variables internes (ligne 64 & 65), transférer l'identifiant de la requête et mettre à jour l'état de la réplique (ligne 67). Une fois l'état mis à jour (7), il envoie un *message* à son *Protocol* (7') contenant la réponse et l'identifiant.

Une fois le code fait, il nous faut le compiler. Pour cela, il faut passer par l'intermédiaire d'un fichier texte CMakeLists.txt.

```
1  ## Executable AFTER PBR
2  add_executable(after_pbr src/after/after.cpp src/after/after_node.cpp)
3  target_link_libraries(after_pbr ${catkin_LIBRARIES})
4  add_dependencies(after_pbr after_pbr_generate_messages_cpp)
5  add_dependencies(after_pbr ${catkin_EXPORTED_TARGETS})
```

Listing 2.2 – Compilation du nœud After du PBR

Ici, dans ce listing 2.2, "after_pbr" est le nom de l'exécutable qui a un rôle important dans le lancement des nœuds mais également pour les dépendances avec d'autres *package*. Il faut donc définir l'exécutable avec les codes sources (ligne 2),

les librairies qui vont être utilisées (ligne 3) et la génération de message dans le cas de l'utilisation de messages personnalisés (ligne 4). Si d'autres *packages* ont des dépendances avec ce code source, il faut ajouter les dépendances (ligne 5).

2.2.2 Avantages de ROS

L'un des avantages de ROS est la facilité de connecter les nœuds et de gérer leurs interactions. ROS masque tout le protocole de communication et permet de relier les nœuds à l'aide de méthodes simples d'utilisation.

L'autre avantage est l'utilisation de *namespaces* pour regrouper des nœuds, des *Topics* ou encore des *Services* sous une même appellation. Dans ROS, les nœuds, les *Topics*, les *Services* ou encore les paramètres sont enregistrés avec un nom qui leur est propre. Un *namespace* fonctionne comme un nom de famille dans le sens ou un préfixe est ajouté devant le nom du nœud. Par exemple, comme deux personnes s'appelant Matthieu peuvent être différenciée par leur nom de famille, deux nœuds *After* peuvent être différenciés par leur *namespace*. Dans notre exemple, l'*After* du Primaire s'appellera en fait /Primaire/After et celui du Secondaire /Secondaire/After. Les *namespaces* vont nous permettre de regrouper les composants d'un FTM sous un même préfixe tout en gardant des noms de nœuds génériques. De cette manière, si nous composons un PBR avec un TR, nous n'avons pas à nous soucier de possible conflit de noms des composants sur une même machine.

Nous remarquons sur la même figure 2.9 que certaines communications sont précédés d'un "/" et sont appelées globales. Le regroupement sous un même *namespace* des nœuds affecte aussi les communications. Si rien est fait, l'*After* du Primaire publierait le message de *checkpoint* sur /Primaire/aft2aft et le Secondaire souscrirait à /Secondaire/aft2aft qui sont deux *Topics* différents. Nous utilisons donc des communications globales pour les communications inter-machines et entre le FTM et l'application (2,5,6,6',7,8).

2.3 Implémentation du graphe de composant

Après l'explication de l'implémentation d'un nœud ROS, nous allons maintenant détailler l'implémentation du graphe de composants à partir du même exemple, le PBR. Nous allons ici décrire la façon dont nous lançons nos FTM à partir des fonctionnalités de ROS.

Il existe deux façons sous ROS de lancer des composants. L'une est un appel de fonction en *shell*, `rosrun`, qui utilise directement l'API de ROS et qui permet de lancer un nœud. L'autre est également un appel de fonction *shell*, `roslaunch` qui utilise des fichiers de description d'architecture formatés dans le langage à balise XML pour lancer une application complète.

2.3.1 Description du fichier XML

Pour lancer les différents graphes de composants qui forment nos FTM, nous avons décidé d'utiliser la commande `roslaunch`. Bien qu'il soit possible de coder

un script pour lancer nos FTM en utilisant plusieurs commandes `rosrun` (une par nœud), nous avons préféré, pour des questions de lisibilité et de facilité de modification, utiliser la seconde option avec la description de l'architecture complète dans un fichier. La figure 2.3 décrit l'architecture d'un PBR ainsi que les deux serveurs contenant l'application.

```
1  <launch>
2    <!--Lancement du Serveur Primaire-->
3    <node pkg="server" name="server_P" type="server_node" output="screen">
4      <remap from="/prd2srv" to="/prd2srv_P"/>
5      <remap from="/getstate" to="/getstate_P"/>
6      <remap from="/setstate" to="/setstate_P"/>
7    </node>
8
9    <!--Lancement du Serveur Secondaire-->
10   <node pkg="server" name="server_S" type="server_node" output="screen">
11     <remap from="/prd2srv" to="/prd2srv_S"/>
12     <remap from="/getstate" to="/getstate_S"/>
13     <remap from="/setstate" to="/setstate_S"/>
14   </node>
15
16   <!--Lancement d'un FTM rattaché au Primaire-->
17   <group ns="Primaire">
18     <node pkg="communication" name="protocol" type="protocol" args="1" />
19     <node pkg="pbr" name="before" type="before_pbr" output="screen"/>
20     <node pkg="pbr" name="proceed" type="proceed_pbr" args="1" />
21     <node pkg="pbr" name="after" type="after_pbr" args="1" />
22     <node pkg="crash_detector" name="fd_prim" type="master_fd.py"/>
23   </group>
24
25   <!--Lancement d'un FTM rattaché au Secondaire-->
26   <group ns="Secondaire">
27     <node pkg="communication" name="protocol" type="protocol" args="2" />
28     <node pkg="pbr" name="before" type="before_pbr"/>
29     <node pkg="pbr" name="proceed" type="proceed_pbr" args="2"/>
30     <node pkg="pbr" name="after" type="after_pbr" args="2" />
31     <node pkg="crash_detector" name="fd_bu" type="slave_fd.py"/>
32   </group>
33
34 </launch>
```

Listing 2.3 – Fichier de description du PBR avec le serveur et sa réplique

Ce fichier *launch* permet de définir l'architecture contenant douze nœuds ROS, les deux serveurs (ligne 3 à 7 & 9 à 13), cinq nœuds pour le Primaire (ligne 16 à 20) et cinq nœuds pour le Secondaire (ligne 23 à 28). Nous avons regrouper les nœuds du Primaire sous le *namespace* Primaire (ligne 15) et fait de même avec le Secondaire (ligne 23).

Pour décrire un nœud que l'on souhaite lancer, il faut utiliser la balise xml `<node>`. Continuons avec l'exemple de l'*After* du PBR. Il faut ensuite définir le *package* auquel le nœud appartient (pkg="pbr"), le nom du nœud tel qu'il sera référencé auprès du *ROS Master* (name="after") qui remplacera le nom définit dans `ros::init(argc, argv, "AfterPBR")` et le nom de l'exécutable définit dans le *CMakeLists.txt* tel qu'illustré dans le listing 2.2 de la section 2.2.1. Entre autre, on peut définir les arguments du nœud (arg="1" pour le Primaire et arg="2" pour le Secondaire). Et finalement, on peut modifier le nom des communications à l'initialisation avec la balise `<remap>`. Par exemple le nom du *Service /getstate* du Serveur Primaire est renommé */getstate_P*.

Pour comparer avec la commande `rosrun`, voici ce que donnerait le lancement du même nœud comme représenté sur le listing 2.4

```
1 rosrun pbr after_pbr __name:=Primaire/after __role:=1
```

Listing 2.4 – Lancement du nœud After du PBR

Finalement `roslaunch` est équivalent à `rosrun` avec un interpréteur xml en plus. L'utilisation de l'interpréteur et de la lecture du fichier augmentent le temps de lancement de l'application face à un script shell contenant douze commandes `rosrun`.

2.3.2 Avantages de roslaunch

Plusieurs avantages se dégage de la commande `roslaunch` pour lancer un application composés de plusieurs nœuds ROS.

Premièrement, la définition des *namespaces* : Ici, il suffit d'utiliser la balise `<group ns="Primaire">` alors que pour une commande de type `rosrun`, il faut changer le nom et lui ajouter manuellement ce *namespace*, c.-à-d., le nœud *After* appartient au *namespace* Primaire si et seulement si on remanie le nom de cette manière : `rosrun pbr after_pbr __name:=Primaire/after`

Deuxièmement, le lancement d'une application : Comparé à un script shell, le fichier de description XML est plus lisible et l'utilisation d'options et de balises permet une vision globale et claire des interactions entre les nœuds.

Troisièmement, les types de nœuds : Nous pouvons définir dans le fichier de description différents types de nœud. Ceux-ci peuvent être codés en C++ ou en Python tant que l'option type le définit comme tel (ligne 22 & 31). Il n'y a donc pas de différences, que ça soit un exécutable compilé à partir du C++ ou un script python.

Enfin, le *remapping* des communications lors de l'initialisation des nœuds : Comme nous pouvons le constater avec le lancement des Serveurs, il suffit d'insérer simplement une balise <remap from="nom_initial" to="nom_final"/> à l'intérieur de deux balises <node></node> pour redéfinir un *Topic* ou un *Service*.

Maintenant que nous avons vu l'implémentation des composants et du graphe sous ROS, il faut définir la façon dont nous allons adapter les mécanismes, c'est-à-dire passer, par exemple, d'un PBR à un LFR. Nous allons considérer trois types d'adaptation, celle interne à un FTM, celle entre deux FTMs et enfin la composition.

3 Transition entre mécanismes

Nous allons maintenant laisser de côté l'implémentation des mécanismes pour nous concentrer sur l'*Adaptive Engine* et les méthodes utilisées lors de transitions pour modifier le graphe de composants et de communications. Nous considérons donc trois types de transitions : **(1)** la reconfiguration, **(2)** l'adaptation d'un FTM à un autre, et **(3)** la composition de deux FTM.

Dans le premier cas, il s'agit de la modification du graphe de communications et des paramètres des nœuds sans toucher au graphe de composants. Par exemple, lors du crash du Primaire dans un FTM duplexe, il faut relier la réplique au *Proxy* pour assurer le recouvrement. Cet exemple correspond appartient plus au fonctionnement nominal du FTM qu'à l'AFT mais la capacité à modifier les communications est importante pour les autres transitions.

Dans le deuxième cas, il s'agit de la substitution d'un ou plusieurs nœuds d'un FTM par ceux d'un autre. Cette transition a lieu lors d'un changement d'hypothèses lié au modèle de faute FT, aux caractéristiques applicatives AC ou aux besoins en ressources RS. Il faut donc pouvoir supprimer des nœuds et en lancer d'autres en ayant un impact minimal sur l'application.

Dans le troisième cas, il faut pouvoir substituer le *Proceed* par un ensemble complet P-BPA. Cette transition a lieu lorsqu'un nouveau modèle de fautes s'ajoute à celui déjà présent. Nous allons maintenant nous intéresser à leur implémentation adaptive dans ROS.

Ces transitions nécessitent des méthodes de modification du graphe de composants (déploiement et suppression de nœuds), des méthodes de liaison dynamique (aussi appelé *dynamic binding*) pour réorganiser un graphe de communications, mais également des méthodes de contrôle de l'exécution des graphes pour arrêter l'application dans un état stable (P.ex. le stockage des requêtes pendant une transition).

3.1 Reconfiguration d'un FTM

Comme dit précédemment, la reconfiguration implique la modification du graphe de communications et des paramètres des nœuds sans modifier le graphe de composants. L'exemple le plus probant est le rattachement du *Protocol* du Secondaire au *Proxy* lors du crash du Primaire. Or nous ne pouvons pas tuer et relancer le Protocol du Secondaire (perte de son état interne) donc nous ne pouvons pas utiliser le *remapping* à l'initialisation des noms des Topics fournit par ROS. Néanmoins, les *Topics* sur lequel le *Proxy* publie existent toujours après le crash du Primaire. Nous devons donc initialiser les ports de communication du Secondaire préalablement instanciés pour communiquer sur les Topics existants.

Rappelons que l'instanciation des ports de communication signifie créer des objets C++ *Publisher*/*Client* et/ou *Subscriber*/*Serveur* servant à communiquer de manière asynchrone/synchrone alors que l'initialisation des ports signifie faire appel aux méthodes `advertise`/`serviceClient` ou `subscribe`/`advertiseService` de ces

objets pour écrire/recevoir les messages des *Topics*/*Services* (cf section 2.2.1 de ce chapitre).

Nous avons ajouté aux nœuds pouvant recevoir une telle modification, une méthode pour contrôler l'initialisation des ports de communication. En effet, un *Topic* avec des types de données déjà connus par le nœud peut être créé lors de l'initialisation des ports de communication. Par exemple, nous avons implémenté un *Service* pour activer ou désactiver pendant l'exécution les *Topics* entre le Secondaire et le *Proxy* par lesquels transitent des données sous forme de tableaux de *char*. Lors de l'activation de ces *Topics*, le nom pourra être modifié mais les données resterons persistantes dans les topics, le format restant un tableau de *char*.

```
1  bool recoverSub(Request &req, Response &res){
2   //désactivation des ports de communication
3   if(req.activation==0){
4     publisher.shutdown();
5   }
6   //instantiation des ports pour reconnecter le nœud
7   else if(req.activation==1){
8     subscriber = n.subscribe(req.topic_sub, req.queue, callback);
9   }
10  return true
11 }
```

Listing 2.5 – Méthode générique d'édition de liens dynamiques

Ici, le code donné dans la figure Listing 2.5 nous montre une version simplifiée des méthodes implémentées pour l'édition de lien dynamique. On remarque dans cette fonction qu'un nœud externe devra utiliser un *Service* avec comme données de requête un *integer* pour activer ou désactiver les ports de communications, un strings *topic_pub* pour définir le nom des *Topics* et un *integer* pour définir la taille du *buffer* de message. Normalement cette méthode permet l'initialisation ou la désactivation des quatre types de ports de communication (*Publisher/Subscriber/Client/Server*) avec pour chacun de ces ports un nom de *Topic* ou de *Service*, un buffer pour les *Topics*. La méthode de *callback* doit être connue à l'avance pour savoir quelle méthode activer lors de l'initialisation d'un *Subscriber* ou d'un *Server*.

Toujours avec l'exemple du PBR de la figure 2.9 à la page 55, nous utilisons le nœud *Recovery* pour appeler ces méthodes dans le Secondaire et ainsi le relier au *Proxy* lors de la chute du Primaire. Le *Recovery* ne recevant plus de message d'un des *Crash Detector* (CD) sur le *Topic /alive*, va faire un appel de *Service* au niveau du *Protocol* (*Service* /rcy2pro) d'une des répliques pour la reconnecter au *Proxy* (*Topics* pxy2pro et pro2pxy) et ainsi assurer la reprise.

Il est évident que d'autres manières d'implémenter les mécanismes sont possibles et donc d'autres solutions pour résoudre se problème de reconnexion sont envisageables. Il serait envisageable, par exemple, d'utiliser des mémoires locales pour stocker périodiquement l'état du *Protocol* du Secondaire, de le supprimer et de le

relancer en utilisant les capacités de ROS. Nous pourrions utiliser le *remapping* à l'initialisation pour connecter le *Protocol* du Secondaire au *Proxy*. Nous allons maintenant nous intéresser à l'adaptation entre deux mécanismes pour pallier une modification des hypothèses issues des analyses de risques.

3.2 Adaptation entre deux FTM

La reconfiguration, dans notre exemple de PBR, fait partie intégrante du bon fonctionnement du FTM pour tolérer les fautes en crash. La notion de reconnexion doit être présente, qu'on utilise l'AFT ou non. Cependant, la reconfiguration utilise l'édition de lien dynamique qui fait partie des deux autres transitions (adaptation & composition). Nous allons maintenant nous intéresser à l'implémentation sous ROS de l'adaptation. Étant donné que la composition n'est qu'une variante de l'adaptation, intéressons nous à cette première au travers un exemple, le passage d'un LFR à un PBR. L'application subit une modification de version impliquant la perte de l'hypothèse de déterminisme. Le mécanisme LFR mis en place n'est plus valide et doit donc être remplacé par un PBR.

Bien que l'on veuille pallier les perturbations, à savoir des fautes non prévues dans les analyses de risques, tout ne peut pas être laissé au hasard lors d'une modification du graphe de composants. Il faut prévoir les transitions entre les mécanismes et être sûr que celui-ci est compatible avec l'application auquel nous le rattachons et bien entendu avec le modèle de faute considéré. Ici, une analyse préliminaire nous garantit que le *Before* et l'*After* doivent être changé.

L'adaptation définie dans la section 1.3 est la modification partielle du graphe de composant. Cela se traduit par le remplacement des nœuds ROS différents entre les deux FTM. Pour pouvoir remplacer ces nœuds et assurer la cohérence avant, pendant et après l'adaptation, et pour que le temps d'interruption de service soit le plus court possible, il faut que le système soit arrêté dans un état stable. Pour cela, les actions suivantes sont requises :

- Arrêter le FTM (et donc l'application) dans un état de quiescence ;
- Stocker les requêtes reçues pendant l'adaptation ;
- Supprimer les noeuds impactés par l'adaptation ;
- Lancer les noeuds de remplacement ;
- Assurer la cohérence des interfaces entre les noeuds
- Redémarrer l'application ;
- Dépiler les méssages stockés.

3.2.1 Contrôle de l'exécution de l'application

Concernant le contrôle de l'exécution de l'application, à savoir l'arrêt et redémarrage d'un composant ou un graphe de composant, ROS n'a pas d'interfaces de

```
void pause(){
  // Envoi le signal Unix SIGSTOP pour arrêter le Proxy
  system("pkill -SIGSTOP /Proxy");
}

void restart(){
  // Envoi le signal Unix SIGCONT pour relancer le Proxy
  system("pkill -SIGCONT /Proxy");
}
```

Listing 2.6 – Exemple des fonctions de gestion d'état de composants

programmation (aussi appelées API) dédiées. Nous devons donc faire appel aux signaux `SIGSTOP` et `SIGCONT` pour mettre les composants dans un état de repos stable puis pour les relancer comme l'illustre la figure 2.6.

Ainsi, l'*Adaptation Engine* contient ces deux méthodes pour contrôler l'exécution de l'application. En arrêtant le *Proxy*, on arrête l'envoi de requête du Client et on bloque son fonctionnement car même s'il envoie une requête, il attend sa réponse avant de pouvoir en générer une autre. Ainsi l'application et le FTM sont arrêtés en état de quiescence et nous pouvons effectuer l'adaptation.

3.2.2 Stockage des messages lors de l'arrêt du FTM

Une des capacités de ROS est de pouvoir stocker les requêtes à traiter dans un buffer défini lors de la création du *Topic* (lors de l'initialisation du port de communication). De ce fait, si le Client génère des requêtes pendant l'adaptation, celles-ci sont stockées en attente d'être traitées par l'application. Si nous souhaitons que seule la dernière requête soit traitée, nous n'avons qu'à définir un buffer de un message et toutes les requêtes précédentes sont écrasées au fur et à mesure.

3.2.3 Contrôle du cycle de vie d'un composant

ROS fournit de nombreuses fonctions pour manipuler les composants. Il permet d'ajouter et de supprimer des nœuds (`rosrun` ou `roslaunch` et `rosnode kill`), et de contrôler la liaison à l'initialisation d'un nœud (en utilisant la méthode de *remapping* de ROS présentée dans la section 2.3).

Néanmoins, la plupart de ses API sont des commandes en shell mais sont utilisables grâce à des appels "system" dans un programme C/C++. Nous avons donc dû utiliser ces commandes pour lancer et supprimer nos nouveaux nœuds, comme représenté sur le code donné dans les figures Listing 2.7.

La figure 2.7 représente les deux méthodes de l'*Adaptation Engine* pour manipuler le graphe de composants. Il peut supprimer n'importe quel composant avec la méthode `kill` qui prend comme paramètre le nom du ou des composants. Il peut également lancer n'importe quel groupe de nœuds à partir d'un fichier de description XML avec la méthode `launch` qui prend comme paramètre le nom de ce

```
void kill(std::string str_k){
  // Ex: Suppression de l'After du Primaire du PBR
  // kill("/Primaire/after") => rosnode kill /Primaire/after
  const char* cmde_k = ("rosnode kill " + str_k).c_str();
  system(cmde_k);
}

void launch(std::string str_l){
  // Ex: Adaptation LFR vers PBR
  // kill("pbr PBR_adapt.launch") => roslaunch pbr PBR_adapt.launch
  const char* cmde_l = ("roslaunch " + str_l).c_str();
  system(cmde_l);
}
```

Listing 2.7 – Exemple des fonctions d'ajout et de suppression de composant

fichier.

3.2.4 Cohérence des interfaces lors de l'adaptation

L'*Adaptation Engine* connaît les différentes actions pour la transition entre mécanismes. Celles-ci ont été validées hors-ligne pour assurer la cohérence des interfaces. De plus, l'uniformisation des communications internes au FTM nous permet de ne pas nous soucier des problèmes de compatibilité lors du changement d'un composant. Les entrées et sorties principales des composants *Before* et *After* sont les mêmes, peu importe le mécanisme.

```
<launch>
  <group ns="Primaire">
    <node pkg="pbr" name="after" type="after_pbr" args="1"/>
  </group>

  <group ns="Secondaire">
    <node pkg="pbr" name="after" type="after_pbr" args="2"/>
  </group>
</launch>
```

Listing 2.8 – Fichier launch pour l'adaptation entre un FTM et un PBR

Concrètement, les étapes de l'adaptation sous ROS pour passer d'un LFR à un PBR sont les suivantes :

1. Après validation par l'Opérateur, le Moniteur transmet à l'*Adaptation Engine* (AE) l'ordre d'adaptation du LFR en PBR ;
2. L'AE utilise la fonction *pause* pour suspendre l'application. Ici, l'arrêt du *Proxy* suffit à suspendre toutes les activités du FTM et du Serveur ;

3. L'AE supprime les *Before After* de chaque réplique du LFR avec la fonction *kill*. La variable str_k ici représente le nom des composants ;
4. L'AE lance à l'aide d'un fichier de description XML, illustré dans la figure Listing 2.8, les *Before After* du PBR avec la fonction *launch*. La variable str_l ici représente le nom du fichier ;
5. L'AE redémarre le *Proxy* et ainsi toute l'application avec la fonction *restart*.

Nous allons maintenant nous intéresser à la composition de deux mécanismes et aux spécificités de l'implémentation de cette transformation sous ROS

3.3 Composition de deux FTM

La composition correspond au remplacement d'un composant *Proceed* par un ensemble P-BPA. Nous allons détailler les étapes pour la composition de deux mécanismes. A ce titre, nous proposons de composer un mécanisme de réplication Duplex (PBR ou LFR) avec un mécanisme de redondance temporelle (TR).

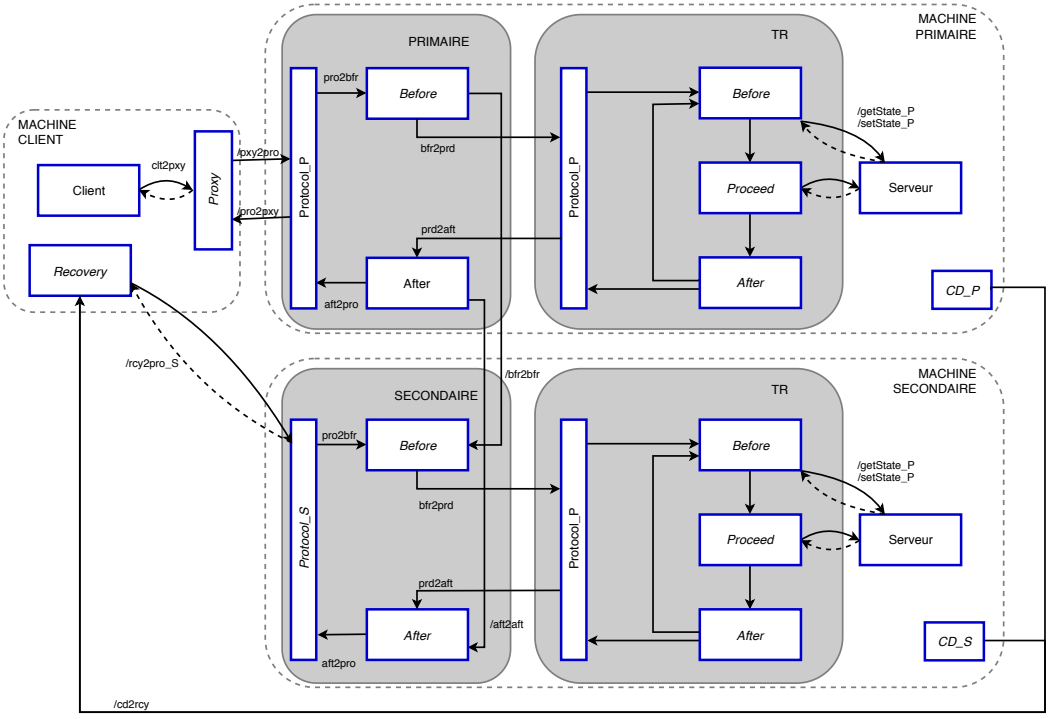


FIGURE 2.10 – Composition entre un LFR et un TR

Comme expliqué dans la section 1.3, nous pouvons composer des mécanismes ayant des hypothèses de modèle de fautes FT et des besoins en ressources RS différents tant que les hypothèses sur les caractéristiques de l'application AC sont compatibles et que la composition permet de tolérer le nouveau modèle de faute supplémentaire. Ici le TR a besoin d'une application déterministe et ayant accès à son état. Si nous composons un TR avec un PBR, il faut que le Serveur soit

déterministe en plus de donner accès à son état, si nous composons un TR avec un LFR, il faut que le Serveur donne accès à son état en plus d'être déterministe.

Le but maintenant va être pour composer un FTM duplex (ici un LFR ou un PBR) avec un TR. Nous allons pour cela substituer le *Proceed* du FTM par l'ensemble P-BPA du TR. Le résultat final est illustré sur la figure 2.10.

Cette figure, complexe au premier regard, permet d'illustrer le remplacement, pour le Primaire et le Secondaire, du nœud *Proceed* par un TR complet. On remarque également que le TR utilise les mêmes *Services* que le PBR en ce qui concerne la sauvegarde et la restauration de l'état du Serveur. De plus, il utilise le même *Protocol* que le Primaire dans les deux cas car ses ports de communications sont initialisés (à la différence du *Protocol* du Secondaire).

Les étapes nécessaires pour faire cette composition sont décrites ci-dessous :

- Le *Proxy* du Primaire est suspendu en utilisant le signal UNIX SIGSTOP ;
- Le nœud *Proceed* est tué par l'*Adaptation Engine* en utilisant la commande kill :

  `kill(Primary/Proceed)`
- Les nœuds TR (*P-BPA*) sont lancés (sur chaque réplique par l'*Adaptation Engine*) en utilisant le fichier de description XML et la commande launch :

  `launch("TR TR_composition.launch")` ;
- L'ensemble P-BPA du TR se connecte au LFR à la place de son *Proceed* en utilisant le *remapping* de ROS à l'initialisation ;
- Le *Proxy* du Primaire est redémarré en utilisant le signal UNIX SIGCONT.

C'est également l'uniformisation des communications qui permet le *remapping* des *Topics* pour connecter le TR au LFR. Il faut cependant tenir compte dans le fichier de description des *namespaces* pour le *remapping* des *Topics*. Ainsi `pxy2pro` et `pro2pxy` (communications entre normalement le *Proxy* et le *Protocol*) deviennent respectivement `Primaire/bfr2prd` et `Primaire/prd2aft` pour le Primaire et `Secondaire/bfr2prd` et `Secondaire/prd2aft` pour le Secondaire (anciennes communications du *Proceed* avec le *Before* et l'*After*).

Maintenant que nous avons examiné les différents moyens d'implémentation sous ROS, nous allons, dans la section suivante, faire une analyse critique des points forts et faibles non seulement de ROS mais également de la méthode elle-même.

4 Analyse critique de notre approche pour l'AFT

Il est temps maintenant d'analyser plus en détail les avantages et les inconvénients, d'une part de notre approche, d'autre part de l'implémentation sur ROS des mécanismes. Nous n'allons pas faire une critique de notre approche sur des critères de performance lors des transitions mais sur des critères de réutilisation de code et d'utilisation de COTS. En effet, bien qu'il soit intéressant de mesurer le temps de transition lors d'une adaptation, ce n'est pas le but de ces travaux. De plus, nous n'aurions que des résultats approximatifs car nous ne testons pas notre approche dans des conditions réalistes, au sens industriel du terme.

Regardons dans un premier temps la méthode que nous avons mise en place avec l'approche par composants substituables.

4.1 Une approche à gros grain suffisante ?

Cette approche par composants substituables a de nombreux avantages. Que cela soit dans sa facilité à créer des mécanismes, à définir les différences entre eux et de passer de l'un à l'autre. En effet, cette projection et ce regroupement en un ensemble P-BPA permet de concevoir simplement les mécanismes. Il suffit de lister les actions de Sûreté de Fonctionnement (SdF) qu'il faut exécuter avant et après une action fonctionnelle et de les regrouper dans des composants logiciels.

De plus, en ajoutant par rapport aux travaux précédents le composant *Protocol* qui sert de "râtelier" (rack logiciel) aux composants exécutant les actions de SdF, on facilite la composition de mécanismes. Nous nous sommes interrogés sur une autre manière d'implémenter ce nœud en utilisant des *Services* pour ordonnancer les composants BPA, depuis le *Protocol* comme le montre la figure 2.11. Nous avons choisi de substituer le *Proceed* par un ensemble P-BPA pour assurer une composition simple de plusieurs FTM. En effet, sur cette figure, nous ne pouvons pas substituer le *Proceed* par un ensemble P-BPA car les communications entre le *Proxy* et le *Protocol* et celles du *Proceed* ne sont plus les mêmes.

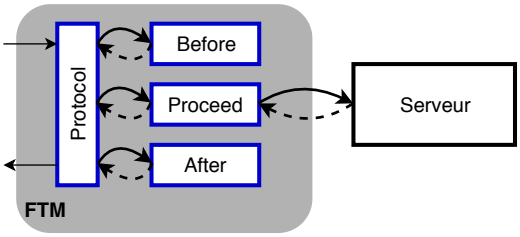


FIGURE 2.11 – Alternative de conception du *Protocol*

Ainsi dans ce schéma, nous perdons l'intérêt de pouvoir remplacer le *Proceed* par le *Protocol*. Il faudrait dans ce cas implémenter une communication spéciale pour composer un FTM avec un autre. C'est pour cela que cette solution n'a pas été retenue.

Le second gros avantage de la méthode que nous avons retenue est la robustesse du mécanisme. La perte d'un composant ne signifie pas la perte du FTM. Un composant peut avoir un état interne simple (p.ex. le *Before* du PBR ou du LFR ne font que de la synchronisation, il n'y a pas de variable à stocker) ou à état complexe (p.ex. Le *Before* du TR et les *After* du TR et du PBR comprennent soit l'état de l'application, soit les réponses à comparer). Si les composants sont à état interne simple, il est possible de relancer très facilement ce mécanisme depuis l'*Adaptation Engine* en ayant un impact faible sur la disponibilité du service fourni par l'application. Pour les composants à état interne complexe, il serait possible de stocker cet état régulièrement sur une mémoire locale et d'en faire une restauration à l'initialisation, ce qui augmenterait le temps d'indisponibilité mais assurerait une reprise correcte.

Toutefois, cette méthode n'est pas parfaite et, en fonction du support d'exécution, peut rapidement déboucher sur la violation des contraintes de temps réel. En effet, si nous prenons l'exemple du PBR, ce n'est pas moins de 17 composants qui sont nécessaires pour assurer le bon fonctionnement d'un Serveur et de sa réplique. Si le support d'exécution choisi est ROS, chaque composant est implémenté sur un nœud (processus Unix) et communication est implémenté avec un *Topic* ou un *Service* (communication TCP/IP). Ainsi l'ajout de ce FTM devient rapidement encombrant.

	Composants	*Topics*	*Services*
PBR	13 composants	11 Topics	3 Services
LFR	13 composants	12 Topics	1 Services
PBR+TR	19 composants	21 Topics	5 Services
LFR+TR	19 composants	22 Topics	5 services

TABLE 2.1 – Nombre de nœuds et de communications par FTM

De plus, certains composants appartenant à un FTM pourraient être réutilisés par d'autres FTM avec quelques modifications. Par exemple, l'*After* du PBR contient les mêmes méthodes que le *Before* du TR. La différence se situe au niveau de la transmission de l'état. Dans le PBR, l'état est transmis entre la capture et la mise à jour par un message de *checkpoint*. Dans le TR, l'état est transmis par une variable interne. Avec une légère modification du composant, on pourrait augmenter le niveau de réutilisation de code.

Un FTM complexe et décomposé peut donc être coûteux en ressource mais c'est le prix à payer pour obtenir ce niveau de robustesse et d'adaptabilité. Chaque FTM est décomposé en éléments indépendants, s'exécutant dans des espaces d'adressage différents ; il n'y a pas propagation d'une défaillance d'un composant (crash d'un processus) vers les autres appartenant au FTM.

Penchons nous maintenant sur le support d'exécution que nous avons choisi et analysons la pertinence de ROS pour implémenter l'AFT.

4.2 ROS, un support d'exécution idéal ?

En ce qui concerne les caractéristiques de ROS pour la mise en œuvre de l'AFT, nous estimons qu'elles ne sont pas entièrement satisfaisantes, car ROS nécessite une instrumentation des nœuds pour assurer les liaisons dynamiques, et l'API pour contrôler le cycle de vie des composants en temps réel est trop faible. En effet, ROS n'ayant pas d'API dédiée pour la manipulation dynamique de communications, nous avons rajouté, dans les nœuds pouvant changer de rôle lors d'une transition, une méthode permettant la manipulation des communications. De plus, le contrôle des nœuds est mis en œuvre grâce à l'exécutif sous-jacent (signaux UNIX). Cependant, nous avons montré que bien qu'imparfaite, la résilience informatique utilisant l'AFT peut être implémentée sous ROS. L'édition de lien dynamique est possible sur des *Topics* existants en ajoutant les méthodes d'activation et de désactivation dans les nœuds comme expliqué dans la section 3.1. ROS permet le confinement spatial et temporel de ses nœuds, puisque les composants peuvent être implémentés sur des processus UNIX qui ont leur propre espace d'adressage et ne sont pas synchronisés par une horloge commune. Le modèle de communication utilisé dans ROS est également un avantage pour concevoir et mettre en œuvre des applications distribuées résilientes.

Le gros point faible de ROS pour son utilisation dans un système fiable et résilient est le fait que le *ROS Master* est un point de défaillance unique dans l'architecture. Ce *ROS Master* doit être opérationnel lors de l'installation d'une application et lors de son exécution. Lorsque le *ROS Master* tombe, l'architecture logicielle entière doit être redémarrée. En collaboration avec l'équipe de Gene Cooperman, nous étudions actuellement une implémentation répliquée du *ROS Master* utilisant la bibliothèque DMTCP (*Distributed MultiThreaded Checkpointing*) développée à NorthEastern University Boston [Ansel 2009]. Ceci est cependant très complexe et il n'est actuellement pas possible d'avoir plusieurs *ROS Master* fonctionnant en parallèle pour une même application. Pour l'instant, notre architecture logicielle, comme toute application ROS, est liée à un *ROS Master* unique. À noter que la prochaine révision majeure de ROS (ROS2) est basée sur un système de communication DDS (Data Distribution Service) qui devrait aider à résoudre ce problème en distribuant les fonctionnalités principales du *ROS Master* entre les nœuds du système. Chaque nœud aurait une connaissance de ses voisins et donc une vision partielle du graphe de composant. Cette approche nécessiterait toutefois des protocoles de diffusion fiable (communication de groupe), correctement mis en œuvre et validés.

Un autre désavantage est l'utilisation des ressources par ROS. Nous avons mené une expérience sur un processeur i7-6600U dual-core cadencé à 2,6 GHz. Deux nœuds communiquent à travers un Topic en s'envoyant uniquement un entier toutes les 100ms. Il nous a suffi de lancer une centaine de nœuds (environs 100 Émetteurs et 100 Récepteurs) pour que des messages soient perdus ou non traités. De plus les nœuds ne respectaient plus la contrainte d'un envoi de message toutes les 100ms.

ROS n'est pas le support d'exécution parfait. La mise en œuvre de l'AFT est

possible, mais reste au niveau du prototypage pour une analyse de concepts à cause de ce point de défaillance unique. Malgré cela, ROS reste un support d'exécution attractif pour appréhender le concept d'architecture logicielle à composants et de l'*Adaptive Fault Tolerance*. Les supports d'exécution concurrents permettant ce type de mise en œuvre ne sont pas légions. Enfin, c'est une étape intéressante dans l'objectifs de mettre en œuvre l'AFT sur Adaptive AUTOSAR.

4.3 ROS, AFT, approche à gros grain, ce qu'il faut en retenir

Les exigences générales pour un support exécutif adapté à la mise en œuvre de l'AFT que nous avons exposées dans des travaux antérieurs reposent sur les caractéristiques suivantes : **(1)** la suppression et le lancement dynamique des composants, **(2)** la suspension et l'activation des composants, **(3)** la manipulation des communications entre les composants. En plus de la *Separation of Concern*, ces caractéristiques sont liées au degré d'observabilité et de contrôle que la plate-forme logicielle fournit sur l'architecture à composants des applications (y compris ceux de nos mécanismes) au moment de leur exécution. De plus, pour assurer la cohérence avant, pendant et après la reconfiguration de l'architecture logicielle à composants, plusieurs exigences doivent être soigneusement examinées : les composants **(1)** doivent être arrêtés à l'état de repos, c'est-à-dire que lorsque tout le traitement interne est terminé, **(2)** les demandes reçues sur le composant arrêté doivent être mises en mémoire tampon.

Nous nous sommes basés sur les travaux précédents et avons conçu une architecture permettant d'implémenter facilement des FTM en respectant le *Design for Adaptation* (D4A) et permettant ainsi de les adapter et de les composer rapidement et simplement sur ROS afin de les rendre opérationnels.

Dans nos expériences, un composant a été implémenté sur un nœud ROS au moment de l'exécution, fournissant ainsi une ségrégation de l'espace mémoire. La liaison entre les composants repose sur des *Topics* enregistrés auprès du *ROS Master*. La manipulation de liaison dynamique est possible mais nécessite une instrumentation des nœuds, ROS ne fournit pas d'API spécifique pour gérer une telle connexion entre composants. Comme nous l'avons vu dans la section précédente 3.1, du code supplémentaire est nécessaire pour gérer cette manipulation dynamique en utilisant les fonctionnalités fournies par le système d'exploitation Linux sous-jacent et les outils de définitions des communications sous ROS.

En ce qui concerne les notions de **contrôle du cycle de vie des composants** nous les avons implémentés car **(1)** ROS fournit des commandes pour supprimer et créer des nœuds, **(2)** les nœuds peuvent être arrêtés et redémarrés grâce aux signaux UNIX, et **(3)** ROS permet aux nœuds de se connecter ou de se déconnecter à des *Topics* ou des *Services* pendant l'exécution avec l'ajout d'une méthode spécifique qui doit être ajouté aux nœuds concernés. Pour les **exigences de cohérence lors d'une transition**, **(1)** l'arrêt du *Protocol* par un signal SIGSTOP suffit à arrêter l'échange de messages et donc l'exécution de l'application, et **(2)** les *Topics* stockent les messages sortants lorsque les *Subscribers* ne sont pas disponibles.

5 Synthèse

Ce deuxième chapitre nous a permis d'expliquer la conception de mécanismes de tolérance aux fautes en respectant le D4A. Il nous a permis également d'illustrer les techniques mises en place pour permettre la transition d'un FTM à un autre ou la composition de deux FTM et ainsi assurer la sûreté de fonctionnement du système face à des perturbations.

Ce qu'il faut retenir finalement de ce chapitre est la conception des différents mécanismes, les composants nécessaires à la mise en œuvre de l'AFT ainsi que leur implémentation sous ROS. De plus, nous avons pu remarquer que nos mécanismes, en dépit d'une complexification du graphe de composants initial (un Client et un Serveur) et des communications, sont robustes et facilement implémentables. ROS, quant à lui, malgré son point de défaillance unique qui le disqualifie immédiatement comme solution industrielle, reste un bon support d'exécution pour appréhender l'AFT. Nous suivrons avec attention les travaux de duplication du *ROS Master* ainsi que l'arrivée de ROS2

Nous étudierons donc dans le prochain chapitre l'affinage de la méthode pour résoudre le premier problème concernant la généricité des composants. En effet, la conception des FTM ne prend en compte que l'ordre d'exécution des actions de Sûreté de Fonctionnement spécifique à un mécanisme. Une action de SdF doit être exécutée avant le Serveur ? Elle sera implémentée dans le *Before*. Or si nous prenons le cas du *Before* du TR et des *Afters* du PBR, ils contiennent les mêmes méthodes. Nous aurions donc pu réutiliser le code de sauvegarde et de restauration d'état du TR pour le PBR et vice versa.

Nous nous pencherons donc non plus sur le graphe de composants P-BPA mais sur les actions élémentaires de SdF nécessaires à la création d'un FTM. Nous allons, dans le Chapitre 3, lister les actions de SdF, regrouper les actions communes en catégorie et analyser au sein des catégories les parties communes (génériques) et celles amenant à une sous-division de la catégorie (spécifiques) en fonction de la conception du FTM et de l'application. Ce chapitre se divisera en trois sections majeures. La première se concentrera sur le niveau de granularité à adopter pour la conception et l'implémentation de nos FTM. La deuxième se focalisera sur l'analyse de différents mécanismes, que ça soit des mécanismes utilisés dans l'automobile, dans l'aéronautique ou tout autre système embarqué. La troisième sera consacrée à l'implémentation de ces fonctions sur ROS en projetant ces fonctions sur des nœuds et en gardant, de l'approche par composants substituables, les techniques d'adaptation et de composition.

Chapitre 3

Du composant à l'action

Sommaire

Introduction

Nous avons décrit dans le Chapitre 2 une méthode pour concevoir, implémenter, mettre à jour, adapter et composer des mécanismes de tolérance aux fautes sur ROS. Pour cela, chaque mécanisme est scindé en composants avec les principales actions de sûreté de fonctionnement regroupées dans les composants *Before* et *After*. Ensuite l'*Adaptation Engine*, chargé de manipuler le graphe de composants, modifie celui-ci en substituant les composants caduques du FTM en place par ceux du nouvel FTM à attacher à l'application. C'est une mise à jour différentielle ne modifiant que les composants qu'il est nécessaire de changer.

Malgré sa généricité et sa facilité à être mise en place, nous avons pu identifier deux problèmes majeurs à cette approche par composants substituables, un problème de réutilisation des composants et un problème de ressources utilisées. Dans ce chapitre, nous allons nous intéresser au premier problème et donc modifier l'approche précédente en ajustant le niveau de granularité. Il nous faut pour cela changer de point de vue et ne plus nous focaliser sur les différents mécanismes mais sur les actions de SdF qui les constituent. Actuellement, si nous souhaitons modifier une action de SdF pour la substituer par une action équivalente (P.ex. remplacer une comparaison par un vote), il faut remplacer un composant complet. Nous devons donc affiner la granularité de nos composants pour en extraire les actions de SdF. Ensuite, nous décomposerons un ensemble représentatif de FTM avec cette nouvelle granularité, analyserons leurs caractéristiques et extrairons les éléments en communs permettant leur création.

Dans ce chapitre, nous n'allons donc plus considérer les FTM comme un ensemble P-BPA mais comme un ensemble d'actions de SdF. Les actions partageant des points communs peuvent être regroupées en catégories d'actions.

Nous allons donc modifier le niveau de granularité de notre méthode et nous concentrer sur ces actions constituant nos FTM. Pour cela, nous identifions différentes étapes pour caractériser ces actions : **(1)** modélisation des FTM avec un niveau de granularité plus fin, **(2)** analyse d'un ensemble représentatif de FTM pour en déduire des catégorie d'action de SdF, **(3)** application de cette nouvelle granularité à l'implémentation précédente sur ROS.

La première section de ce chapitre se concentrera sur le niveau de granularité à adopter pour la conception et l'implémentation de FTM. Nous commencerons par modéliser nos mécanismes projetés sur l'approche P-BPA avec des graphes de type place/transition puis nous éclaterons ces composants.

La deuxième section se focalisera sur l'analyse de différents mécanismes, que ça soit des mécanismes utilisés dans l'automobile, dans l'aéronautique ou tout autre système embarqué. De cette analyse nous en tirerons des catégories d'actions de SdF génériques.

La troisième section sera consacrée à l'implémentation de ces actions sur ROS en les projetant sur des nœuds et en gardant de l'approche précédente les techniques d'adaptation et de composition.

1 Décomposition fine des mécanismes

Le principal avantage de l'approche par composants substituables est sa facilité à être mise en œuvre, l'adaptation différentielle des mécanismes avec le minimum de composants à changer et la composition des mécanismes (par substitution du *Proceed* par un ensemble P-BPA) pour répondre à un modèle de fautes complexe. Le FTM est également robuste car même si la perte d'un composant empêche le bon fonctionnement de l'application, le composant peut être relancé sans impliquer la perte complète du FTM.

Toutefois, avant d'être un graphe de composants, un FTM est avant tout un ensemble d'actions de SdF pour détecter et tolérer une erreur ou une défaillance de l'application. Dans l'approche par composants substituables, nous projetons pour chaque FTM l'ensemble de ces actions sur les deux composants *Before* et *After*. Chaque composant regroupe donc plusieurs actions. Or des actions peuvent être communes à plusieurs FTM et donc réutilisées sans avoir à tout développer de nouveau.

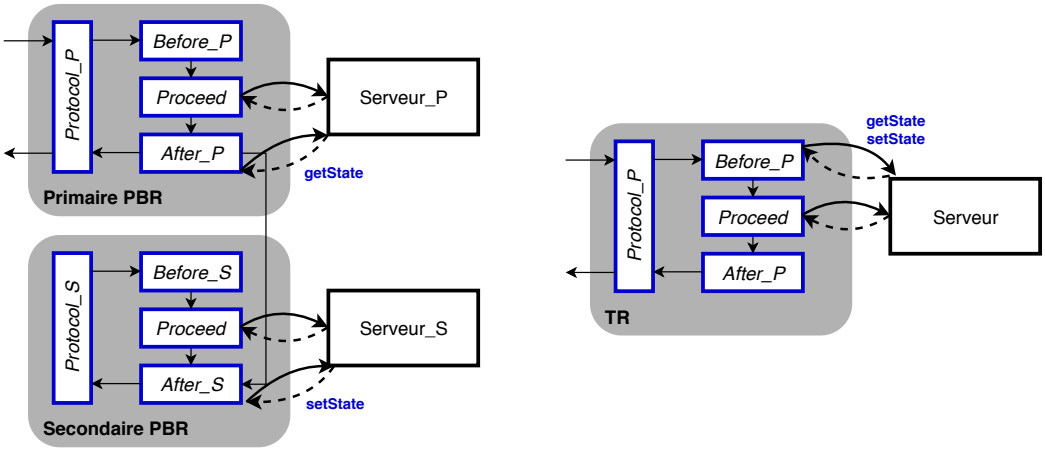


FIGURE 3.1 – Actions communes entre le PBR et le TR

Par exemple, la figure 3.1 expose les actions de capture et de mise à jour de l'état qui sont communes dans les *After* du PBR et dans le *Before* du TR. La différence majeure au niveau conceptuel est la transmission de l'état entre l'action de capture et l'action de mise à jour. Dans le cas d'un PBR, la transmission se fait par un message de *checkpoint* alors que dans le TR, l'état est stocké sur une variable locale partagée. À cette étape, le réseau de Petri permet de modéliser nos FTM avec un modèle mathématique formel et non plus une charte graphique .

Nous allons dans cette section proposer une conception à grain fin des composants pour exhiber les différentes actions qu'ils remplissent et les classer comme briques élémentaires d'un FTM. Nous utiliserons deux FTM, le PBR et le TR pour illustrer notre méthode de classification des actions qui sera réutilisée pour un ensemble représentatif de FTM. Cette méthode sera ensuite utilisée dans la section 2 de ce chapitre pour étendre la classification au travers de l'analyse.

1.1 Modélisation en réseau de Petri

Dans l'approche précédente, la définition des mécanismes s'exprime de la manière suivante : lors de la conception de l'application, on effectue une analyse de risques et en fonction de la sévérité des conséquences d'une défaillance et on définit un ensemble de mécanismes à rattacher à l'application pour la rendre sûre de fonctionnement. Une fois ces mécanismes définis, ils sont projetés sur un ensemble *Before*, *Proceed* et *After*. On ajoute à cela les composants *Proxy* et *Protocol* parallèlement aux Services Communs de TaF. Les transitions entre les mécanismes recommandés sont étudiées à l'avance pour assurer une adaptation ou une composition avec un impact minimal sur la disponibilité et l'intégrité du système. En effet, il ne faut pas que la transition lors de l'adaptation d'un FTM vers un autre ou lors d'une composition ait des conséquences néfastes sur l'exécution de l'application critique à laquelle le FTM est rattaché.

Dans le chapitre précédent, nous avions modélisé nos mécanismes avec des graphes de composants. L'avantage de ce graphe est qu'il représente parfaitement l'ensemble des composants formant l'application et les FTM ainsi que leur communications. Cependant ce type de graphe n'a pas de langage descriptif standard à l'inverse des diagrammes issus de l'UML ou encore des GRAFCET. Nous allons utiliser des graphes de type place/transition, ici des réseaux de Petri, pour modéliser les interactions entre les composants. Nous avons choisi les Réseaux de Petri pour leur lisibilité, leur facilité de mise en œuvre et leur expressivité qui permet de modéliser facilement l'ensemble BPA de n'importe quel FTM.

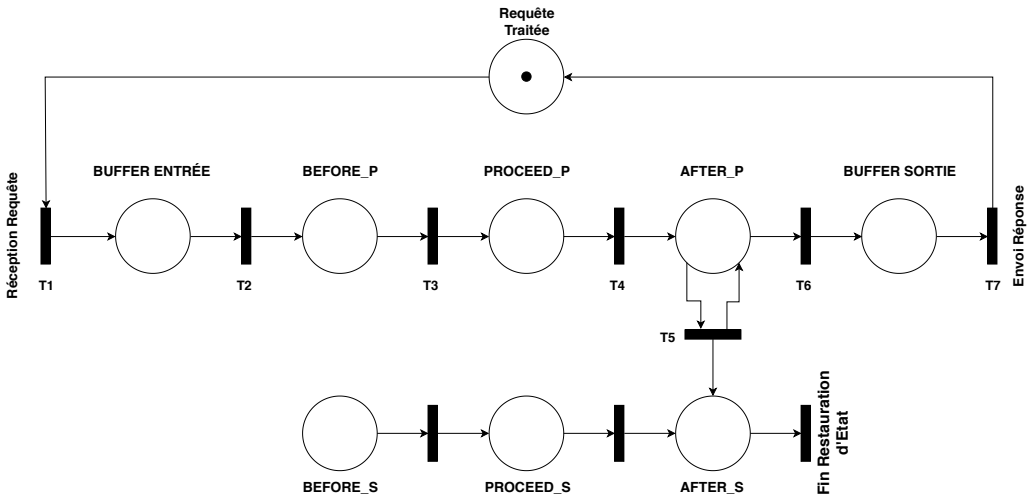


FIGURE 3.2 – Modélisation d'un PBR en réseau de Petri

La figure 3.2 représente un PBR décomposé en BPA et modélisé en réseau de Petri. Ici, le *Protocol* est modélisé par les *buffers* d'entrée et de sortie. Sur envoi d'une requête par le Client au travers du *Proxy*, celle-ci est réceptionnée par le *Protocol* (T1) qui la transmet au *Before* du Primaire (T2). La requête ensuite est transférée au *Proceed* (T3) qui se charge de la transmettre à l'application. Sur

réception de la réponse, cette dernière est transférée à l'*After* du Primaire (T4). L'*After* capture l'état de l'application, crée le message de checkpoint qu'il envoie à l'*After* du Secondaire (transition T5 entre *After_P* et *After_S*. Enfin, la réponse est transférée au *Protocol* (transition T6 avec le *buffer* de sortie) avant d'être renvoyée au Client (T7) par l'intermédiaire du *Proxy*.

Ce réseau de Petri illustre les branches du Primaire et du Secondaire avec les interactions entre celles-ci lors d'un fonctionnement nominal. Pour affiner la granularité des composants, nous ne nous intéressons pas aux interactions entre ceux-ci lors d'une reconfiguration, d'une adaptation ou d'une composition.

Le but de notre analyse est d'extraire des composants les différentes actions de SdF pour les catégoriser. Or, le niveau de granularité de cette modélisation n'est pas satisfaisant car il ne permet pas d'exprimer les différentes actions de SdF contenues dans un composant. Par exemple, nous avons décrit plusieurs actions dans l'*After_P* qui ne sont pas modélisées sur la figure. On ne voit ni la capture de l'état, ni la création et l'envoi du message de *checkpoint* au Secondaire, ni l'envoi de la réponse au *Protocol*. Ce problème est également présent pour la modélisation du TR.

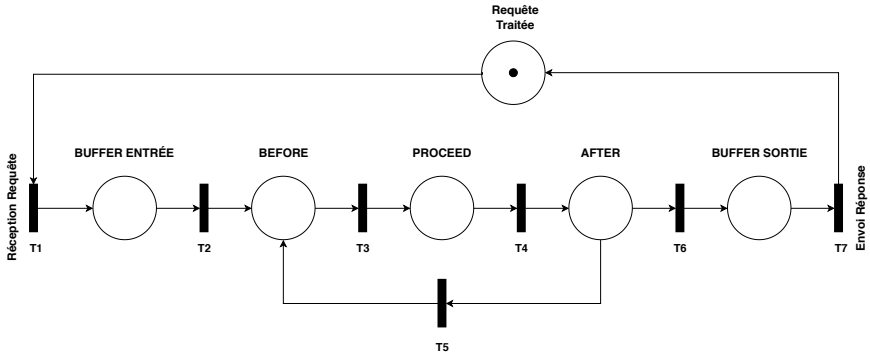


FIGURE 3.3 – Modélisation d'un TR en réseau de Petri

La figure 3.3 représente ici un TR décomposé en BPA modélisé également en réseau de Petri. Comme pour le PBR, le *Protocol* reçoit la requête (T1) qu'il transfert au *Before* (T2). Celui-ci capture l'état de l'application puis transmet la requête au *Proceed* (T3). Une fois la requête envoyée à l'application et la réponse transmise au *Proceed*, ce dernier la transfert à l'*After* (T4). L'*After* stocke la réponse puis envoie au *Before* une demande de mise à jour de l'état (T5). Le *Before* restaure l'état initial de l'application puis transmet au *Proceed* la requête. Celui transfert après une nouvelle exécution de l'application la réponse de l'application à l'*After*. L'*After* stocke la réponse puis compare les deux réponses. Si les deux sont identiques, la réponse est transférée au *Protocol* (transition T6 avec le *buffer* de sortie) avant d'être renvoyée au Client (T7) par l'intermédiaire du *Proxy*.

Nous remarquons que les actions multiples contenues dans le *Before* et l'*After*, à savoir la capture et la mise à jour de l'état pour le premier et le stockage et la comparaison pour le second, ne sont pas représentées. Nous avons remarqué, dans le chapitre précédent, une similitude des actions entre le *Before* du TR et les *After*

du PBR. En effet, dans les deux cas il faut des composants de gestion d'état pour sauvegarder et restaurer l'état de l'application. Enfin, la granularité des composants ne permet pas la représentation des services de TaF. La nécessité du stockage des réponses sur une mémoire locale avant leur comparaison pour garantir l'intégrité des données n'est pas présente. Nous en déduisons que la granularité de l'approche BPA n'est pas efficace concernant la conception des FTM car elle fait abstraction des similarités entre ceux-ci et ne permet pas une différentiation des actions internes des FTM.

L'avantage de l'utilisation des réseaux de Petri est la possibilité de modéliser l'ensemble BPA de n'importe quel FTM facilement et d'en extraire des fichiers de description du système. Ainsi, nous pourrions automatiser le lancement, l'adaptation et la composition avec un interpréteur de langage déclaratif, par exemple, un fichier de description XML comme utilisé dans ROS avec la commande `roslaunch`. La thèse ne s'est cependant pas concentrée sur l'utilisation d'un interpréteur pour lancer les FTM automatiquement, préférant se focaliser sur l'utilisation de ce langage déclaratif pour représenter ses FTM.

Cependant, même si notre modélisation sur les réseaux de Petri illustre simplement les interactions entre les composants, le niveau de granularité de ces derniers ne permet pas une description complète des actions de SdF. Le fonctionnement interne du FTM, que ça soit la différentiation des actions de SdF ou l'utilisation des Services Communs de TaF n'est pas représenté. Le graphe est macroscopique et demande un affinage pour laisser apparaître les briques élémentaires qui nous intéressent.

Nous allons maintenant changer notre façon de modéliser nos mécanismes en éclatant les composants pour laisser apparaître les actions de SdF. Nous pourrons commencer une classification des actions en fonction des similarités de conception et d'implémentation et ainsi mettre en avant les capacités de réutilisation de code.

1.2 Affinage de la granularité

Prenons pour exemple le TR : la modélisation par composants ne différencie pas les actions regroupées dans le *Before* ainsi que l'utilisation de certains Services Communs de TaF. L'*After* du TR est défini comme une comparaison de réponses et l'utilisation de mémoire locale pour stocker les réponses possibles avant la comparaison de celles-ci.

En comparaison avec la figure 3.3, la figure 3.4 nous offre de nombreuses précisions sur les exigences concernant le fonctionnement interne du TR et les Services Communs de TaF invoqués lors de son exécution. Ainsi, la figure modélise la séparation du *Before* en deux actions exécutées en séquence (P2 & P4) avec une sauvegarde sur mémoire locale (P3). Cela nous permet de mieux appréhender le concept de capture et de mise à jour de l'état avant l'exécution. De plus, le réseau de Petri expose la nécessité d'une mémoire locale (P8) et d'une action de stockage (P6) pour comparer deux réponses (P7) en étant sûr de l'intégrité de celles-ci.

Nous estimons que ce niveau de granularité est plus satisfaisant que la modélisa-

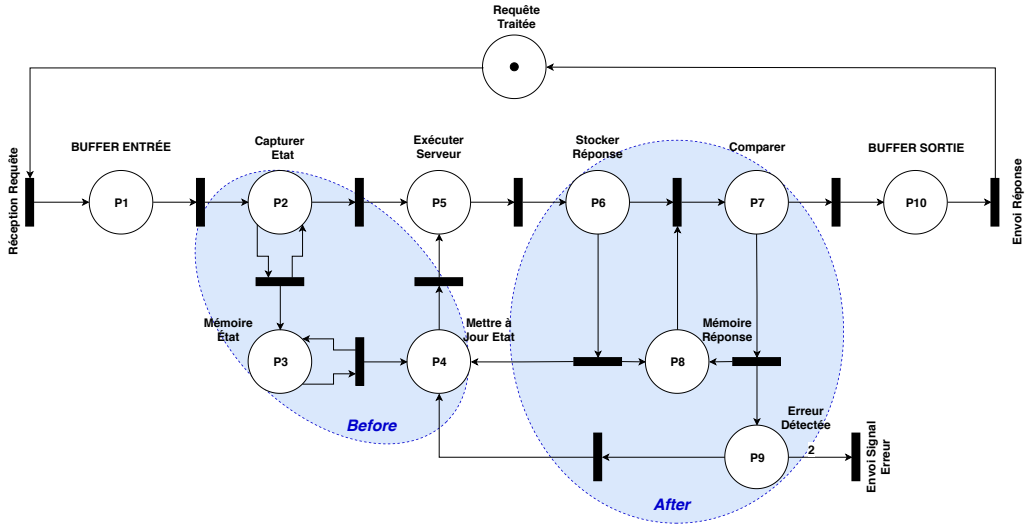


FIGURE 3.4 – Approfondissement du réseau de Petri d'un TR

tion BPA pour mettre en évidence les actions constituant les différents mécanismes. Nous pouvons ainsi commencer la classification des actions de SdF. L'adaptation et la composition ici sont plus délicates à appréhender étant donné l'éclatement des composants. On ne peut plus définir ici l'adaptation comme la substitution de composants contenant plusieurs actions mais comme la substitution fine d'actions élémentaires. Il y a plus d'éléments et d'interactions à prendre en compte.

Cette approche à grain fin amène la modification différentielle à un autre niveau. Sur la figure 3.4, si nous souhaitons changer la façon dont nous comparons les données pour détecter une défaillance en valeur transitoire, nous pouvons dans l'assemblage d'actions juste changer l'action de comparaison (P7) et ne pas changer un composant *After* complet (P6, P7, P8 et P9). Nous commençons ainsi à appréhender quelques catégories d'action de SdF comme la catégorie de Gestion d'État (Capture et mise à jour) ou la catégorie de Sélection de données (Comparaison, vote...).

Ces catégories ne peuvent pas être utilisées telles quelles. Nous ne pouvons pas simplement dire que le TR et le PBR ont des actions issues de la catégorie Gestion d'État. Il y a des différences de conception au niveau de la technique employée pour la gestion de l'état. Nous pouvons utiliser une technique de type *Getter/-Setter* demandant une instrumentation de l'application pour avoir accès à son état [Kinder 2012], ou une technique comme DMTCP [Ansel 2009] qui fait une copie mémoire du processus. Un autre choix se situe au niveau de la transmission de l'état entre la capture et la mise à jour (Transmission par message, accès à une variable locale). De plus, il faut pouvoir préciser ensuite les différences d'implémentation comme l'interface logicielle, le protocole de communication, ou encore le format de message. Ainsi chaque catégorie générique contenant les éléments communs sera spécifiée en sous-catégories en fonction des choix de conception et d'implémentation.

Prenons l'exemple de la Gestion d'État comme catégorie générique. Cette caté-

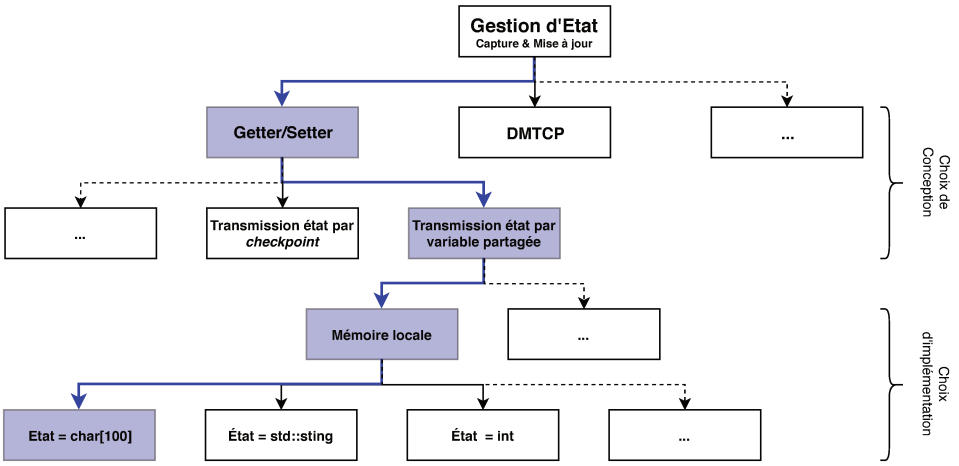


FIGURE 3.5 – Catégorie Gestion d'État et ses sous-catégories

gorie contient au moins deux actions de capture et de mise à jour de l'état. Elle est divisée, pour l'instant, en deux sous catégorie en fonction de la technique employée. La figure 3.5 représente la catégorie Gestion d'État et ses sous-catégories possibles. Le chemin représenté en bleu illustre les choix qui ont été faits pour implémenter les actions de Gestion d'État pour un TR avec une technique de type *Getter/Setter*, une transmission d'état par variable partagée avec un stockage sur mémoire locale et un état d'application du format d'un tableau de cent *char*.

Cette section ne se concentre que sur la classification des actions en catégories et sous-catégories pour définir un ensemble permettant leur réutilisation au sein d'un grand nombre de FTM. Certaines catégories peuvent contenir plusieurs actions. Par exemple, il n'est pas logique d'avoir une action de capture d'état sans une action de mise à jour de l'état. Nous allons maintenant, à partir de la décomposition du TR et du PBR, définir une méthode d'extraction et de classification des actions de SdF en fonction de la conception du FTM. Cela nous permettra de reproduire cette méthode sur un ensemble représentatif de FTM pour définir une liste non-exhaustive de ces actions et des catégories les regroupant.

1.3 Méthode de classification : Un cas concret

Prenons un schéma de conception d'un FTM. Listons les actions de SdF qui composent ce mécanisme et classifions ces actions en fonction de catégories génériques. Une fois ces catégories définies, étudions différentes conceptions et implémentations de ses mécanismes au travers d'articles scientifiques. Enfin, spécifions les catégories génériques en sous-catégories en fonction des choix de conception et d'implémentation.

Les choix d'implémentation étant très nombreux et très spécifique au support d'exécution (formats de message ou les protocoles de communications), nous ne les prendrons pas en compte dans un premier temps. Les sous-catégories seront donc

issus de choix de conception.

Finissons en premier l'analyse du TR. Nous avons extrait du *Before* la catégorie Gestion d'État qui contient les actions de capture et de mise à jour de l'état. Elle est constituée d'au moins deux niveau de sous-catégories. Le premier niveau est dû à un choix de conception concernant la technique utilisée pour capturer l'état et le mettre à jour. Nous avons vu dans le Chapitre 2 une technique *getter/setter* définie un peu plus tôt mais il existe de nombreuses techniques de *checkpointing* comme le *Diskless checkpointing* [Plank 1998], DMTCP ou encore la bibliothèque *libckpt* [Plank 1994]. Le second niveau est dû à la façon dont l'état est transmis comme le rappelle la figure 3.5. De plus, le *Before* contient un service de TaF pour sauvegarder l'état sur une mémoire locale.

Le *Proceed*, quant à lui, a une action d'envoi de la requête et de réception de la réponse. C'est une communication synchrone dans le cas d'un Client/Serveur, et deux communications asynchrones dans le cas d'une chaîne fonctionnelle. C'est finalement lui qui synchronise l'exécution de l'application. Nous avons donc ici une catégorie Synchronisation illustrée par la figure 3.6 qui apparaît. De plus, nous avons défini dans le Chapitre 2 le besoin d'avoir des communications uniformes (format unique des messages) à l'intérieur du FTM. Le *Proceed* contient une action d'encapsulation et de désencapsulation des données. Il faut une action pour désencapsuler les données avant d'envoyer la requête à l'application, et une action pour encapsuler la réponse avant de la transmettre à l'*After*. Ces deux actions (encapsulation & désencapsulation) sont regroupées dans le traitement de données.

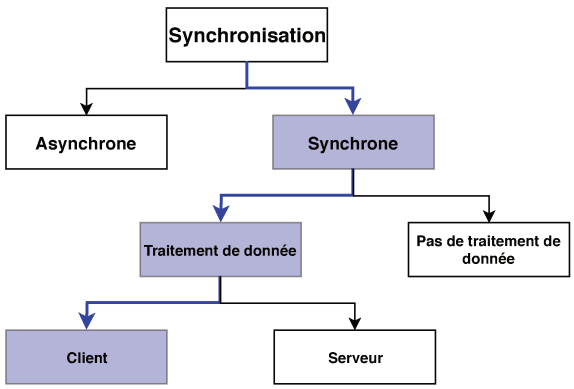


FIGURE 3.6 – Catégorie Synchronisation issue du PBR

Enfin l'*After* du TR contient une action de comparaison des données pour détecter l'apparition d'une faute transitoire en valeur. Cette action permet de sélectionner la réponse à renvoyer. Nous avons donc une catégorie Sélection de Données illustrée par la figure 3.7 qui contient une action générique qui détermine la donnée correcte. Plusieurs techniques existent que cela soit de la sélection par vote [Delgado 1999], par moyenne [Goel 1979] ou encore par assertion [Mera 2009]. La première sous-catégorie que l'on peut extraire est un choix de conception lié à la technique employée. Chacune de ces techniques peut avoir une sous-catégorie s'il

existe plusieurs façons de les concevoir (vote par majorité absolue, par majorité relative, par pondération). De plus, l'*After* contient un service de TaF pour sauvegarder les différentes réponses sur une mémoire locale.

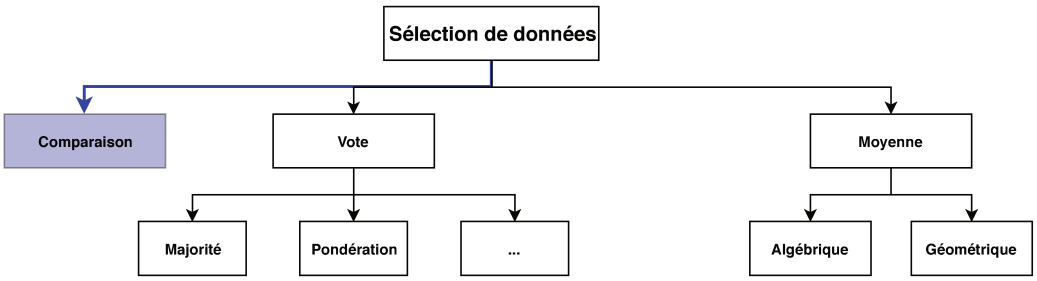


FIGURE 3.7 – Catégorie Sélection de Données issue du TR

Prenons maintenant le cas du mécanisme PBR. Nous reprenons le PBR qui est représenté sous sa forme BPA sur la figure 3.2 et décomposons le en actions élémentaires le constituant. Nous nous intéressons aux actions des *Before*, du *Proceed* et des l'*After*.

Dans un premier temps, prenons l'*After* du PBR que ça soit du côté Primaire ou Secondaire. Précédemment, l'*After* était conçu avec les deux rôles (Primaire et Secondaire) avec dans le premier cas la capture de l'état de l'application et dans le second cas la mise à jour de l'état de la réplique. La catégorie Gestion d'État a déjà été réalisé dans cette section et les actions extraites des *After* du PBR confirment nos sous-catégories.

Dans un deuxième temps, prenons les *Before*. Comme nous l'avons exprimé dans le chapitre précédent, les *Before* du PBR sont vides (dans notre cas) et ne servent qu'en cas d'adaptation car la manipulation des communications est délicate et la substitution plus facile à réaliser. Il n'apporte donc pas de modifications à nos trois catégories. Cependant, nous le conservons pour l'instant dans la modélisation pour conserver les propriétés pour l'adaptation. De même le *Proceed* étant un élément communs à tous les FTM car servant d'interface avec l'application, il est identique à celui du TR.

Dans un troisième temps, nous devons considérer les Services Communs de TaF qui s'exécutent parallèlement au graphe de composants BPA, ici les *Crash Detector* (CD). Nous avons implémenté nos mécanismes duplex avec un *watchdog* qui remettait à zéro un compteur. Ce n'est pas la seule façon de faire, il existe d'autres techniques comme celle du *heartbeat* [Coekaerts 2008]. Nous pouvons considérer une catégorie **Détection de Crash** qui regroupe ces différentes techniques.

La figure 3.8 représente le PBR modélisé avec l'utilisation des catégories génériques et des sous-catégories tirées des choix de conception qui sont utilisées. Nous avons conservé le *Before* pour l'adaptation. Nous avons remplacé le *Proceed* par de la Synchronisation par message synchrone avec traitement de données. Nous avons remplacer les *After* par de la Gestion d'État avec transmission de l'état par message de *checkpoint*.

Nous appliquons maintenant cette méthode au LFR pour étendre nos catégories. Les *Before* ont pour actions de la synchronisation de réplique par message asynchrone (transmission/réception de la requête). Le *Proceed* du LFR est identique au PBR et au TR. Enfin, les *After* ont pour actions de la synchronisation de réplique également par message synchrone (Le *Leader* synchronise le *Follower* après l'exécution parallèle de l'application) .

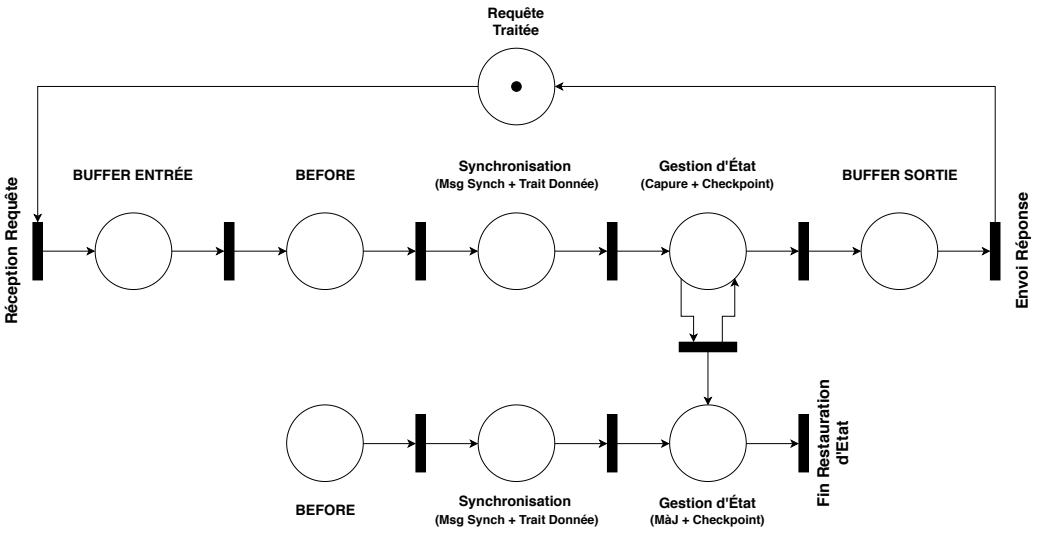


FIGURE 3.8 – Modélisation d'un PBR avec des catégories génériques

Dans la prochaine section, nous analyserons un ensemble de FTM répondant à différents modèles de fautes que nous décomposerons comme précédemment en catégories. Le but de cette section va être de compléter nos catégories d'action en analysant un ensemble de schémas conceptuels de FTM tolérant des erreurs diverses. Les implémentations étant multiples pour une même stratégie de tolérance aux fautes, nous garderons un point de vue conceptuel et non d'implémentation. Notre but n'est pas ici de faire une liste complète des catégories d'actions mais de montrer qu'avec un ensemble restreint de catégories, nous pouvons concevoir un nombre important de mécanismes.

2 Classification en action

En nous intéressant aux actions de SdF qui composent nos FTM, nous avons pu remarquer qu'il est possible de classifier des catégories d'action que nous spécialisons selon la conception et l'implémentation du FTM et de l'application. Ces catégories nous permettent d'augmenter le niveau de réutilisation de code en plus d'affiner la granularité des FTM. Nous avons précédemment explicité une méthode pour discerner les actions de SdF constituant un FTM et pour les classer dans des catégories que nous avons appliquées au PBR et au TR.

Nous allons dans cette section analyser la conception d'un ensemble assez large de mécanismes de tolérance aux fautes pour en déduire des actions de SdF et les ranger dans de nouvelles catégories ou dans celles existantes. Nous formaliserons ces actions en utilisant les principes d'héritage et de surcharge de méthodes abstraites utilisée dans le langage C++.

2.1 Décomposition d'un ensemble de FTM

Nous étudions dans la suite de ce chapitre différents patrons de conception de FTM pour classer les différentes actions qui les composent. Ces FTM sont associés à un type de faute à tolérer et possèdent des implémentations diverses pour compléter au mieux notre analyse des actions de SdF.

Plusieurs travaux ont été réalisés pour modéliser des schémas de conception de FTM. Les travaux les plus connus sont regroupés dans des articles tels [Douglass 1999], [Douglass 2003] ou encore [Hanmer 2013]. Ces travaux proposent plusieurs schémas conceptuels de FTM pour les systèmes critiques, basés sur des méthodes de tolérance aux fautes et intégrant certaines modifications pour augmenter le niveau de sûreté de ces modèles. Nous nous sommes aidés des travaux [Armoush 2010] listant un certain nombre de patrons de conception pour commencer l'analyse. De plus, nous avons utilisé la figure 2.8, page 52, qui définit un ensemble de FTM pour compléter notre liste. Enfin, nous analyserons différentes conceptions de ces patrons de conception de FTM pour définir les différentes catégories et sous-catégories.

En allant de la redondance physique aux mécanismes de vote en passant par les mécanismes de contrôle de vraisemblance, nous avons analysé un ensemble pertinent de mécanismes pour en extraire nos catégories génériques ainsi que les sous-catégories qui les composent. Ces mécanismes sont regroupés en fonction du type de fautes qu'ils tolèrent :

- **Fautes matérielles permanentes par crash :**
  1. *Homogenous Duplex Pattern* (HmD) ;
  2. *Heterogenous Duplex Pattern* (HtD) ;
  3. Réplication Semi-Passive ;
  4. Réplication Passive (PBR) ;
  5. Réplication Semi-Active (LFR) ;

6. *Roll-Forward Recovery.*

- **Fautes matérielles en valeur :**
  1. *M-out-of-N Pattern* (MooN) ;
  2. *Statefull Time Redundancy* (TR) ;
  3. Vérification d'assertion ;
  4. *Sanity Check Pattern* (SnC).
- **Fautes logicielles :**
  1. *N-Version Programming* (NVP) ;
  2. *N-Self Checking Programming Pattern* (NSCP) ;
  3. *Acceptance Voting* (AV) ;
  4. *Recovery Block Pattern* (RB).

Nous allons maintenant étudier chaque FTM en fonction du type de fautes qu'ils tolèrent pour extraire les actions de SdF qui les composent. Une fois ces actions définies, nous créerons ou compléterons les catégories auxquelles ces actions appartiennent.

2.1.1 Les fautes matérielles permanente par crash

Dans cette catégorie, nous retrouvons les quatre types de techniques de réplication ainsi que le *Roll-Forward Recovery.*

***Homogenous Duplex Pattern* (HmD) :** Deux répliques sur du matériel et du logiciel identiques exécutent la requête. Un arbitre sélectionne l'une des deux réponses avant de l'envoyer.

***Heterogenous Duplex Pattern* (HtD) :** Deux répliques sur du matériel et du logiciel différent exécutent la requête. Un arbitre sélectionne l'une des deux réponses avant de l'envoyer.

Concernant la réplication active, plusieurs implémentations sont explicitées dans des travaux comme [Chérèque 1992]. Ici, nous avons deux mécanismes qui conceptuellement sont identiques mais qui sont implémentés différemment. Nous avons choisi ces deux FTM pour confirmer que des différences d'implémentation n'ont pas d'impact sur les choix de conception lors de la classification des actions. Les hypothèses concernant les caractéristiques applicatives AC et les besoins en ressources RS sont respectivement le déterminisme de l'application, et la cohérence des entrées et la présence de deux calculateurs distincts.

De ces deux descriptions, les deux répliques reçoivent les requêtes de la part du Client, exécutent celles-ci (synchronisation équivalente au *Proceed*) puis envoient leur réponse à un organisme (l'arbitre) qui va utiliser une action de sélection de données pour choisir l'un des deux réponses. Il nous faut donc au moins trois catégories pour concevoir ces deux mécanismes, à savoir la catégorie **Synchronisation**, la catégorie **Sélection de Données** et la catégorie **Détection de Crash**. De plus, pour garder la cohérence des entrées, à savoir que les deux répliques recevrons les

mêmes entrées, nous pouvons ajouter une action de contrôle d'intégrité. Nous créons ainsi une nouvelle catégorie, la **Détection de Corruption** comme représenté sur la figure 3.9 . Cette catégorie a comme partie générique les actions d'encodage et de décodage des données. Nous la spécifions en fonction des techniques de contrôle d'intégrité comme le contrôle de parité [Gallager 1962], le CRC [Boudreau 1971], le *checksum* [Braden 1989] ou encore une signature cryptographique.

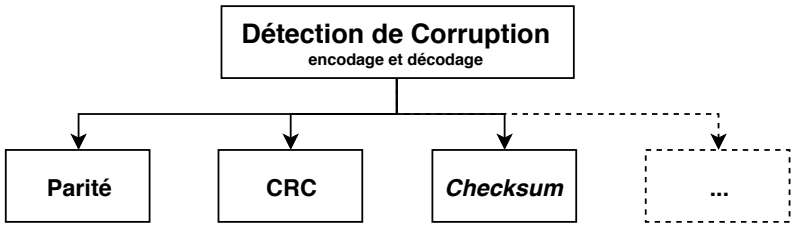


FIGURE 3.9 – Catégorie générique Détection de Corruption

Réplication Semi-Passive : C'est un fonctionnement équivalent à celui d'un PBR. Toutes les répliques reçoivent la requête et le Primaire est choisi par consensus. Celui-ci capture l'état de l'application après exécution et la transmet aux répliques.

Concernant la réplication semi-passive, il s'agit d'un PBR avec une sélection aléatoire du Primaire. Les hypothèses AC et RS sont respectivement l'accès à l'état, la cohérence des entrées et la présence d'au moins deux calculateurs distincts.

Nous voyons ici une nouvelle action de synchronisation, le consensus. Il s'agit d'envoyer un message asynchrone 1-to-N pour définir quel serveur fournira son état et/ou sa réponse aux autres. Le consensus garantit qu'une seule réplique parmi N sera élue Primaire. Le reste des actions est identique au PBR qui a déjà été traité dans la section précédente. Plusieurs implémentations sont illustrées dans [Defago 1998].

Les réplications passives (PBR) et semi-actives (LFR) ont été traitées dans la section précédente et ont permis de définir les premières catégories d'actions. Nous pouvons trouver d'autres implémentations de ces deux mécanismes dans les travaux [Fabre 1995] [Powell 1991].

***Roll-Forward Recovery* :** le principe est toujours d'avoir une réplication avec une ou plusieurs répliques qui exécutent les requêtes en parallèle. Lorsqu'une réplique tombe, elle est relancée avec une vérification de bon fonctionnement dans un nouvel état stable postérieur à l'état au moment du crash.

Concernant le *Roll-Forward Recovery* et contrairement aux schémas de réplication passive, on recherche un nouvel état de reprise pour assurer un service nominal ou dégradé. Cet état peut être défini à l'avance (voiture à l'arrêt sur le bas-côté en cas de défaillance) ou récupéré à partir de répliques s'exécutant en parallèle. Les hypothèses AC et RS sont respectivement l'accès à l'état, et la présence d'au moins deux calculateurs distincts.

Ce mécanisme n'ajoute pas d'autres catégories génériques ni de sous-catégories à celles déjà présentées. Nous pouvons trouver des exemples de conception de ce mécanisme dans les travaux [Xu 1996].

		Synchronisation	Gestion d'État	Sélection de Données	Détection de Crash	Détection de Corruption	Service Commun de TaF
Réplication active (HmD & HtD)	Client	Émission requête : message Asyncrone 1-to-2 Pas de traitement de données				Encodage données : CRC	
	Primaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données			Détection de crash : Heartbeat	Décodage données : CRC	Recouvrement
	Secondaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données			Détection de crash : Heartbeat	Décodage données : CRC	Recouvrement
	Arbitre	Réception réponse : message Asyncrone -to-1 Pas de traitement de données		Sélection Réponse : Vote majorité relative			
Réplication Semi-Passive	Client	Consensus (Émission requête) : message Asyncrone 1-to-2 Traitement de données					
	Primaire	Consensus (Réception requête) : message Asyncrone 1-to-1 Traitement de données	Capture + ckpt : DMTCP		Détection de crash : contrôle d'activité		Recouvrement Sauvegarde : Mémoire locale
	Secondaire	Consensus (Réception requête) : message Asyncrone 1-to-1 Traitement de données	Mise-à-Jour + ckpt : DMTCP		Détection de crash : Contrôle d'activité		Recouvrement Sauvegarde : Mémoire locale
Réplication Semi-Active	Primaire	Émission message : message Asyncrone 1-to-1 Pas de traitement de données			Détection de crash : "I'm Alive" message		Recouvrement
	Secondaire	Réception message : message Asyncrone 1-to-1 Pas de traitement de données			Détection de crash : "I'm Alive" message		Recouvrement
Réplication Passive	Primaire		Capture + ckpt : Getter/Setter		Détection de crash : "I'm Alive" message		Recouvrement Sauvegarde : Mémoire locale
	Secondaire		Mise-à-Jour + ckpt : Getter/Setter		Détection de crash : "I'm Alive" message		Recouvrement
Roll-Forward Recovery	Primaire		Capture + ckpt : Diskless Checkpoint		Détection de crash : Heartbeat		Recouvrement Sauvegarde : Mémoire locale
	Secondaire		Mise-à-Jour + ckpt : Diskless Checkpoint		Détection de crash : Heartbeat		Recouvrement Sauvegarde : Mémoire locale

TABLE 3.1 – Synthèse des Catégories : Fautes matérielles par crash

Le tableau 3.1 récapitule les actions de SdF élémentaires de chaque patron de conception de FTM en fonction des catégories auxquelles elles appartiennent. Nous détaillons une conception particulière d'un FTM en choisissant une sous-catégorie par patron et par catégorie. Il est possible de modifier la sous-catégorie pour avoir une autre conception d'un FTM. Pour la réplication active (HmD ou HtD), nous faisons du contrôle d'intégrité avec une technique de CRC. Nous pouvons tout à fait utiliser une autre technique et concevoir ainsi un autre FTM de réplication active.

Comme nous pouvons le voir dans ce tableau, nous avons besoin d'actions pour gérer la synchronisation inter-répliques ou l'état de l'application. Nous avons donc toujours besoin d'une catégorie Synchronisation séparée en deux branches en fonction du type de communication (synchrone ou asynchrone). Ces branches sont divisées en fonction du traitement ou non des données.

Concernant les Services Communs de tolérance aux fautes (SC de TaF), comme nous avons de la duplication, nous avons besoin de mécanismes de recouvrement en cas de crash. Nous avons également besoin, pour certains FTM de stockage sur mémoire locale persistante.

Il est intéressant de noter que pour les deux techniques de réplication active, l'hétérogénéité du matériel et du logiciel de l'application n'a pas d'impact sur les actions de SdF lors de la conception du mécanisme. Les données d'entrée/sortie sont encapsulées dans un message standard à l'intérieur du FTM et désencapsulées lors de la synchronisation avec le Serveur. Ces deux actions de traitement de données sont dépendantes de l'implémentation.

Nous allons maintenant nous pencher sur les FTM tolérant les fautes matérielles en valeur et compléter ainsi la liste des catégories et des sous-catégories qui les composent.

2.1.2 Les fautes matérielles en valeur

Dans cette catégorie, nous retrouvons quatre façons de contrôler si une erreur transitoire ou permanente en valeur est apparue dans l'application. La première est d'exécuter l'application sur N calculateurs et de faire un vote (M out-off N ou MooN). La deuxième est d'exécuter deux fois l'application avec le même état initial et de comparer les deux réponses. Si elles sont différentes, on peut faire une troisième exécution, comparer les trois réponses et d'envoyer la bonne (TR). La troisième est de faire un test de vraisemblance pour vérifier que la donnée est crédible (Vérification d'assertion). La quatrième est de comparer la consigne et la réponse (SnC). Dans le cas d'une détection d'erreur, on peut changer de chaîne de traitement (mécanisme de duplication), ou envoyer un message d'erreur pour prévenir l'utilisateur et stopper le traitement dans un état sûr.

***M-out-of-N Pattern* (MooN) :** Plusieurs répliques exécutent la requête puis un organisme de vote sélectionne la réponse.

Concernant le patron MooN, l'une des techniques les plus connues utilisant ce schéma est le *Triple Modular Redundancy* (TMR) avec 3 répliques et un vote majoritaire pour sélectionner la réponse. Les hypothèses AC et RS sont respectivement

le déterminisme de l'application et ses répliques, et la cohérence des entrées et la présence d'au moins trois calculateurs distincts.

Nous avons étudié différentes conceptions du TMR dans différents travaux [Lyons 1962]. Le mécanisme est constitué d'actions de contrôle d'intégrité pour éviter la corruption des entrées (**Détection de Corruption**), d'envoi des réponses entre les répliques et un arbitre (Synchronisation). De plus, cet arbitre va choisir la réponse, souvent par vote à majorité absolue (Sélection de Données).

Le TR, comme le PBR et le LFR, a été étudié dans la section précédente. Ce mécanisme contenait des actions appartenant aux catégories de **Gestion d'État**, de **Sélection de Données**, de **Synchronisation** et utilisait des Services Communs de TaF de sauvegarde sur mémoire locale persistante.

La vérification d'assertion est une stratégie de tolérance aux fautes très répandue dans l'industrie automobile et qui permet de fournir un test d'acceptance rapide. Elle utilise une expression logique pour définir des plages acceptables pour les valeurs de données. Nous considérons que cette stratégie n'a qu'une action appartenant à la catégorie **Sélection de Données**. Nous enrichissons donc cette catégorie en ajoutant la sous-catégorie Vérification d'assertion au même niveau que les techniques de vote, de comparaison ou de moyenne. Des exemples de conception et d'utilisation de la vérification d'assertion dans l'automobile sont disponibles dans [Lu 2009].

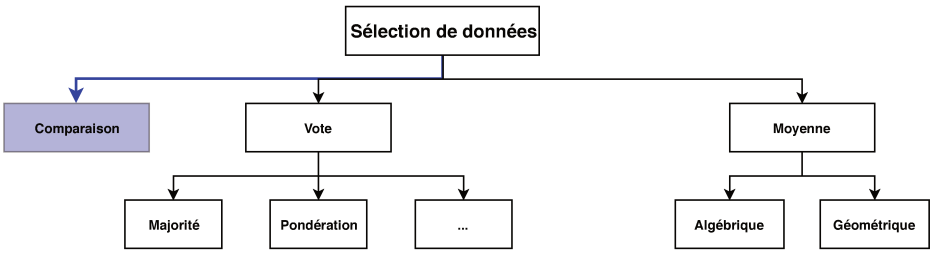


FIGURE 3.10 – Catégorie Sélection de Données

***Sanity Check Pattern* (SnC) :** Une chaîne de surveillance est ajoutée à la chaîne de traitement et compare la consigne théorique, la consigne calculée ainsi que la réponse d'un actionneur. Cela permet de vérifier le bon fonctionnement de la chaîne de traitement et de l'actionneur associé (P. ex. vérification dans une boucle fermée de régulation du bon fonctionnement du régulateur).

La technique *Sanity Check* (SnC) est très souvent liée à l'automatique pour les systèmes de contrôle-commande. Les hypothèses RS sont la cohérence des entrées et la présence d'au moins deux calculateurs distincts.

Ce patron de conception est assez spécial car il contient deux chaînes fonctionnelles différentes. La chaîne de traitement a besoin d'une action de contrôle d'intégrité pour ses entrées (**Détection de Corruption**) et d'envoi de sa consigne et de sa réponse (**Synchronisation**). La chaîne de contrôle, quant à elle, a besoin d'une action de réception des consignes et de la réponse (**Synchronisation**) et de comparer les différentes valeurs (**Sélection de donnée**).

		Synchronisation	Gestion d'État	Sélection de Données	Détection de Crash	Détection de Corruption	Service Commun de TaF
MooN (TMR)	Primaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Secondaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Tertiaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Arbitre	Réception réponse : message Asyncrone 3-to-1 Pas de traitement de données		Sélection Réponse : Vote majorité relative			
Time Redundancy			Capture + variable Mise à jour + variable : LibCkpt				Sauvegarde : Mémoire Locale
Verification d'assertion				Sélection Réponse : Assertion			
Sanity Check	Traitement	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données				Décodage des données : Checksum	
	Moniteur	Réception réponse : message Asyncrone 1-to-1 Pas de traitement de données					

TABLE 3.2 – Synthèse des Catégories : Fautes matérielles en valeur

Finalement, comme le montre le tableau 3.2, ces quatre techniques utilisent les mêmes catégories que précédemment. Ce qui va différencier les catégories ici va être la technique utilisée pour sélectionner la valeur dans le cas du MooN. En effet, nous pouvons utiliser un vote à majorité absolue ou relative, un vote pondéré, une comparaison ou tout autre algorithme de décision comme le montre la figure 3.10. Il est intéressant de constater que six catégories d'actions génériques permettent finalement de concevoir un très grand nombre de mécanismes. Si N représente le nombre de FTM et C le nombre de catégorie alors nous obtenons :

$$N > 2^C$$

Avec nos 6 catégories, nous sommes en mesure de concevoir au moins 64 mécanismes différents sans prendre en compte les considérations d'implémentation.

Nous allons maintenant nous pencher sur les FTM tolérant les fautes logicielles et compléter également la liste des catégories et des sous-catégories qui les composent.

2.1.3 Les fautes logicielles

Les fautes logicielles sont regroupées sous la même bannière et n'ont pas de différentiation aussi forte que les fautes matérielles. Dans cette catégorie, nous retrouvons quatre façons de contrôler si une erreur logicielle est apparue dans l'application. Ce sont principalement des mécanismes ressemblant fortement à leurs homologues pour les fautes matérielles.

***N-Version Programming* (NVP) :** N versions de l'application diversifiées sont exécutées sur le même calculateur en parallèle pour effectuer la même tâche sur la même entrée et produire N sorties dont une est sélectionnée par un vote.

Le *N-Version Programming Pattern* (NVP) est équivalent au MooN mais avec de la diversité logicielle [Avizienis 1995]. Pour la conception d'un mécanisme selon ce patron, nous devons avoir de la diversité logicielle pour au moins trois applications.

Encore une fois, pas de nouvelles catégories ni sous-catégories. Il nous faut des action de **Synchronisation** pour que chaque réplique transmette à un arbitre sa réponse et une action de **Sélection de Données** pour choisir une réponse parmi les N proposées. On peut agrémenter la conception de ce FTM avec du contrôle d'intégrité (**Détection de Corruption**) pour assurer la cohérence des entrées entre la machine Client et la machine contenant les Serveurs.

***Acceptance Voting* (AV) :** N versions de l'application diversifiées sont exécutées puis une vérification d'assertion valide chaque réponse avant qu'une vote en sélectionne une.

Le principe de l'*Acceptance Voting* est similaire au NVP mais avec un test d'acceptance de type vérification d'assertion avant la sélection de la réponse. Les hypothèses sont identiques au NVP.

Il faut donc non plus une mais deux actions de **Sélection de Données**, à savoir un test d'acceptance pour chaque sortie et un vote effectué par un arbitre

pour sélectionner une réponse parmi celles qui sont valides.

***N-Self Checking Programming Pattern* (NSCP) :** N version de l'application s'exécutent puis sont comparées deux à deux avant de faire une sélection parmi les réponses retenues [Laprie 1990].

Les hypothèses sont identiques à l'AV et au NVP.

Ce patron de conception ressemble à l'AV mais effectue, à la place du test d'acceptance, une comparaison entre deux réponses puis l'arbitre sélectionne la réponse parmi les N/2 proposées.

***Recovery Block Pattern* (RB) :** Plusieurs versions diversifiées de l'application sur un même calculateur. La première version sauvegarde son état puis s'exécute et une vérification d'assertion valide la réponse. Si elle est incorrecte, une seconde version restaure l'état initial et s'exécute puis une nouvelle vérification d'assertion valide la réponse.

Le *Recovery Block* contient N versions diversifiées sur un calculateur. La première sauvegarde son état, s'exécute puis un test valide la réponse. Si elle est incorrecte, la seconde version doit restaurer son état pour être dans le même état initial que la première version puis elle s'exécute jusqu'à ce qu'un test d'acceptance valide une réponse ou que toutes les versions aient retourné un résultat incorrect. Nous avons besoin de diversité logicielle pour au moins deux versions de l'application et nous devons pouvoir avoir accès à l'état de l'application.

C'est un FTM intéressant à analyser mais difficile à utiliser. En effet, la difficulté ici c'est que les versions doivent être diversifiées tout en ayant un état identique pour que chaque version puissent être restaurées. Le RB nécessite donc une action de **Gestion d'État** et une action de **Sélection de Réponse**.

Les tableaux 3.3 et 3.4 montrent ici qu'après la décomposition de ces quatre mécanismes tolérant les fautes logicielles, il n'y a pas d'autres catégories que celles énoncées précédemment. La plupart des mécanismes de tolérance aux fautes sont proches en termes d'architectures les uns des autres. Il n'est donc pas surprenant de trouver beaucoup de point commun entre ceux-ci. Ici, l'arbitre est un élément externe au FTM qui récupère les réponses (Synchronisation) et les sélectionne. Le nombre de réponse peut varier en fonction du patron choisi. Soit il sélectionne une réponse parmi N (NVP), soit parmi M<N réponses (AV), soit P<N/2 (NSCP). Il n'est pas rare de retrouver ces Services de TaF (p.ex. composant de recouvrement pour les réplications ou de détection de crash) dans la conception et l'implémentation de ces mécanismes.

		Synchronisation	Gestion d'État	Sélection de Données	Détection de Crash	Détection de Corruption	Service Commun de TaF
N-Version Programming	Primaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Secondaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Tertiaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Arbitre	Réception réponse : message Asyncrone 3-to-1 Pas de traitement de données		Sélection Réponse : Moyenne Algébrique			
Acceptance Vote	Primaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données		Sélection Réponse : Assertion			
	Secondaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données		Sélection Réponse : Assertion			
	Arbitre	Réception réponse : message Asyncrone 2-to-1 Pas de traitement de données		Sélection Réponse : Vote pondéré			Sauvegarde : Mémoire Locale
Recovery Blocks	Primaire		Capture + variable Mise à jour + variable : DMTCP				Sauvegarde : Mémoire Locale
	Secondaire		Capture + variable Mise à jour + variable : DMTCP				Sauvegarde : Mémoire Locale

Table 3.3 – Décomposition en catégorie des FTM tolérant les fautes logicielles (NVP, AV & RB)

		Synchronisation	Gestion d'État	Sélection de Données	Détection de Crash	Détection de Corruption	Service Commun de TaF
N-SelfChecking Programming	Primaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Secondaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Comparateur1	Réception réponse : message Asyncrone 2-to-1 Émission réponse Arbitre : message Asyncrone 1-to-1		Sélection Réponse : Comparaison			Sauvegarde : Mémoire Locale
	Tertiaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Quaternaire	Émission réponse : message Asyncrone 1-to-1 Pas de traitement de données					
	Comparateur2	Réception réponse : message Asyncrone 2-to-1 Émission réponse Arbitre : message Asyncrone 1-to-1		Sélection Réponse : Comparaison			Sauvegarde : Mémoire Locale
	Arbitre	Réception réponse : message Asyncrone 2-to-1 Pas de traitement de données		Sélection Réponse : Vote pondéré			Sauvegarde : Mémoire Locale

Table 3.4 – Décomposition en catégorie des FTM tolérant les fautes logicielles (NSCP)

2.1.4 Synthèse des catégories

Après analyse de ces quatorze mécanismes, nous avons extrait des tableaux six catégories génériques : **(1)** Synchronisation, **(2)** Gestion d'État, **(3)** Détection de crash, **(4)** Sélection de Données, **(5)** Détection de Corruption, et **(6)** Service de TaF. Il est important de noter que nous nous sommes basés sur des schémas de conception et non des implémentations pour faire ces tableaux. Étant donné qu'il existe une multitude de façon d'implémenter un FTM, il n'aurait pas été pertinent d'en faire l'analyse pour en tirer les catégories génériques d'action de SdF.

A ces six catégories, nous pouvons en rajouter une septième qui est **(7)** la **Détection Temporelle** qui n'était pas présente dans les schémas mais qu'il faut prendre en compte. En effet, donner une valeur périmée à un utilisateur (actionneur, humain...) peut entraîner de graves catastrophes. Si nous prenons un exemple automobile avec le Freinage Actif d'Urgence, si la détection d'un piéton devant la voiture est retardée à cause d'une valeur de capteur datant de quelques dixièmes de seconde, le véhicule pourrait réagir trop tard et le renverser.

Maintenant que nous avons défini nos sept catégories, nous allons nous intéresser à la spécificité des actions qui les composent et commencer à entrevoir les premières classes abstraites en C++ utilisées pour les coder. Nous allons passer du nœud ROS à l'objet

2.2 Définition de classes d'action

Nous avons mentionné dans la partie précédente le regroupement des actions de SdF dans une même catégorie. Cependant, ces catégories sont génériques et utiles pour augmenter la réutilisation du code mais nous sommes incapable de concevoir un FTM juste avec elles. De plus, nous avons divisé ces catégories en sous-catégorie en fonction de choix de conception. Nous allons maintenant utiliser un langage orienté-objet, ici le langage C++, pour projeter nos catégories et sous-catégories sur des classes mères et filles. Les classes mères abstraites contiennent des méthodes virtuelles regroupant les parties génériques. Les classes filles héritent des classes mères les méthodes qu'elles surchargent pour implémenter les actions.

Les fonctionnalités du C++ comme les classes abstraites, l'héritage et les surcharges de méthodes sont très intéressantes pour implémenter les actions de SdF. En effet, cela permet aux objets appartenant à la même catégorie de se partager des méthodes et des variables communes. Il est à noter que cette façon de concevoir ces objets contenant les actions de SdF est utile pour implémenter la même action avec des techniques différentes.

Les sept catégories que nous avons définies peuvent être amenées à évoluer ou à être modifiées. De même nous avons défini les parties communes de nos catégories mais ce n'est pas la seule façon de faire et il est possible qu'elles puissent être modifiées.

Prenons maintenant comme exemple la catégorie Gestion d'État pour expliciter l'utilité du langage C++. Premièrement, nous avons deux méthodes communes qui

sont la capture et la mise à jour de l'état. En fonction de la technique utilisée, ces deux méthodes peuvent avoir des implémentations différentes. Nous créons ainsi une classe mère abstraite Gestion d'État avec ces deux méthodes virtuelles. Nous créons ensuite autant de classes filles que de techniques de *checkpointing*.

Deuxièmement, dans le cas d'une technique type *Setter/Getter*, nous avons une autre différence de conception comme entre le PBR et le TR, la transmission de l'état de l'application. Nous créons donc deux autres classes filles pour cette technique de *checkpointing* qui ajoutent une transmission d'état par message de checkpoint pour l'une et par variable partagée pour l'autre.

Une fois les classes, les méthodes et les variables définies, il ne reste plus qu'à instancier l'objet contenant la ou les actions de SdF sur un processus, un nœud ou tout autre unité d'exécution.

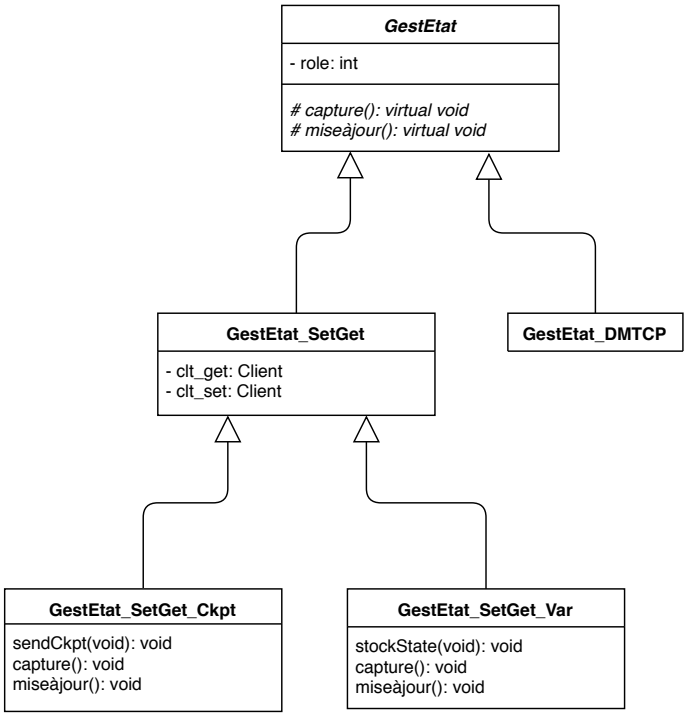


FIGURE 3.11 – Classe abstraite GestionEtat

La figure 3.11 représente un diagramme de classe partiel avec la technique de *checkpointing Getter/Setter*. Dans le PBR sera implémenté l'objet issu de la classe GestEtat_GetSet_Ckpt qui hérite de la classe GestEtat_GetSet qui elle même hérite de la classe GestEtat. Dans le TR sera implémentée la classe GestEtat_GetSet_Var.

Le même travail est effectué pour les autres catégories nécessaires à l'implémentation des trois mécanismes du Chapitre 2. Nous allons maintenant implémenter le PBR et le LFR+TR pour évaluer le gain de généricité en vue d'améliorer la réutilisation de code. Nous en déduirons les avantages et les inconvénients de cette approche.

3 Analyse critique de l'approche sur la généricité

Dans cette section, nous allons évaluer, du point de vue de la généricité, l'utilité de la nouvelle granularité et des catégories que nous avons extraite des analyses de schémas de conception de FTM. Nous allons évaluer si cette approche améliore deux critères : **(1)** la capacité à implémenter nos FTM à partir des catégories et sous-catégories et **(2)** la réutilisation du code pour les implémenter.

Nous allons donc concevoir à nouveau nos FTM avec ces actions issus de nos catégories génériques et analyser l'impact qu'à ce changement de granularité sur les méthodes utilisées pour l'AFT. Nous prendrons la même application Client/Serveur que dans le chapitre précédent et nous implémenterons nos mécanismes avec, pour l'instant, une action par composant.

3.1 Implémentation des catégories d'actions

Dans la section précédente, nous avons défendu l'idée de projeter nos catégories et sous-catégories d'actions sur des classes C++. Dans le Chapitre 2, nous avons implémenté nos composants en séparant la création du nœud (boîtier) des actions de SdF regroupées dans des objets C++ (puce). Cependant, ces objets avaient une granularité trop grossière. Nous allons donc voir l'impact de ce changement de granularité sur l'implémentation précédente sur ROS.

Nous conservons de notre implémentation précédente l'architecture à composants, les nœuds *Client*, *Serveur*, l'*Adaptation Engine* et le couple Moniteur/Opérateur. Ici, nous passons d'un mécanisme constitué d'un ensemble P-BPA à un ensemble P-{A} où A représente les actions de SdF. Nous gardons les *Protocol* pour servir de râtelier et conserver la méthode de composition de l'approche par composants substituables. Les communications uniformes (qui ne dépendent pas de l'action dans le nœud) restent identiques à celles du Chapitre 2. En pratique, seule la "puce" contenant les actions de SdF est remplacée par une "puce" contenant une action élémentaire. Nous illustrons cette implémentation en reprenant le mécanisme PBR et la composition de mécanisme LFR+TR.

La figure 3.12 représente l'implémentation d'un PBR avec une action élémentaire par nœud. Les *Before* sont remplacés par un nœud vide ne faisant que transférer la requête pour être substitués lors d'une adaptation. Le *Proceed* est remplacé par un nœud contenant une action de synchronisation avec traitement de données et une communication de type synchrone. Une action de gestion d'état avec capture de l'état de l'application pour le Primaire, de mise-à-jour de l'état pour le Secondaire et deux Services Communs de TaF remplacent des *After*.

En absence de *Before* ou de ce nœud vide, nous pourrions utiliser l'édition de lien dynamique pour insérer un nœud entre le *Protocol* et la communication avec le *Serveur* mais l'utilisation du *remapping* à l'initialisation de ROS est plus simple à appliquer.

Comme énoncé dans la section précédente, le PBR a bien des nœuds implémentant des actions issues de quatre classes mères : **(1)** la **Gestion d'État** avec

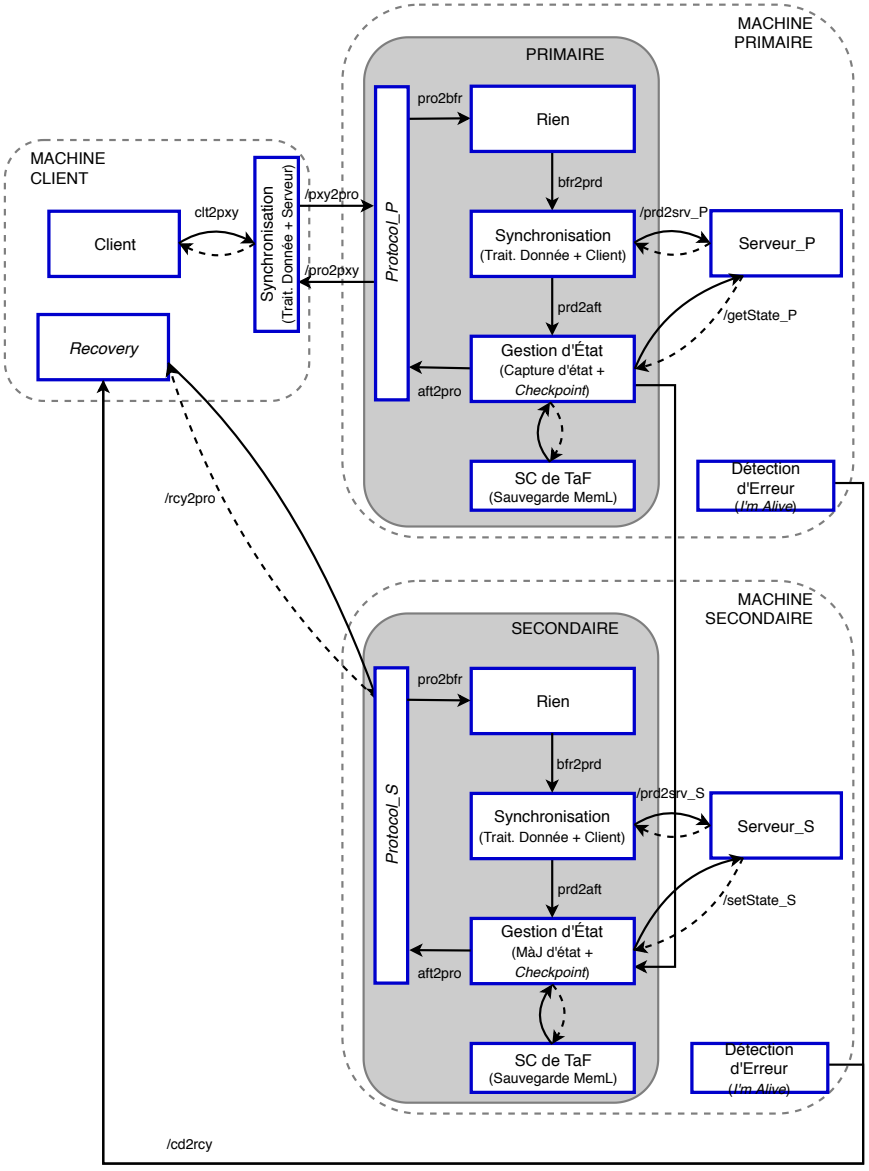


FIGURE 3.12 – Implémentation sous ROS du PBR avec la nouvelle granularité

transmission d'état par message de checkpoint, **(2)** la **Synchronisation** par message synchrone avec traitement de données (Anciennement *Proceed*), (3) un **Service de TaF** avec sauvegarde de l'état sur mémoire local, et **(4)** la **Détection d'Erreur** avec un *Crash Detector* envoyant un message "*I'm Alive*" périodiquement. Il faut ajouter bien sûr un nœud de recouvrement. Ce dernier ne peut pas être générique car son action va dépendre du contexte applicatif et de la stratégie de recouvrement en cas de crash.

La figure 3.13 représentant le mécanisme LFR+TR implémenté avec des nœuds issus cinq classes mères différentes. Ici, la classe **Sélection de Données** pour com-

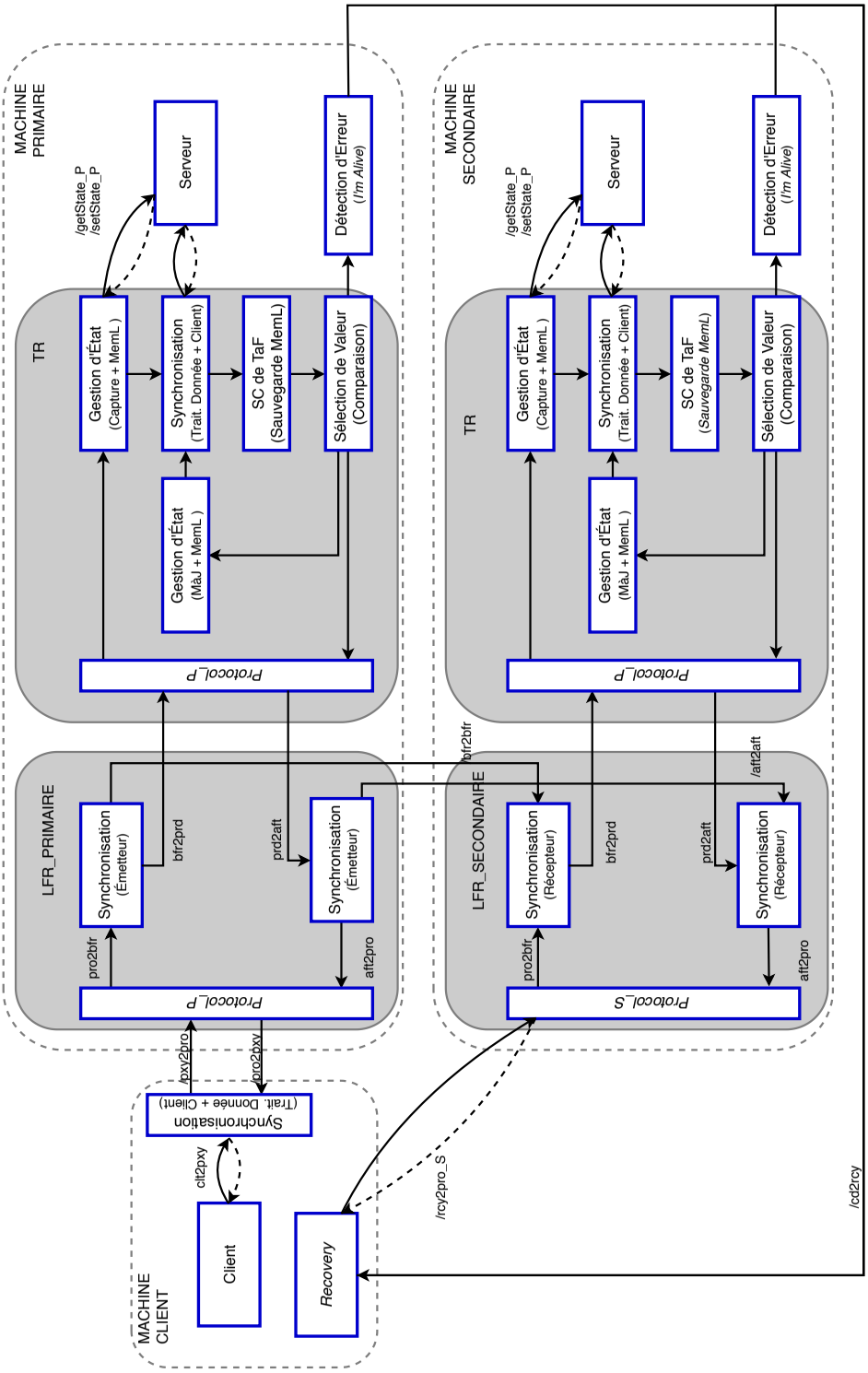


FIGURE 3.13 – Implémentation sous ROS du TR avec la nouvelle granularité

parer les réponses de l'application s'ajoutent aux quatre précédentes. Nous pouvons donc, à partir de cinq classes mères issues des catégories, concevoir cinq mécanismes (PBR, TR, LFR, PBR+TR, LFR+TR) au lieu d'avoir 11 composants (cinq *Before*, cinq *After* et un *Proceed*). Nous avons donc bien une amélioration de la généricité pour la réutilisation de code.

Ces implémentations sous ROS mettent en lumière trois points : **(1)** le fonctionnement interne des FTM reste identique malgré la séparation en actions, **(2)** dans le cas d'une reconfiguration avec un FTM duplex, il faut également substituer des nœuds implémentant les actions **(3)** le *Before* utilisé dans l'approche BPA n'a aucune utilité dans notre exemple autre qu'être présent pour être remplacé lors de l'adaptation avec un autre FTM.

Pour détailler le deuxième point, prenons un exemple. Dans le cas de la perte du Primaire, le Secondaire devient le nouveau Primaire. Or, l'action de Gestion d'État dans le Secondaire est une mise à jour et non une capture d'état. Il faut donc substituer le nœud de mise à jour par un nœud de capture d'état. Ce point met en lumière un problème de granularité non plus dans la conception (le composant a été remplacé par l'action) mais dans l'implémentation. Une unité d'exécution pour une action n'est peut être pas le meilleur choix à faire pour implémenter nos actions élémentaires.

Nous allons maintenant étudier plus en détail les avantages et les inconvénients de ce changement de granularité et déterminer les améliorations possibles à apporter à cette approche.

3.2 Avantages et Inconvénients de l'approche

Nous avons implémenté dans cette section les FTM en utilisant non plus le découpage en un ensemble P-BPA mais en un ensemble d'actions de SdF élémentaire. Les avantages de cette approches sont multiples concernant la réutilisation du code.

Le premier avantage est de pouvoir concevoir et implémenter un nombre important de FTM avec un nombre faible de catégorie générique. Nous avons extrait de notre analyse de quatorze patrons de conception de FTM et de leur implémentation, six catégories génériques. En spécifiant ces catégories et en se limitant au choix de conception, il est possible, par exemple, de concevoir au moins dix-huit *Triple Modular Redundancy* à partir des trois catégories Synchronisation, Sélection de Données (6 techniques différentes) et Détection de Corruption (3 techniques différentes).

Le deuxième avantage concerne la mise en œuvre de l'AFT. Il est possible de ne modifier qu'une action spécifique pour changer la stratégie de FTM et adapter l'architecture à un changement de contexte. Prenons l'exemple du PBR. Nous avons utilisé précédemment une technique de type *getter/setter* dans le Chapitre 2 ce qui demande une instrumentation de l'application. Une mise à jour de l'application ne permet plus cette instrumentation. Nous avons juste besoin de changer les deux actions de capture et de mise à jour de l'état en utilisant une autre technique (Comme DMTCP ou *Diskless Checkpointing*).

Le troisième avantage concerne l'évolutivité de cette approche. Notre analyse

est représentative mais en aucun cas complète. D'autres catégories sont amenées à apparaître et d'autres choix de conception et d'implémentation peuvent les étendre. Il est donc possible de conserver les catégories génériques et les classes mères telles quelles et d'ajouter des sous-catégories et de faire évoluer la liste. Une nouvelle technique de contrôle d'intégrité a été mise au point, il est fort probable qu'elle nécessite une action d'encodage et de décodage des communications et qu'elle s'intègre facilement dans la catégorie Détection de Corruption.

Cependant cette approche n'est pas qu'avantageuse et présente un certain nombre d'inconvénients notamment au niveau de la multiplication du nombre d'unité d'exécution.

Le premier inconvénient se rapporte à la granularité de l'action. Des catégories peuvent contenir plusieurs actions intrinsèquement liées. Il est évident que la capture de l'état d'une application n'est pas gratuite et nécessite une mise à jour soit d'une réplique (PBR, Réplication Semi-Passive, *Recovery Block*), soit de la même application pour recommencer un calcul dans les mêmes conditions (TR). Or, en choisissant de n'implémenter qu'une action par unité d'exécution (nœud/processus/composant), nous augmentons leur nombre inutilement. Si nous analysons le fonctionnement du PBR, lors de la perte du Primaire, le Secondaire prend son rôle à l'aide d'une reconfiguration. Si nous n'implémentons pas l'action de capture d'état dans le même composant que la mise à jour, il faut entreprendre une substitution (remplacer le composant contenant l'action de mise-à-jour d'état par celui contenant l'action de capture d'état) en plus d'une reconnexion du Secondaire au Client.

Le second inconvénient concerne l'adaptation. Qui dit un plus grand nombre d'unité d'exécution, dit un plus grand nombre à substituer et donc nous augmentons le nombre d'étapes à valider hors-ligne et à effectuer en ligne. Cela peut avoir un impact négatif sur le temps d'indisponibilité lors d'une transition et augmente les ressources allouées à la validation d'une modification du graphe de composants hors-ligne. Ce que nous gagnons en coût de développement concernant la réutilisation de code est diminuée par le coût de la validation qui augmente.

Ces inconvénients viennent surtout de notre choix d'implémentation d'une action par nœud ROS. Nous augmentons le nombre d'unité d'exécution ce qui aggrave notre problème de ressource mais nous réalisons les FTM de manière plus générique. Il nous faut trouver une nouvelle façon de les concevoir en n'utilisant non plus un graphe de composants mais un graphe d'actions. De plus, le passage de la catégorie d'action à une classe C++ est simple à réaliser et nous pouvons étudier une conception orientée objets au lieu d'un conception orientée composants.

4 Synthèse

Ce troisième chapitre nous a permis de modifier la granularité de l'approche par composants substituables pour augmenter le niveau de généricité et de réutilisation de code lors de la conception de mécanismes de tolérance aux fautes. Nous avons pu exposer une nouvelle façon de modéliser nos mécanismes mais également de les décomposer. Nous sommes partis des actions de sûreté de fonctionnement réunies dans les composants BPA et nous les avons regroupés dans des classes génériques que nous avons spécifiées en fonction des besoins des FTM et de l'application.

Ce qu'il faut retenir finalement de ce chapitre est la modélisation des différents mécanismes sur des réseaux de Petri, qui a permis de modélisé les actions élémentaires mais également d'utiliser un langage formel, la nouvelle décomposition et hiérarchisation des éléments formant les FTM ainsi que leur implémentation sur des nœuds ROS. La nouvelle granularité et l'utilisation d'acteur est très avantageuse au niveau de la généricité des éléments composants le FTM. Ainsi, nous ne faisons plus abstraction des actions communes entre les mécanismes. Nous concevons nos FTM à partir d'éléments (ici des acteurs) issus de classes génériques et nous ne concevons plus nos éléments à partir des FTM eux-mêmes. Cette inversion du point de vue a permis d'augmenter, notamment, la réutilisation d'éléments et donc de code pour implémenter nos FTM. Quant à l'implémentation sur ROS, elle aggrave le problème du nombre de composants actifs dans le système et donc l'utilisation de ressources de calcul.

Nous étudierons donc dans le prochain chapitre une autre façon d'implémenter nos FTM en ne projetant plus un acteur sur un nœud ROS mais tous les acteurs sur des *threads* appartenant à un unique nœud. Pour ce faire, nous utiliserons la notion de *plug-in* et nous analyserons comment manipuler ces *plug-in*, comment les ordonnancer et comment adapter et composer les FTM. Ce chapitre se concentrera non pas comme celui-ci sur de la conception mais plus sur de la technique d'implémentation.

Nous allons détailler dans le Chapitre 4 comment optimiser les ressources de calcul utilisées en diminuant le nombre de nœud composant nos FTM à un seul. Ce chapitre se divisera en quatre sections. La première se concentrera sur la conception des FTM sous forme de graphe d'actions. La deuxième présentera l'implémentation des FTM sur ROS en mettant en avant et en détaillant l'utilisation de *plugins*, à savoir des objets dynamiques. La troisième concernera l'adaptation des *plugins* dans ROS. La quatrième section se focalisera sur la validation de cette approche en reprenant les méthodes utilisées dans le chapitre 2.

Chapitre 4

Approche par actions ordonnançables

Sommaire

Introduction

Dans le Chapitre 2, nous avons présenté une architecture logicielle nous permettant de concevoir des mécanismes de tolérance aux fautes (FTM) sous la forme d'un graphe de composants P-BPA (*Protocol - Before Proceed After*). En décomposant nos FTM et en les projetant sur ce graphe, nous avons pu facilement les adapter et les composer en faisant des mises à jour différentielles. Ainsi nous pouvons passer d'un mécanisme à un autre en ne substituant que les composants propres à tel ou tel FTM.

Dans le Chapitre 3, nous avons effectué une analyse de plusieurs FTM pour identifier des catégories d'action de SdF. Ces actions de SdF sont des actions élémentaires ayant pour but d'assurer les propriétés de sûreté de fonctionnement en présence de fautes. Cette analyse nous a permis de créer un schéma de conception pour une adaptation plus fine des FTM.

De cette analyse, nous avons pu conclure que l'approche P-BPA ne permet pas une adaptation à grain fin. En effet, les composants contiennent souvent plusieurs actions de SdF. Par exemple, il existe plusieurs façons de faire de la sélection de donnée (comparaison, moyenne, vote) et nous ne pouvons pas substituer une méthode par une autre sans modifier tout le composant les contenant. De plus, chaque composant est projeté à l'exécution sur une unité d'exécution indépendante (processus UNIX, nœud ROS, tâche OSEK/VDX) ce qui peut rapidement dégrader les performances sur des systèmes embarqués à ressources limitées.

Dans ce chapitre, nous allons donc présenter une seconde architecture logicielle nous permettant de concevoir des mécanismes non plus sous la forme d'un graphe de composants mais d'un graphe d'action. Ainsi, nous exploitons l'analyse fine des FTM du chapitre précédent pour projeter à l'exécution nos FTM sur des entités exécutables légères (comme des *threads*). En outre, nous validons cette nouvelle architecture en l'implémentant sur ROS, notamment à l'aide de la notion de *plugin*.

En bref les avantages de la méthode proposée dans ce chapitre sont : **(1)** la performance en terme de consommation de ressources et **(2)** le niveau très fin de granularité de la mise à jour au niveau d'un objet.

La première section de ce chapitre se concentrera sur la conception des FTM en utilisant les résultats du Chapitre 3. Nous listerons et détaillerons les éléments constituant nos FTM mono-processus. Nous développerons également les méthodes employées pour adapter et composer nos FTM.

La deuxième section présentera l'implémentation des FTM sur ROS en mettant en avant et en détaillant l'utilisation de *plugins*, à savoir des objets dynamiques. Ainsi nous expliquerons les différentes caractéristiques des *plugins* pour mettre en œuvre nos FTM.

La troisième section concernera l'adaptation des *plugins* dans ROS. Notre but va être d'implémenter les méthodes pour l'adaptation et la composition des FTM décrites dans la première section et ainsi mettre en œuvre l'AFT.

La quatrième section se focalisera sur la validation de cette approche en reprenant les méthodes utilisées dans le chapitre 2.

1 Nouvelle méthode de conception

Dans cette section, nous allons construire un FTM adaptable à grain fin et à faible coût en ressource de calcul. Ici, un FTM est composé d'un ensemble d'actions de SdF permettant d'assurer la fiabilité et la continuité d'un service nominal ou le passage à un service dégradé sûr de fonctionnement. Vis-à-vis de l'approche présentée dans le Chapitre 2, nous passons d'un FTM constitué de composants à un FTM constitué d'actions.

Dans cette approche, le FTM est un conteneur d'objets C++. Les actions de SdF sont instanciées dans les méthodes d'une partie de ces objets. Nous appelons les objets contenant les actions de SdF des objets dynamiques. Ces objets sont issus des catégories et sous-catégories définies dans le Chapitre 3. En plus des objets dynamiques, le FTM contient trois objets statiques permettant la manipulation des objets (Gestionnaire d'Objets), l'ordonnancement des méthodes (l'Ordonnanceur) et enfin, la coordination des transitions de type reconfiguration, adaptation et composition à l'intérieur du FTM (l'*Adaptive Entity*).

Le but de cette approche est de réduire la consommation en ressource de l'approche par composants substituables. Nous voulons réduire le nombre d'unité d'exécution en ne modifiant plus un graphe de composants mais un graphe d'actions interne à un unique composant. De plus, nous réduisons le nombre de communication en faisant interagir les méthodes des objets par variables partagées.

1.1 Du composant à l'objet

Nous rappelons que pour sélectionner un mécanisme de tolérance aux fautes, celui doit répondre aux différentes hypothèses induites du modèle de faute (FT), des caractéristiques de l'application (AC) et des besoins en ressource (RS).

Dans l'approche précédente, nous choisissions un FTM en fonction de ces trois hypothèses. Une fois le FTM sélectionné, nous le projetions sur un ensemble de trois composants, le *Before*, le *Proceed* et l'*After*, contenant les actions de SdF et nous ajoutions un Protocol servant d'interface de communication avec la machine contenant le Client. Un FTM est donc constitué du graphe de composant appelé P-BPA. Finalement nous implémentions nos composants sur des unités d'exécution (processus Unix, nœuds ROS).

Dans cette nouvelle approche, un FTM est un ensemble de méthodes d'objets coordonnées par un ordonnanceur interne pour réaliser un algorithme. Comme l'illustre la figure 4.1, un FTM est donc un graphe d'actions orchestré dans une seule entité exécutable. Ainsi ce n'est plus le graphe de composant qui est amené à évoluer avec le système mais le graphe d'actions. L'adaptation ne consistera donc plus à modifier les composants constituant le FTM mais à modifier l'ordre d'exécution des différentes méthodes qui le composent.

Un objet peut contenir plusieurs actions de SdF qui peuvent être actives ou dormantes lors de son instanciation. Prenons toujours l'exemple du PBR et du TR ainsi que de la Gestion d'État. Les objets contenant les actions de capture et de

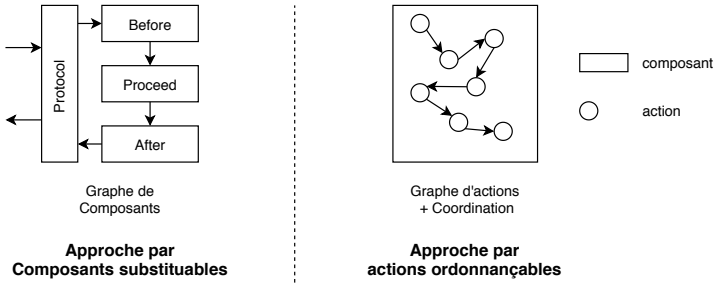


FIGURE 4.1 – Comparaison de la conception d'un FTM entre les deux approches

mise à jour de l'état sont issues de la classe **GestEtat** (Voir figure 3.5). Dans le cas du PBR, l'action "capture" est active dans le Primaire mais dormante dans le Secondaire et inversement pour l'action mise à jour. Dans le cas du TR, les deux actions sont actives. Une action dormante peut être activée si le FTM en a besoin, p.ex. l'action capture du Secondaire devient active en cas de crash du Primaire puisque le Secondaire devient le nouveau Primaire et un Secondaire est relancé.

L'approche par composants substituables repose sur une activation et coordination des composants du P-BPA par messages. Le composant *Before* s'exécute puis envoie un message au composant *Proceed* avec les informations dont il a besoin et ainsi de suite jusqu'au renvoi de la réponse au Client. Dans notre nouvelle approche, l'activation du FTM se fait également par message, mais la coordination des actions est orchestrée par un ordonnanceur interne.

Nous remarquons donc que nos FTM vont être composés de deux catégories d'objets qui vont avoir des rôles différents :

1. **Les objets dynamiques :** Ils contiennent les actions de SdF ;
   - Gestion d'État ;
   - Sélection de données ;
   - Synchronisation.
   - ...
2. **Les objets statiques :** Ils permettent la manipulation des objets dynamiques et l'ordonnancement de l'algorithme du FTM ;
   - **Ordonnanceur :** Il orchestre les objets pour réaliser l'algorithme du FTM à l'aide d'une table d'orchestration qui définit la séquence d'appel des méthodes ;
   - **Gestionnaire d'objet :** Il manipule le graphe d'actions en ajoutant ou en supprimant un élément du graphe. Il construit la table d'orchestration qui est récupérée par l'Ordonnanceur.
   - ***Adaptive Entity* :** Elle coordonne le fonctionnement interne du FTM et définit l'état désiré du graphe d'actions lors d'une transition. Elle va arrêter l'exécution du graphe d'action, formater et transmettre au Gestionnaire d'Objets les manipulations à effectuer.

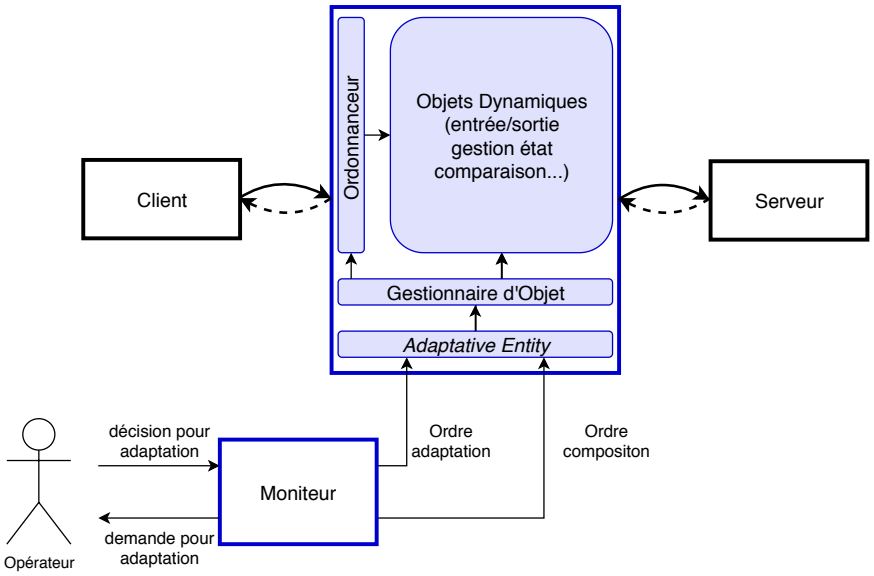


FIGURE 4.2 – Architecture complète d'un FTM à graphe d'actions

La figure 4.2 présente la structure complète du FTM avec les différents éléments qui le composent. Nous reprenons des éléments présents dans l'approche par composants substituables, à savoir le couple Opérateur/Moniteur permettant d'observer l'évolution du système et de donner un ordre de transition (adaptation & composition). Nous avons déporté les fonctionnalités du contrôle de l'exécution et de la manipulation du graphe d'actions de l'*Adaptation Engine* dans l'*Adaptive Entity* et dans le Gestionnaire d'Objets. C'est un choix de conception pour faciliter les modifications du graphe d'actions qui sont internes au FTM.

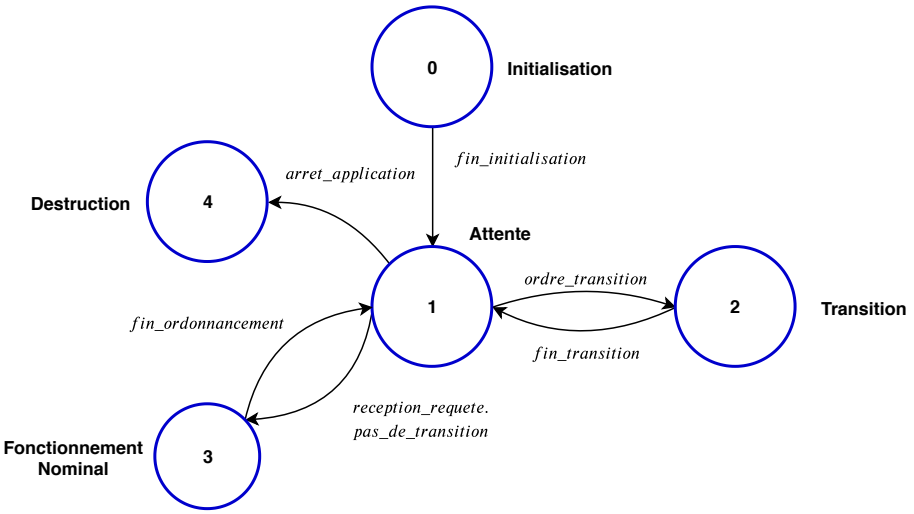


FIGURE 4.3 – Graphe de comportement global d'un FTM

La figure 4.3 représente le fonctionnement interne d'un FTM. Le FTM est un conteneur dans lequel nous instancions les objets statiques et qui exécute la machine d'état que nous allons détailler.

Lors de la phase d'initialisation (État 0), les objets statiques, à savoir l'Ordonnanceur, le Gestionnaire d'Objets et l'*Adaptive Entity*, sont instanciés et initialisé. Le Gestionnaire d'objet instancie à son tour les objets dynamiques requis pour le FTM initial. Il récupère les méthodes des différents objets et construit l'algorithme de fonctionnement en remplissant la table d'orchestration. L'Ordonnanceur récupère la table puis le FTM entre en phase d'attente d'une requête ou d'un ordre de transition (État 1).

S'il n'y a pas d'ordre de transition et si le FTM reçoit un message de la part du Client, il entre en fonctionnement nominal (État 3). Sur réception d'une requête, l'Ordonnanceur exécute l'algorithme du FTM en parcourant la table d'orchestration. Une fois la table parcourue, le FTM repasse en phase d'attente.

Si il y a une demande de transition, l'*Adaptive Entity* coordonne les transitions à effectuer (État 2). L'*Adaptive Entity* empêche le FTM d'entrer en fonctionnement nominal et sélectionne la liste des transformations à effectuer (objets à supprimer, à instancier, ordre d'appel des méthodes). Le Gestionnaire d'Objets récupère cette liste, effectue les transformations et construit la table d'orchestration. Enfin, l'Ordonnanceur récupère la table d'orchestration puis le FTM repasse en phase d'attente.

Finalement, lors de l'arrêt de l'application par l'utilisateur, nous supprimons le FTM (État 4) en appelant les destructeurs des différents objets (statiques et dynamiques) pour libérer les espaces mémoire qui ont pu être alloués.

Dans la suite de cette section 1, nous allons détailler le fonctionnement de ces trois objets statiques en commençant par le Gestionnaire d'Objets. Nous expliquerons comment nous avons instancié nos actions dans des méthodes et comment nous créons, instancions et initialisons les objets dynamiques contenant ces méthodes.

1.2 Conceptions des objets dynamiques

Intéressons nous maintenant à la conception des objets dynamiques avant d'expliquer leur manipulation par les objets statiques. Un objet dynamique est un objet issu du langage C++ qui contient au moins une méthode implémentant une action de SdF. À partir de ces objets, nous construisons le graphe d'actions.

Pour concevoir les objets dynamiques, nous reprenons nos catégories et sous-catégories du Chapitre 3 et nous les projetons sur des classes d'objets C++. La figure 4.4 représente un exemple de projection d'une action de capture et de mise à jour de l'état de l'application dans le cas d'un TR. La catégorie Gestion d'État est projetée sur une classe mère abstraite GestEtat. Nous choisissons ensuite une technique type *Getter/Setter* qui est projetée sur la classe fille GestEtat_GetSet et qui hérite de la classe mère. Enfin, nous choisissons une transmission de l'état par variable partagée ce qui nous donne la classe GestEtat_GetSet_Var qui contient deux méthodes, une pour la capture et une pour la mise à jour de l'état. Nous

appellerons les méthodes implémentant les actions de SdF les méthodes d'action.

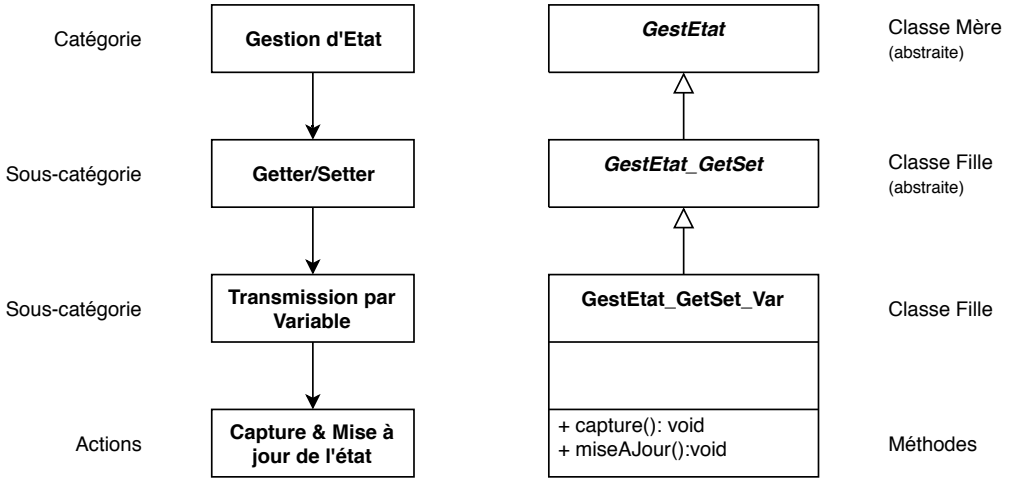


FIGURE 4.4 – Projection des catégories sur des classes d'objets C++

En plus de ces méthodes d'action, nous devons analyser les aspects clefs pour la manipulation et l'appel de ces méthodes en vue de la création, de la manipulation et de l'ordonnancement du graphe d'actions.

Les actions de SdF vont de la synchronisation par message à la capture de l'état d'une application tout en passant par la sélection de données ou encore la détection de corruption. Les façons dont nous pouvons les implémenter sont multiples avec des signatures de méthode et des paramètres différents. Nous souhaitons uniformiser ces méthodes pour les rendre génériques. Il est nécessaire que toutes les méthodes appelées aient la même signature et les mêmes paramètres pour avoir un ordonnancement lui aussi générique. Nous aborderons, dans la section 3 de ce chapitre, la solution que nous avons mise en place pour avoir chaque méthode d'action avec une signature et des paramètres uniformes.

Enfin, il est essentiel pour pouvoir construire le graphe d'actions de récupérer les méthodes d'actions en vue de les stocker dans la table d'orchestration. Il nous faut donc, pour chaque objet dynamique un accesseur (ou *getter*) pour appeler les méthodes d'action. Cet accesseur sera appelé par le Gestionnaire d'Objets pour construire la table d'orchestration.

Pour résumer, un objet dynamique doit contenir au moins une méthode d'action qui implémente l'action de SdF. De plus cette méthode doit avoir une signature et des paramètres uniformes pour être générique. Il faut également un accesseur à cette méthode pour construire la table d'orchestration.

Nous allons maintenant nous intéresser au Gestionnaire d'Objets dont le rôle est essentiel dans notre approche. Le Gestionnaire va construire le graphe d'actions en instanciant les objets dynamiques et qui va construire la table d'orchestration avant quelle ne soit récupérée par l'Ordonnanceur.

1.3 Un Gestionnaire pour les gouverner tous

Le Gestionnaire d'Objets va permettre de créer le graphe d'actions. Son rôle est de créer les objets dynamiques, de les initialiser et de récupérer les méthodes contenant les actions de SdF pour les transmettre ensuite à l'Ordonnanceur. Si l'Ordonnanceur est le chef d'orchestre du FTM, le Gestionnaire d'Objets est le compositeur du graphe d'actions.

1.3.1 Identification des besoins

À l'instar de l'*Adaptation Engine* de l'approche par composants substituables, le Gestionnaire d'Objets doit connaître tous les objets qui peuvent être instanciés dans un FTM. Pour que le FTM soit fonctionnel sans prendre en compte pour l'instant l'AFT, il doit comporter au moins quatre méthodes :

1. **Méthode d'instanciation d'objet :** Création et initialisation de n'importe quel nombre d'objets dynamiques requis pour le FTM sélectionné ;
2. **Méthode de suppression d'objet :** Suppression des objets dynamiques lors de l'arrêt de l'application par l'utilisateur ;
3. **Méthode de construction de la table :** Récupération des méthodes des objets et construction de la table d'orchestration réalisant l'algorithme du FTM ;
4. **Méthode d'exportation de la table :** Récupération de la table par l'Ordonnanceur.

C'est bien le Gestionnaire d'objet qui crée la stratégie de tolérance aux fautes en créant la table d'orchestration. Cette table doit pouvoir contenir pour chaque méthode à appeler la méthode suivante et ainsi de suite jusqu'à ce que le graphe d'action soit complet.

1.3.2 Fonctionnement et analyse du Gestionnaire

Le Gestionnaire d'Objets est créé pendant la phase d'initialisation du FTM. Lors de l'initialisation du FTM, le Gestionnaire construit le graphe d'actions en lançant les objets contenant les méthodes d'actions, récupère ces méthodes puis construit la table d'orchestration.

Nous allons maintenant expliquer la création du graphe d'actions à travers l'exemple du TR. Nous partons d'un FTM vide, sans les objets statiques et nous décrivons les étapes de la phase d'initialisation qui concerne principalement le Gestionnaire d'Objets :

1. L'Ordonnanceur et le Gestionnaire sont instanciés et initialisés. Création et l'initialisation des diverses variables (table d'orchestration, variables internes...) ;
2. Le Gestionnaire ensuite instancie les objets du TR du graphe illustré sur la figure 4.5 :

- Objet de Gestion d'État pour la capture et la mise à jour de l'état de l'application avec transmission de l'état par variable partagée ;
- Objet de Sélection de Données pour comparer les réponses des différents exécutions successives ;
- Objets de Synchronisation, deux communications asynchrones pour communiquer avec le *Proxy* et une communication synchrone pour le Serveur ;
- Objet de stockage de données ;

3. Le Gestionnaire d'Objets récupère les méthodes d'actions des six objets ;
4. Le Gestionnaire d'Objets crée la table d'orchestration du TR ;
5. L'Ordonnanceur récupère la table auprès du Gestionnaire.
6. Le FTM passe en phase d'attente de réception de requêtes du Client.

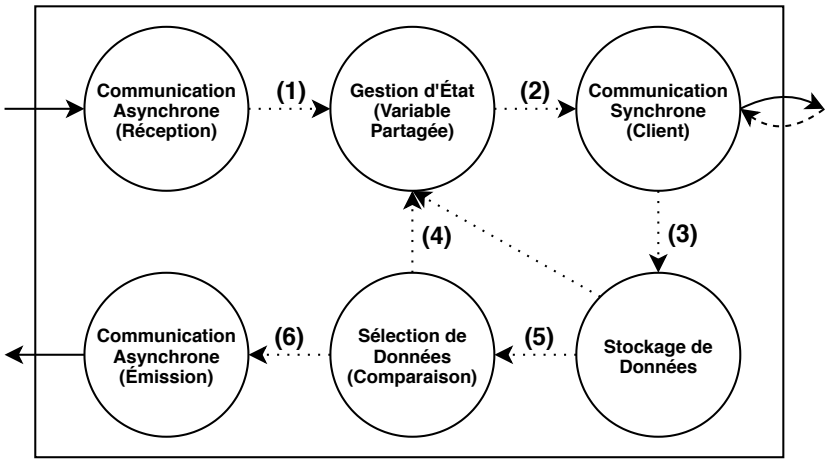


FIGURE 4.5 – Graphe d'actions du TR

La figure 4.5 représente les objets qui forment le graphe d'action ainsi que l'ordre d'appel des actions. Il faut que la table d'orchestration contienne les méthodes d'action à appeler mais également l'ordre d'exécution de ces méthodes.

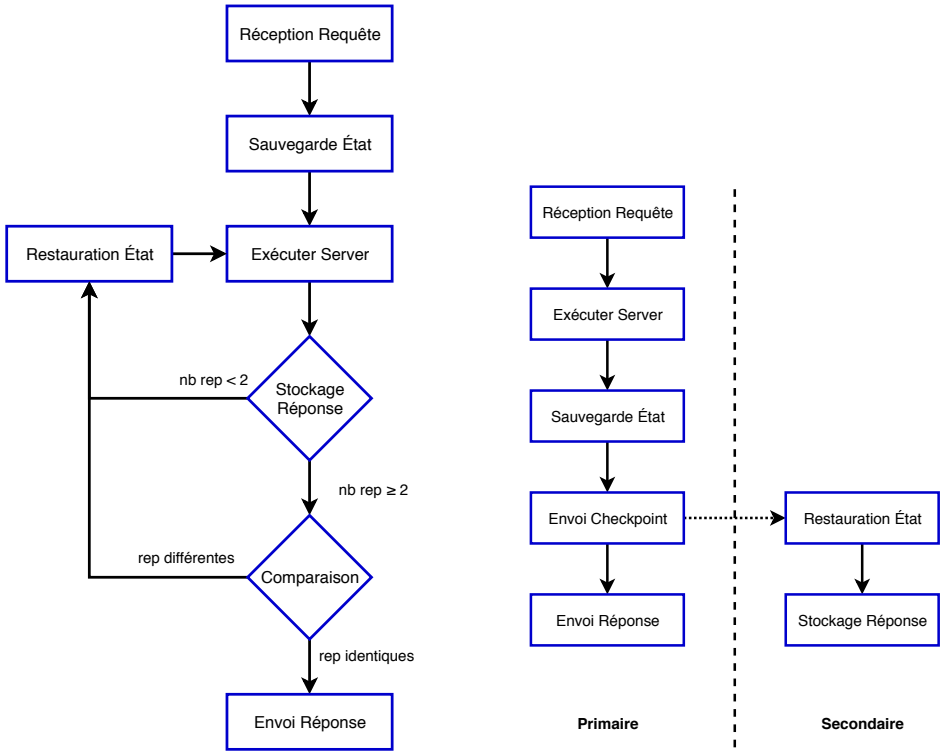
La première remarque que nous pouvons faire est que nous avons décidé de conserver le *Proxy* lorsque le Client et le Serveur communiquent par message synchrone. Ce composant permet une transformation simple de la communication synchrone en deux communications asynchrones. De plus, il ajoute un identifiant à la requête pour assurer le *Exactly-once-semantics* [Griffin 1990].

La seconde remarque que nous pouvons écrire concerne la relation entre le Gestionnaire d'Objets et l'Ordonnanceur. Ces deux objets statiques se transmettent la table d'orchestration. Il faut que la table d'orchestration soit lisible par l'Ordonnanceur. C'est donc l'Ordonnanceur qui va imposer des règles de construction de la table d'orchestration.

1.4 Un Ordonnanceur pour les appeler et dans une table les lier

1.4.1 Identification des besoins de l'Ordonnanceur

Le principe de l'Ordonnanceur est d'appeler les méthodes d'action dans un certains ordre pour parcourir le graphe d'actions. Il faut donc qu'il puisse appeler les méthodes d'action des objets et qu'il sache dans quel ordre les appeler. Même si c'est le Gestionnaire d'Objets qui définit l'ordre, il faut que l'Ordonnanceur puisse suivre cet ordre. Nous avons donc étudié l'exécution des algorithmes de deux FTM pour définir les séquences d'exécution des actions.



(a) Algorithme du TR (b) Algorithme du PBR

FIGURE 4.6 – Exemples d'algorithme de deux FTM

La figure 4.6 représente les algorigrammes illustrant le fonctionnement du TR et du PBR. Lorsqu'une action n'a qu'une action suivante possible, elle est représentée par un rectangle. Par exemple, dans le cas du TR, la seule action possible après une capture de l'état est l'exécution de l'action du Serveur. Lorsqu'il faut choisir entre deux actions possibles, l'action est représentée par un losange régulier. Toujours dans le cas du TR, après une comparaison, si les deux réponses sont identiques, l'action suivante est l'envoi de la réponse. Si les deux réponses sont différentes, l'action suivante est la mise à jour de l'état.

Nous pouvons remarquer que le fonctionnement nominal des FTM présentés est séquentiel. Bien qu'il semble y avoir du parallélisme pour le PBR, le Primaire et

le Secondaire ont chacun un ordonnanceur qui leur est propre et qui exécutera sa propre séquence d'actions.

Dans le cas d'une action avec deux actions suivantes possibles, l'Ordonnanceur va devoir sélectionner l'action suivante en fonction du résultat de la méthode d'action qu'il vient d'appeler. Nous définissons deux résultats possibles lors de l'exécution d'une action, un résultat "Vrai" et un résultat "Faux". Par convention, lorsqu'une méthode d'action est appelée et qu'une action suivante est possible, la méthode d'action appelée aura toujours un résultat Vrai.

De plus, si l'action ne s'est pas correctement exécutée, celle-ci renverra un code d'erreur exprimant le problème rencontré pour pouvoir effectuer un diagnostique et une opération de maintenance.

Finalement, l'Ordonnanceur doit être capable d'appeler les méthodes d'action et en fonction du résultat de celles-ci (Vrai ou Faux), choisir la méthode suivante à appeler. De plus, il doit prendre en compte le traitement d'erreur dans le cas où la méthode appelée ne se serait pas exécutée correctement.

Pour l'instant nous avons pris des exemples de FTM avec une ou deux actions suivantes possibles. Nous devons définir une solution qui soit extensible à N actions suivantes.

1.4.2 De l'algorigramme à la table d'orchestration

Dans un premier temps, pour répondre aux besoins de l'Ordonnanceur, nous avons projeter ces algorigrammes dans trois tableaux. Ce tableau contient quatre colonnes, l'action de SdF à exécuter, l'action suivante si le résultat est Vrai, l'action suivante si le résultat est Faux et enfin une colonne de traitement d'erreur. Dans le cas où une seule action suivante est possible nous choisissons, par convention, d'associer au résultat Vrai l'action suivante et au résultat Faux la valeur NULL. De plus, nous avons ajouté une valeur INIT pour définir la première méthode à appeler et END pour signifier à l'Ordonnanceur la fin de l'exécution de l'algorithme.

Action de SdF	Action Suivante si Vrai	Action Suivante si Faux	Traitement d'erreur
INIT	Réception de requête	NULL	NULL
Réception Requête	Capture État	NULL	Err1
Capture État	Exécuter Serveur	NULL	Err2
Exécuter Serveur	Stockage Réponse	NULL	Err3
Stockage Réponse [1]	Comparaison	Mise à jour État	Err4
Mise à jour État	Exécuter Serveur	NULL	Err5
Comparaison	Envoi Réponse	Mise à jour État	Err6
Envoi Réponse	END	NULL	Err7

Table 4.1 – Table d'orchestration du TR

Le tableau 4.1 représente l'ordre d'exécution des actions pour le fonctionnement du TR [2]. Notre Ordonnanceur parcours le tableau et appelle les méthodes des objets concernés. Finalement ce tableau décrit parfaitement la figure 4.5 de la section 1.3.2 de ce chapitre à la page 115.

Le principe de ce tableau est simple et permet de traduire un algorigramme tels que ceux présentés sur la figure 4.6.

1. Toute action (colonne 1) peut être suivie par une autre action (colonne 2) de façon séquentielle
2. Dans le cas d'une condition, deux actions peuvent être définies en fonction du résultat booléen de la condition (colonne 2 pour Vrai, colonne 3 pour Faux).
3. Dans le cas d'une erreur lors de l'exécution de l'action, une action peut être définie pour mettre en œuvre une politique de gestion d'erreur(colonne 4).

Regardons maintenant les deux tableaux4.2 et 4.3 pour le Primaire et le Secondaire. Ce besoin d'avoir un tableau par machine est commun à tous les mécanismes ayant plus d'une réplique (mécanisme duplexe, TMR...)

Action de SdF	Action Suivante si Vrai	Action Suivante si Faux	Traitement d'erreur
INIT	Réception Requête	NULL	NULL
Réception Requête	Exécuter Serveur	NULL	Err1
Exécuter Serveur	Capture État	NULL	Err3
Capture État	Envoi Réponse	NULL	Err2
Envoi Réponse	END	NULL	Err7

Table 4.2 – Table d'orchestration du Primaire du PBR

Action de SdF	Action Suivante si Vrai	Action Suivante si Faux	Traitement d'erreur
INIT	Mise à jour État	NULL	NULL
Mise à jour État	Stockage Réponse	NULL	Err5
Stockage Réponse	END	NULL	Err4

Table 4.3 – Table d'orchestration du Secondaire du PBR

1. Lors de la première exécution, la méthode de stockage de réponse sauvegarde celle-ci. Comme il n'y a qu'une réponse, nous restaurons l'état initial de l'application pour l'exécuter une seconde fois. Lorsque la méthode de stockage a sauvegardé plusieurs réponses, nous les comparons.

2. Dans cet exemple, lorsque les réponses sont toujours différentes, l'algorithme boucle indéfiniment. La technique du TR tolère en effet les fautes transitoires du matériel, Pour des fautes permanentes, un test d'arrêt au bout de N réponses doit être ajouté.

Si nous examinons l'algorigramme du PBR, chaque machine a sa propre table d'orchestration comme l'illustrent les tableaux 4.2 & 4.3. L'avantage d'utiliser ces tableaux est leur facilité de modification en vue d'une transition. Dans le cas du PBR, si le Primaire crashe, il est simple de transformer le Secondaire en Primaire. En effet, il suffit de reconstruire le tableau pour qu'il corresponde à celui du Primaire. Le Secondaire a déjà tous les objets présents et les méthodes qui lui manquent (ici les méthodes d'action Capture État et Réception Requête) sont dormantes et peuvent être activée par le Gestionnaire d'Objets lors de leur récupération.

Cependant, nous ne souhaitons pas mettre en œuvre ce tableau directement car cela implique de manipuler les méthodes d'action. Il est plus aisé d'affecter à chaque méthode une clef (type string) pour sélectionner la méthode suivante à appeler. Nous allons donc transformer ce tableau en un dictionnaire.

1.4.3 Mise en œuvre de la table d'orchestration

Pour que l'Ordonnanceur puisse mettre en œuvre cette table, nous avons décidé d'imiter le fonctionnement d'un dictionnaire. Un dictionnaire fonctionne de manière simple. Pour une clef, nous associons une définition. Ce système va nous permettre de recherche facilement la méthode d'action que l'on souhaite appeler. Par exemple, pour la méthode d'action de comparaison des réponses du TR, nous lui donnons la clef Comparaison. La définition contient un pointeur de fonction sur la méthode d'action à appeler, la clef si le résultat est Vrai (ici Envoi Réponse), la clef si le résultat est Faux (ici Mise à jour État) et la clef du traitement d'erreur (ici Err6). Chaque ligne de la table est enregistrée de la manière suivante :

$$Clef_{Action} \rightarrow \{^{*}Méthode,\ Clef_{Vrai},\ Clef_{Faux},\ Clef_{Err}\}$$

Pour appeler une méthode d'action, l'Ordonnanceur va lire la Clef associée à l'action, appeler la méthode d'action puis en fonction du résultat (Vrai ou Faux) va choisir la clef de l'action suivante. Si la méthode d'action ne s'est pas exécutée correctement, elle renvoie un code erreur et c'est au concepteur de définir les mesures à prendre dans un tel cas.

Prenons l'exemple de l'action de stockage de réponse dans le TR. Voici à quoi ressemble une ligne de la table d'orchestration :

$$Clef_{Stock} \rightarrow \{^{*}Stockage,\ Clef_{Comp},\ Clef_{MaJ},\ Clef_{Err}\}$$

l'Ordonnanceur regarde la clef associé à l'action `Stockage Réponse` qui est $Clef_{stock}$. Il appelle la méthode d'action au travers d'un pointeur de fonction. S'il n'y a qu'une réponse, la méthode renvoie Faux et l'Ordonnanceur choisit la clef $Clef_{MaJ}$ puis appelle la méthode `Mise à Jour État`. S'il y a plusieurs réponses, l'action renvoie Vrai et l'Ordonnanceur choisi la clef Key_{Comp} puis appelle la méthode `Comparaison`.

	INIT	⇒	{SauvEtat}			
(1)	SauvEtat	⇒	{*getState(),	SendReq,	NULL,	ERR}
(2)	SendReq	⇒	{*sendReq(),	Stock,	NULL,	ERR}
(3)	Stock	⇒	{*stockData(),	RestEtat,	Comp,	ERR}
(4)	RestEtat	⇒	{*setState(),	Stock,	NULL,	ERR}
(5)	Comp	⇒	{*setState(),	RestEtat,	SendAns,	ERR}
(6)	SendAns	⇒	{*sendAns(),	END,	NULL,	ERR}
(7)	END	⇒	{NULL,	NULL,	NULL,	NULL}

TABLE 4.4 – Table d'orchestration du TR

La table 4.4 représente la projection du tableau 4.1 sur la table d'orchestration. Les Clefs sont représentées en orange et les pointeurs de fonction en bleu. Maintenant que nous avons notre table d'orchestration, regardons le fonctionnement nominal du FTM.

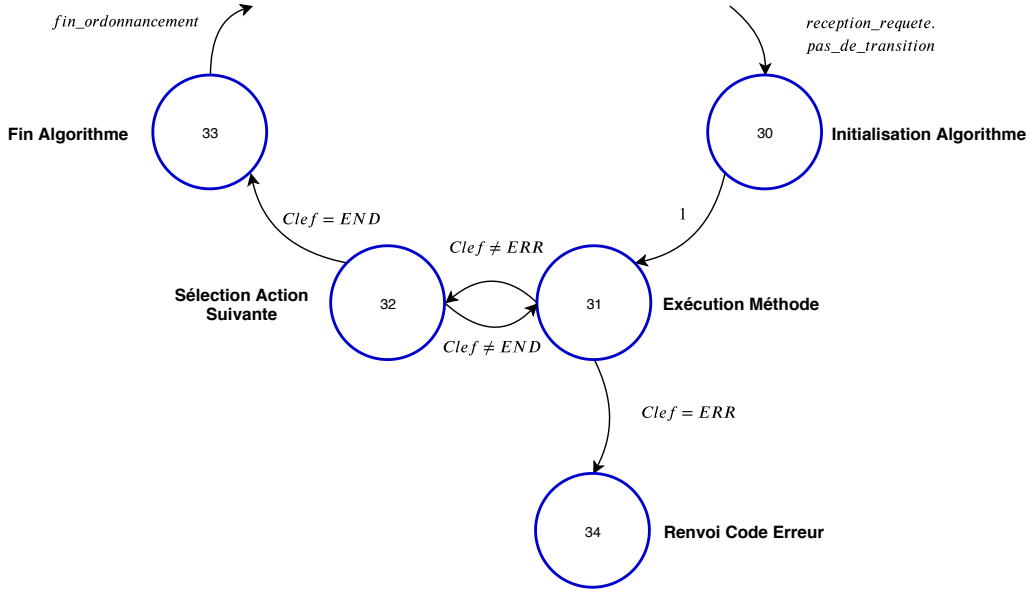


FIGURE 4.7 – Sous-Machine d'état du fonctionnement nominal (État 3)

La figure 4.7 détaille les étapes du fonctionnement nominal du FTM qui correspond à l'État 3 de la machine d'état 4.3 à la page 111. L'Ordonnanceur exécute l'algorithme sur réception d'un message de la part du Client en suivant cette machine d'état. L'initialisation (État 31) consiste à choisir la première méthode d'action à appeler.À la clef INIT est associé la clef de la première méthode d'action à appeler. Une fois la première clef sélectionnée, L'Ordonnanceur va appeler toutes les méthodes d'actions. Si toutes les méthodes s'exécutent sans erreur (État 32 & 33), l'Ordonnanceur lit la table jusqu'à la clef END qui signifie que l'algorithme est terminé et que le FTM peut repasser en phase d'attente (État 1). Si une mé-

thode ne s'exécute pas correctement, elle renvoie une code d'erreur et l'algorithme est interrompu (État 35).

Nous avons étudié jusqu'à présent le fonctionnement nominal du FTM et la conception des objets dynamiques, du Gestionnaire d'Objets et de l'Ordonnanceur. Ces objets ne sont pas suffisant pour mettre en œuvre l'AFT. Nous allons maintenant compléter le Gestionnaire d'objet pour y intégrer des méthodes de manipulation du graphe d'actions et ajouter le cœur de l'AFT pour les FTM, l'*Adaptive Entity*.

1.5 Mise en œuvre de l'AFT

Nous rappelons que l'AFT consiste en l'adaptation des FTM pour garantir la tolérance aux fautes d'une application en dépit de perturbation qui peuvent être causées par des changements brutaux imprévus (P. ex. perte immédiate d'un des 3 capteurs de recul invalidant un mécanisme comparant les trois valeurs reçues) ou encore par des changements prévus à long terme (changement de version de l'application). Nous devons donc être capable de modifier, dans notre cas, le graphe d'action lorsqu'un changement de contexte applicatif est détecté.

Dans l'approche par composants substituables, nous avons deux manières d'adapter les mécanismes. Soit nous faisons une modification différentielle du graphe de composant (Remplacement d'un ou plusieurs composants) soit nous composons nos FTM pour répondre à un modèle de faute complexe sans avoir à développer un mécanisme complet (Substitution du *Proceed* par un ensemble P-BPA).

Dans l'approche par objets ordonnançables, la composition et l'adaptation ne sont plus si différentes. En effet, il n'y a plus de problème de manipulation des communications internes au FTM qui nous empêchait d'intégrer facilement un composant entre deux autres déjà présents. Il suffit de supprimer et d'ajouter les bons objets et de modifier la table d'orchestration.

La figure 4.8 illustre la sous-machine d'état du FTM lors d'une transition correspondant à l'État 2 de la machine d'état 4.3 à la page 111. Lorsque le Moniteur détecte un changement dans le contexte applicatif, il demande à l'Opérateur la validation du besoin d'une transition. Après validation, il donne l'ordre à l'*Adaptive Entity* de modifier le FTM en place. L'*Adaptive Entity* empêche l'exécution du fonctionnement nominal (État 21), formate la liste d'objets à instancier avec l'ordre d'exécution (Etat 22). Le Gestionnaire récupère la liste et effectue les transformation du graphe d'actions. S'il s'agit d'une reconfiguration (P.ex. Perte du Primaire du PBR qui est remplacé par le Secondaire), l'ordre n'est pas donné par le Moniteur mais par le composant de Recouvrement.

Nous pouvons remarquer qu'en fonction des types de modification certaines méthodes ne sont pas prises en compte. Pour expliquer ce cas, nous partons d'une application Client/Serveur avec un PBR rattaché au Serveur et sa réplique.

Reconfiguration : Lors du crash du Primaire dans le cas d'un PBR, il suffit de modifier la table d'orchestration du Secondaire. Tous les objets requis sont déjà présents dans le Secondaire et il suffit d'activer les méthodes nécessaires en recons-

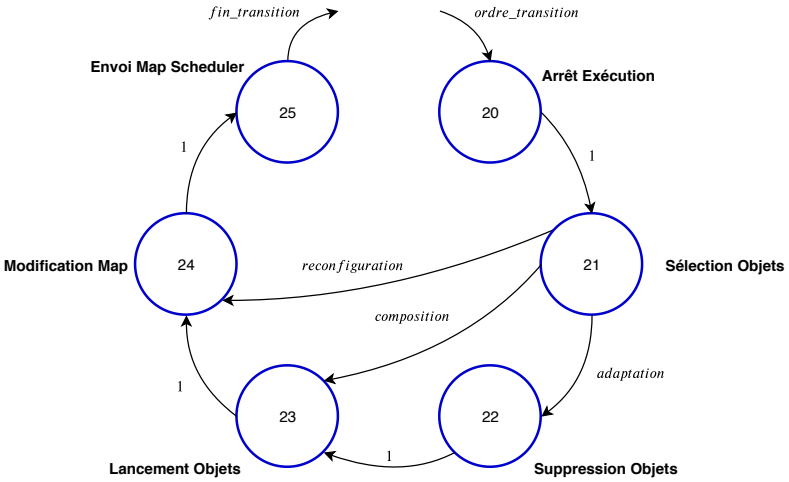


FIGURE 4.8 – Sous-Machine d'état lors d'une demande de transition (État 2)

truisant la table. L'objet Gestion d'État possédant les deux méthodes de sauvegarde et de restauration, il n'y a pas besoin d'instancier à nouveau l'objet.

Composition : Lors de l'apparition de fautes transitoires en valeur en plus de fautes matérielles par crash, nous devons ajouter au PBR un TR. Pour ce faire, il suffit d'ajouter l'objet Sélection de Donnée pour comparer les réponses des exécutions successives et de modifier à nouveau la table.

Adaptation : Lors de la diminution de la bande passante entre les deux ECU assurant la redondance, nous devons passer du PBR à un LFR. Ici, il faut supprimer l'objet Gestion d'État et ajouter les objets de synchronisation puis reconstruire à nouveau la table.

L'avantage de l'approche par actions ordonnançables par rapport à l'approche par composants substituables est double : **(1)** Il n'y a plus de communications entre composants ce qui rend la manipulation du graphe du FTM plus facile et **(2)** il n'y a plus de composants vides comme le *Before* dans le cas d'un PBR simple de l'approche par composants substituables. Concernant le point **(1)**, l'ordonnancement est effectué par l'Ordonnanceur et non plus par message asynchrone. Nous n'avons plus besoin de faire de l'édition de lien dynamique pour manipuler le graphe d'actions. Concernant le point **(2)**, le *Before* du PBR était présent pour faciliter l'adaptation mais ne contenait pas d'actions de SdF. Il était en effet plus facile de substituer un composant vide que de faire de l'édition de lien dynamique. Ici, seuls les objets dynamiques nécessaires au fonctionnement du FTM sont présents.

Dans cette nouvelle approche, nous avons décidé de garder le même schéma de conception pour la détection et la sélection des mécanismes avec deux entités externes au FTM, le Moniteur et l'Opérateur. L'*Adaptation Engine* est remplacé par l'*Adaptive Entity*, car le contrôle du graphe (arrêt, redémarrage de l'exécution du graphe) doit être interne au FTM. Le Moniteur reste le même et a pour rôle d'ob-

server l'évolution du système et de détecter les changements de contexte applicatif entraînant la perte de validité du FTM mis en place. L'Opérateur est l'organisme décisionnaire qui valide la transition d'un FTM vers un autre sur demande du Moniteur.

Nous pouvons voir l'*Adaptive Entity* comme un contrôleur et un coordinateur lorsqu'une manipulation du graphe d'actions (reconfiguration, adaptation ou composition) est nécessaire. Ainsi, pour mettre en place l'AFT, nous estimons que les objets statiques Gestionnaire d'objet et *Adaptive Entity* sont essentiels. Nous allons examiner les différentes étapes lors d'une adaptation pour vérifier que ces objets sont suffisants.

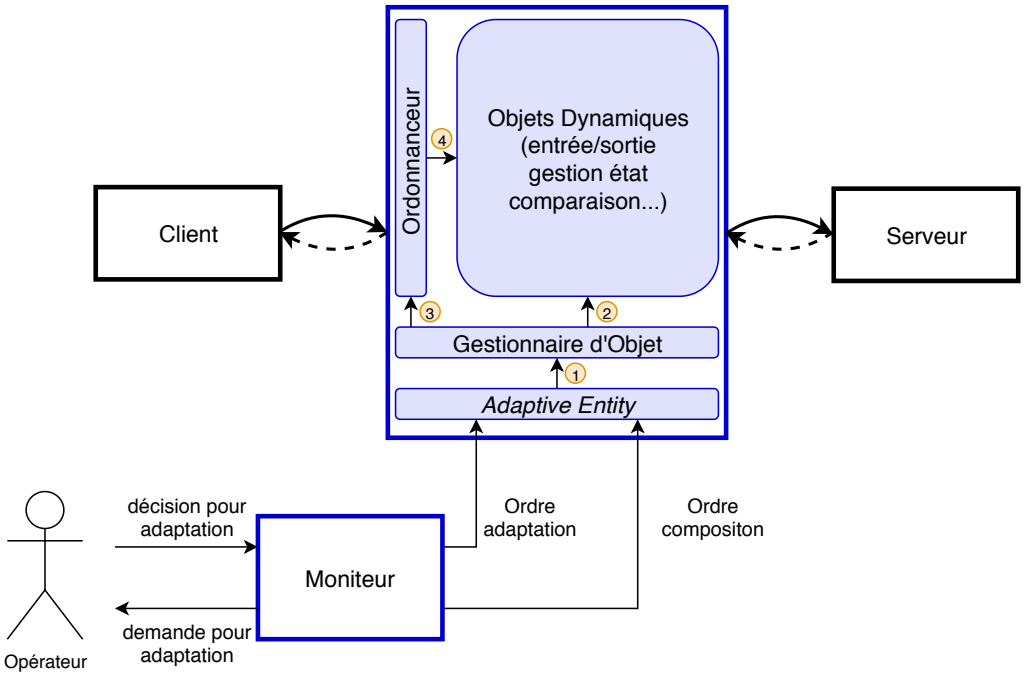


FIGURE 4.9 – Ordre des interactions entre les objets pour l'adaptation

La figure 4.9, rappelle la structure complète du FTM en ajoutant l'ordre d'exécution des différentes interaction entre les objets :

0. Le Moniteur, après validation de l'Opérateur, envoie à l'*Adaptation Entity* le mécanisme à mettre en place
1. L'*Adaptation Entity* liste et formate les nouveaux objets dynamiques requis et envoie la liste au Gestionnaire
2. Le Gestionnaire compare la nouvelle liste d'objets avec sa propre liste d'objets actifs et modifie le graphe d'actions.
3. Le Gestionnaire construit la nouvelle table en fonction du FTM choisi puis elle est récupérée par l'Ordonnanceur.

4. L'Ordonnanceur peut maintenant orchestrer l'algorithme de fonctionnement du FTM.

Nous estimons donc que ces objets conçus ainsi sont suffisants pour le fonctionnement du FTM et pour la mise en œuvre de l'AFT. Nous avons maintenant toutes les cartes en main pour implémenter notre approche par objets ordonnançables. Nous allons nous intéresser à une possibilité d'implémentation des objets dynamiques offerte par ROS, les *plugins*. Nous allons dans la prochaine section expliquer le fonctionnement des *plugins* avant d'entrer dans le détail de l'implémentation de l'AFT dans la section suivante.

2 Les *plugins* dans ROS

Pour pallier le problème de ressources inutilement utilisées, nous avons conçu plus tôt dans ce chapitre une architecture pour concevoir des FTM mono-processus se basant sur la manipulation d'objets dynamiques à travers un trio d'objets statiques, à savoir, l'Ordonnanceur, le Gestionnaire et l'*Adaptive Entity*. Nous allons expliquer, dans le suite de se chapitre, la façon dont nous avons implémenté nos FTM mono-processus sur ROS en utilisant les *plugins*. Nous nous concentrons dans un premier temps, sur l'implémentation pour un fonctionnement nominal des FTM.

2.1 Définition d'un *plugin*

Le *plugin* est par définition un composant logiciel qui peut être rattaché à une application pour en étendre les fonctionnalités. Historiquement utilisé par les navigateurs internet, ces modules externes permettaient, par exemple, la lecture de contenus multimédias avec Adobe Flash Player.

L'avantage certain des *plugins* est de permettre l'évolution du système a posteriori ou encore d'augmenter la compatibilité du logiciel avec de nouveaux formats de données. Nous pouvons prendre comme exemple le logiciel d'édition audio Audacity qui a recourt à un *plugin* pour exporter les fichiers audio au format mp3. Un autre exemple, toujours avec Audacity, est l'ajout des *plugins* LADSPA (*Linux Audio Developer's Simple Plugin API*) enrichissant les capacités de traitement du signal d'Audacity.

Ce qui est intéressant avec la notion de *plugin*, c'est la possibilité d'ajouter ou de retirer des fonctionnalités à tout moment. C'est une notion qui nous attire pour l'implémentation de nos objets dynamiques pour pouvoir changer facilement les actions de SdF qui composent nos FTM.

2.2 Une bibliothèque pour créer des *plugins*

Rappelons rapidement la structure interne d'un projet ROS. ROS utilise la notion de *package* pour structurer les différents nœuds qui forment une application logicielle. Chaque *package* contient les codes sources d'un ou plusieurs nœuds, un *CMakefile* pour créer les exécutables ou exporter les librairies et enfin un fichier *package.xml* pour importer les dépendances nécessaires à la compilation. Si nous souhaitons utiliser dans un *package1* une librairie issue d'un *package2*, il faut exporter la librairie du *package2* et inclure la dépendance dans le *package1*. Imaginons que nous souhaitons instancier dans notre FTM des objets issus des différentes classes génériques du Chapitre 3. Pour ce faire, il faut que le FTM importe les dépendances de tous les *packages* contenant les différentes classes et connaisse tous les différents objets qu'il doit instancier.

Cette façon de procéder n'est pas idéale pour permettre l'évolution du système de manière simple et rapide. Cependant, ROS utilise la bibliothèque ***pluginlib*** pour permettre l'utilisation de *plugins* à l'intérieur d'un composant. Dans ROS, les plugins sont des classes C++ instanciables dynamiquement. Celles-ci sont chargées

à partir d'une bibliothèque d'exécution (p.ex. objet partagé, *dynamically linked library*).

La création détaillée des plugins est expliquée dans la section suivante. Le principe est de créer des objets C++ issus d'une classe mère abstraite. Ces objets sont ensuite exportés dans une bibliothèque référencée sous le nom de la classe mère. Enfin, nous chargeons dynamiquement en mémoire la librairie en utilisant une commande de la bibliothèque ***pluginlib*** et nous instancions dynamiquement les classes filles en utilisant également une commande de ***pluginlib*** équivalente à une instanciation dynamique en C++ (`new object`).

L'avantage principal de l'utilisation de cette bibliothèque ***pluginlib*** en comparaison d'une instanciation dynamique d'objet est de ne charger en mémoire que les objets que nous utilisons. De plus, tant que l'interface des méthodes utilisées dans le programme n'est pas modifiée, il est possible de remanier le code source des plugins sans avoir à recompiler tout le projet, par exemple, si nous décidons de changer un algorithme de calcul de moyenne dans un FTM de type MooN (*Triple Modular Redundancy*).

2.3 Étapes de création d'un *plugin*

Nous détaillons, dans cette sections, les étapes de création d'un *plugin* à travers l'exemple du tutoriel de ROS. Nous expliciterons la manière dont nous avons implémenté nos méthodes d'action dans des *plugins* plus loin car celles-ci demandent quelques astuces d'implémentation indépendantes de *pluginlib* et nuisent à l'explication. Dans cet exemple nous allons voir la création et l'utilisation d'un *plugin*, un triangle équilatéral issus de la classe mère polygone :

```
1  namespace polygon_base{
2    class RegularPolygon{
3      public:
4        virtual void initialize(double side_length) = 0;
5        virtual double area() = 0;
6        virtual ~RegularPolygon(){}
7      protected:
8        RegularPolygon(){}
9    };
10 };
```

Listing 4.1 – Classe mère polygone

Étape 1 : Concevoir la classe mère polygone la classe fille comme l'illustre les figures du Listing 4.1 et 4.2. La classe mère contient deux méthodes virtuelles, une pour initialiser le *plugin*, l'autre pour calculer l'aire du polygone. La classe fille Triangle a en plus une méthode pour calculer sa hauteur. Dans cette étape, la conception des *plugins* est réalisée sous forme d'une collection d'objets (ici C++), qui seront

```
#include <package/polygon_base.h>
#include <cmath>
namespace polygon_plugins{
  class Triangle : public polygon_base::RegularPolygon  {
    public:
      Triangle(){}
      void initialize(double side_length){
        side_length_ = side_length;
      }
      double area(){
        return 0.5 * side_length_ * getHeight();
      }
      double getHeight() {
        return sqrt((side_length_ * side_length_) - ((side_length_ / 2) *
        ↪  (side_length_ / 2)));
      }
    private:
      double side_length_;
  };
};
```

Listing 4.2 – Classe fille contenant le plugin Triangle

Étape 2 : Définir, pour ROS, les objets comme des *plugins*. Pour cela, les classes doivent être marquées comme des classes contenant des *plugins*. Ceci est fait par la macro spéciale `PLUGINLIB\_EXPORT\_CLASS` que l'on peut voir sur la figure du Listing 4.3. Cette macro a deux fonctions, définir l'objet Triangle comme un plugin et l'enregistrer comme étant dérivé de la classe *RegularPolygon*.

```
#include <pluginlib/class_list_macros.h>
#include <pluginlib_tutorials_/polygon_base.h>
#include <pluginlib_tutorials_/polygon_plugins.h>

PLUGINLIB_EXPORT_CLASS(polygon_plugins::Triangle,
  ↪  polygon_base::RegularPolygon)
```

Listing 4.3 – Exportation de la bibliothèque de *plugins* polygone

Étape 3 : Compiler la bibliothèque en ajoutant, dans le CMakeFile la commande add_library avec les fichiers sources comme l'illustre la figure 4.4 (ligne 6). De plus, il faut exporter les *plugins* dans le fichier *package.xml* (ligne 16 à 18) pour que les autres *packages* puissent l'utiliser.

```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)

add_library(polygon_plugins src/polygon_plugins.cpp)
add_dependencies(polygon_plugins ${stmngr_EXPORTED_TARGETS}
  ↪  ${catkin_EXPORTED_TARGETS})
```

Listing 4.4 – Exportation de la bibliothèque de *plugins* polygone

```
<?xml version="1.0"?>
<package format="2">
  <name>polygon_plugins</name>
  <version>0.0.0</version>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>pluginlib</build_depend>
  <build_export_depend>pluginlib</build_export_depend>
  <exec_depend>pluginlib</exec_depend>

  <build_depend>roscpp</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <exec_depend>roscpp</exec_depend>

  <export>
    <stmngr plugin="${prefix}/polygon_plugins.xml" />
  </export>
</package>
```

Listing 4.5 – plugin.xml

Étape 4 : Créer le fichier *polygon_plugins.xml* pour définir le chemin de la librairie qui sera chargée dynamiquement ainsi que les *plugins* qu'elle contient et la classe dont ils dérivent comme indiqué sur la figure du Listing 4.6.

```
1 <library path="lib/libpolygon_plugins">
2   <class type="polygon_plugins::Triangle"
    ↪  base_class_type="polygon_base::RegularPolygon"/>
3 </library>
```

Listing 4.6 – plugin.xml

Nous insistons sur ce qui est nommé *base_class* (ligne 2), la classe dont dérivent les *plugins* car la bibliothèque est référencée sous ce nom. ***Pluginlib*** récupérera le chemin de la bibliothèque *libpolygon_plugins* (ligne 1) en utilisant le nom de cette classe mère à travers une commande expliquée plus loin.

2.4 Utilisation d'un *plugin*

Maintenant que nous avons le *plugins* Triangle enregistrés nous allons voir comment les instancier et utiliser leur méthode.

```
1  #include <pluginlib/class_loader.h>
2  #include <package/polygon_base.h>
3
4  int main(int argc, char** argv){
5    //création d'un objet ouvrant la librairie RegularPolygon
6    pluginlib::ClassLoader<polygon_base::RegularPolygon>
     ↪  poly_loader("package", "polygon_base::RegularPolygon");
7
8    try{
9      //création d'un pointeur chargeant dynamiquement le plugin Triangle
10     boost::shared_ptr<polygon_base::RegularPolygon> triangle =
       ↪  poly_loader.createInstance("polygon_plugins::Triangle");
11     triangle->initialize(10.0);
12
13     //Appel de la méthode du plugins
14     ROS_INFO("Triangle area: %.2f", triangle->area());
15   }
16   catch(pluginlib::PluginlibException& ex){
17     ROS_ERROR("The plugin failed to load for some reason. Error: %s",
       ↪  ex.what());
18   }
19   return 0;
20 }
```

Listing 4.7 – main.cpp

Le code présenté sur la figure du Listing 4.7 montre la création du *plugin* Triangle, son initialisation et l'utilisation de la méthode du calcul d'aire. Nous remarquons deux étapes importantes pour utiliser nos *plugins* :

Étape 1 : Création de l'objet *poly_loader* (ligne 5) prenant en paramètre le *package* dont sont issus les *plugins* ainsi que le nom de la classe mère. En créant cet objet, on charge la bibliothèque de classe en mémoire permettant l'instanciation dynamique des objets qu'elle contient.

Étape 2 : Instanciation dynamique des plugins (ligne 8). L'objet *poly_loader* inclut la méthode `createInstance` qui prend en paramètre le nom de classe du *plugin* et renvoie un pointeur.

Étape 3 : Appel des méthodes du *plugin* (ligne 11 & 14) en utilisant le pointeur auquel il est associé.

Concernant l'étape 2, il n'y a pas de différence majeure entre la commande :

```
poly_loader.createInstance("Triangle")
```

et la commande :

```
Triangle *triangle = new Triangle()
```

Sauf que nous ne pouvons pas utiliser cette commande (`new`) avec le code tel quel. En effet, la bibliothèque est chargée dynamiquement en mémoire, donc après compilation et le programme ne connaît pas l'objet *polygon_plugins*::*Triangle*.

Ce que l'on peut retenir de la création de *plugins* sous ROS peut être résumé en trois points : **(1)** les objet peuvent être développés sans se soucier de la bibliothèque *pluginlib*, **(2)** il faut faire attention aux *namespaces* utilisés pour la création des *plugins*, **(3)** deux éléments sont essentiels pour la manipulation des *plugins*, l'objet *loader* permettant de charger en mémoire la bibliothèque contenant les *plugins* et la commande `createInstance()` permettant d'instancier dynamiquement les *plugins*.

L'avantage de l'utilisation des *plugins* est de pouvoir créer et instancier des objets en langage C++ de manière dynamique en ne connaissant que la classe mère. C'est principalement pour éviter de charger en mémoire toutes les bibliothèques possibles (et non juste celle qui nous intéresse pour instancier certains objets dynamiques) que nous utilisons *pluginlib* et non une instanciation dynamique d'objet classique avec le passage par référence.

Nous allons maintenant nous intéresser à l'implémentation des FTM et la réalisation des fonctions d'adaptation et de composition en utilisant les *plugins*.

3 Implémentation des FTM sous ROS

Nous avons besoin, pour créer nos mécanismes de tolérance aux fautes monoprocessus, d'un support d'exécution qui permette la création et la manipulation d'objets dynamiques. Pour cela, le support doit permettre l'instanciation et la suppression d'objets C++ à l'exécution. Nous avons pu découvrir dans la section précédente que ROS avait comme fonctionnalité la création et la manipulation de *plugins*. Nous allons maintenant expérimenter cette fonctionnalité pour concevoir nos FTM.

Nous allons dans un premier temps implémenter les objets statiques avec les méthodes exigées pour le fonctionnement nominal. Dans un deuxième temps, nous implémenterons les objets dynamiques et examinerons les contraintes liées à leur manipulation. Enfin dans un troisième et dernier temps, nous analyserons l'implémentation des méthodes utilisées pour la reconfiguration d'un FTM (modification du secondaire en cas de crash du primaire), l'adaptation et la composition.

3.1 Implémentation des objets statiques

Avant de parler de l'implémentation des objets dynamiques, il est intéressant d'examiner l'implémentation des objets statiques qui vont imposer des contraintes lors du développement. Ces objets sont contenus dans le corps du FTM qui contient la machine d'état globale du mécanisme. Nous nous intéresserons d'abord au fonctionnement nominal du FTM. Pour cela, nous regarderons en premier l'Ordonnanceur puis le Gestionnaire d'Objet.

3.1.1 Le Gestionnaire d'objet

Le rôle du Gestionnaire d'objet est multiple. Il doit charger en mémoire les bibliothèques contenant les *plugins*, instancier dynamiquement les *plugins*, récupérer leur méthode par l'intermédiaire des pointeurs de fonction, construire la table d'orchestration et la transmettre à l'Ordonnanceur. Pour valider l'implémentation de l'approche par actions ordonnançables, nous avons implémenté les trois FTM qui sont le PBR, le TR, le LFR et leur composition. Ces trois mécanismes suffisent à effectuer les différentes transitions (reconfiguration, adaptation, composition) pour valider la mise en œuvre de l'AFT. Pour des raisons de simplification, les tables d'orchestration sont construites manuellement et non automatiquement depuis un fichier. Nous n'avons pas effectué le travail d'interprétation nécessaire pour transformer le fichier en table d'orchestration, ce qui ne pose pas de difficulté particulière.

3.1.2 Création des *plugins*

La création de *plugins* passe d'abord par le chargement en mémoire de la bibliothèque les contenant. Pour rappel, ces bibliothèques contiennent les classes mères et filles qui correspondent aux catégories et sous-catégories de nos actions de SdF. Pour charger dynamiquement les bibliothèques, nous créons autant d'objets *ClassLoader* que de classes génériques d'action de SdF que nous utilisons comme montré

sur la figure du listing 4.7 de la section 2.4 de ce chapitre (ligne 5).

```
psm_loader =
new pluginlib::ClassLoader<gestEtat_base::GestEtat>("GestionEtat",
↪  "gestEtat_base::GestEtat");
```

Cet exemple illustre le chargement de la bibliothèque Gestion d'État qui se situe dans le *package GestionEtat* et dont la classe mère est *gestEtat_base : :GestEtat*. L'objet *psm_loader* créé va permettre d'instancier un objet dynamique à l'aide de la commande createInstance (ligne 8 de la même figure).

Une fois toutes les bibliothèques nécessaires chargées en mémoire, nous instancions dynamiquement nos *plugins*. Pour ce faire, nous utilisons les *loaders* précédemment créés avec la méthode.

```
boost::shared_ptr psm =
↪  psm_loader->createInstance("gestEtat_plugin::GestEtat_GetSet_Ckpt")
psm->init();
```

Ce code illustre la création et l'initialisation d'un *plugin* de Gestion d'État qui utilise une technique de type *Getter/Setter* avec transmission de l'état par message de *checkpoint*. C'est l'implémentation de l'objet dynamique de gestion d'état utilisé dans un PBR. Nous effectuons la même démarche pour tous les *plugins* nécessaires à la création du graphe d'actions. Nous allons maintenant étudier la construction de la table d'orchestration.

3.1.3 De la table d'orchestration à la *map*

Nous avons vu dans la section 1.3.2 que la table d'orchestration fonctionne comme un dictionnaire. Nous utilisons des clefs pour représenter les actions de SdF et qu'à ces clefs sont associées des définitions. La définition contient le pointeur de fonction vers la méthode d'action, la clef suivante si le résultat de la méthode est Vrai, la clef suivante si le résultat est Faux et la clef de traitement d'erreur.

$$Clef_{Action} \Rightarrow \{^{*}Méthode,\ Clef_{Vrai},\ Clef_{Faux},\ Clef_{Err}\}$$

Il existe dans la bibliothèque standard C++ un objet permettant de réaliser notre table d'orchestration. Cet objet s'appelle une *map* et permet d'associer à une clef (type string) n'importe qu'elle donnée :

```
std::unordered_map<std::string,S>
```

Ainsi, nous avons créé une structure présentée sur la figure de Listing 4.8 permettant de regrouper les pointeurs de fonction ainsi que les différentes clefs associées au résultat Vrai ou Faux et au traitement d'erreur. Le `typedef` définit l'interface des méthodes des *plugins*. Cela permet de définir le type de nos méthodes utilisées. Nous allons expliquer, dans l'implémentation de l'Ordonnanceur, le choix de cette interface pour l'implémentation des méthodes d'action.

```
#include <functional>

typedef std::function<int(void*,void*)> sche_function;

struct S{
      sche_function funct;
      std::string next0;
      std::string next1;
      std::string err;
};
```

Listing 4.8 – Structure représentant la définition associées à une clef d'action

Le Gestionnaire va remplir chaque ligne de la *map* en associant à une même clef cette structure. Pour ce faire, il va récupérer les pointeurs de fonction des méthodes d'action des *plugins*. Nous avons donc pour chaque bibliothèque contenant des *plugins* un accesseur qui récupère la méthode d'action du *plugin*. De plus, il va ajouter à la *map* la clef INIT à laquelle il associe juste à la clef suivantVrai, la clef de l'action suivante. La clef END est ajoutée comme clef suivantVrai pour la dernière action. Une fois la *map* remplie avec l'algorithme du FTM choisi, le Gestionnaire d'objets la transmet à l'Ordonnanceur pour entamer le fonctionnement du FTM.

3.1.4 L'Ordonnanceur

Avant d'entrer dans les détails de l'implémentation de l'Ordonnanceur, nous nous intéressons à l'implémentation de l'algorithme d'ordonnancement qu'il suit. Nous avons défini plus tôt une table d'orchestration dans laquelle une clef est associée à un ensemble contenant la méthode à appeler, et les clefs suivantes en fonction du résultat retourné par la méthode (Vrai, Faux, Erreur). Cette table est construite sous forme de *map* dans le Gestionnaire d'objets.

Pour que l'Ordonnanceur puisse appeler les méthodes d'objets différents, il faut appliquer une contrainte forte. L'interface de la méthode doit être générique (définition du type de retour et des paramètres). Ainsi, l'Ordonnanceur peut appeler n'importe quelle méthode d'action par l'intermédiaire d'un pointeur de fonction.

Occupons nous d'abord de la première contrainte. Les méthodes appelées sont issues d'objets différents et doivent donc être standardisée pour que l'Ordonnanceur puissent avoir un algorithme générique. De plus, la méthode doit renvoyer une valeur pour définir la méthode suivante à appeler. Ensuite, les paramètres doivent être génériques. Nous avons décidé de faire en sorte que les méthodes d'actions aient comme type de retour un entier et comme paramètre deux *void**. De plus, comme le graphe d'actions est séquentiel, nous avons décidé que la valeur de sortie (out) d'un objet sera la valeur d'entrée (in) de l'objet suivant.

```
int methode(void* in, void* out)
```

Le choix de ce type pour les paramètres vient d'une volonté d'avoir une interface générique alors que le type des paramètres diffère en fonction des méthodes choisies. En effet, cela nous permet de transmettre des données différentes aux objets sans se préoccuper pas du type des paramètres. C'est une solution acceptable car nous passons par des variables intermédiaires dans les méthodes pour ne pas manipuler directement les paramètres. De plus, la taille de ces paramètres est allouée à l'initialisation dans le conteneur du FTM et transmis à l'Ordonnanceur pour qu'il puisse appeler les méthodes d'actions avec ces paramètres.

Pour simplifier l'utilisation de cette interface, nous avons utilisé le template de classe `std::function` qui permet de manipuler une cible appelable (ici une fonction) en définissant par avance son type et ses paramètres. Cela permet de créer un alias (*typedef*) pour définir des méthodes génériques. Nous verrons dans la section suivante comment nous avons récupéré ces méthodes à partir des *plugins*

La figure du listing 4.9 explicite les différentes méthodes contenues dans l'Ordonnanceur. Nous avons une méthode d'initialisation (ligne 3) avec récupération des adresses contenant les paramètres qui seront transmis aux différentes fonctions, une méthode d'ordonnancement (ligne 10), une méthode de récupération de la *map* (ligne 23), une méthode de récupération de la première méthode à appeler (ligne 28) et enfin la méthode de récupération de la méthode suivante à appeler en fonction de la condition de retour (ligne 34). De plus nous avons un constructeur et un destructeur qui ne sont pas représentés ici car ils ne présentent pas de spécificités particulières.

A ce point, nous connaissons les différentes méthodes incluses dans l'Ordonnanceur et le Gestionnaire d'objet. Nous allons maintenant nous intéresser aux objets dynamiques et aux méthodes liées à cette nouvelle approche.

3.2 Implémentation des objets dynamiques

Les objets dynamiques sont le cœur des FTM mono-processus car ce sont eux qui contiennent les méthodes implémentant les actions de SdF. De plus, ils sont contrôlés par le Gestionnaire d'objet et orchestrés par l'Ordonnanceur. Nous avons vu dans la section 1 que l'Ordonnanceur orchestre le graphe d'actions en appelant les méthodes d'action selon un certain ordre. De plus il faut que les méthodes appelées aient une interface générique.

Chaque objet dynamique est un *plugin*. Les méthodes d'action ne sont pas impactées par la transformation en *plugin*. Nous avons donc réutilisé le code des composants de l'approche par composant substituables en adaptant l'interface pour qu'elle corresponde au `typedef` inclus dans la structure de la *map*.

Pour récupérer les méthodes des *plugins* et les inclure dans la *map*, nous avons dû utiliser `std::bind`, une méthode également issue de la bibliothèque standard. La méthode `std::bind` génère une déviation d'appel pour une méthode. Appeler cette déviation équivaut à invoquer la méthode elle même avec ses arguments. En bref, cela permet de récupérer le pointeur de fonction d'une méthode avec ses arguments, sans contraintes sur le type de la méthode et de ses paramètres. La figure du Listing

```
#include "scheduler.h"

void Scheduler::initialize(void* in,void* out){
  _in = in;
  _out = out;
  cond = 0;
};

// Algorithme d'ordonnancement
void Scheduler::schedule(){
  // Récupération de la première méthode
  fct=getInit();
  while(fct!=NULL){
    // Exécution de la méthode
    cond=fct(_in, _out);
    std::memcpy(&_in,&_out,sizeof(_out));
    // Sélection de la fonction suivante
    fct=getNext(cond);
  }
}

// Récupération de la table d'orchestration
void Scheduler::setMap(std::unordered_map<std::string,S> m){
  _m=m;
}

// Sélection de la fonction d'initialisation
sche_function Scheduler::getInit(void){
  key=_m["init"].nextok;
  return _m[key].funct;
}

// Selection de la clef suivante en fonction de la valeur de cond
sche_function Scheduler::getNext(int cond){
  if (cond==1){
    key = _m[key].nextok;
  }
  else if (cond==0){
    key = _m[key].nextnotok;
  }
  else {
    key = _m[key].err;
  }
  return _m[key].funct;
}
```

Listing 4.9 – Implémentation des classes de l'Ordonnanceur

4.10 illustre la récupération de la méthode de comparaison de l'objet de Sélection

de données *Comparator*.

```
1 sche_function getFunction(){
2   return std::bind(&Comparator::compare, this,
    ↪  std::placeholders::_1,std::placeholders::_2);
3 };
```

Listing 4.10 – Méthode de récupération de méthode si une action

Les paramètres de cette méthode sont l'adresse de la méthode concernée (ici &Comparator : :compare), l'adresse de l'objet auquel appartient la méthode (*this*) ainsi que les emplacements des arguments. Ici, la fonction `getFunction` renvoie l'équivalent d'un pointeur de fonction de la méthode `int compare(void*, void*)` qui sera utilisée dans le TR.

Pour les *plugins* ayant plusieurs méthodes pouvant être utilisées par l'Ordonnanceur (Gestion d'État qui comprend la capture et la mise à jour d'état), nous avons ajouté à la fonction `getFunction` une variable pour sélectionner l'une ou l'autre méthode comme l'illustre la figure du listing 4.11. La méthode getFunction de l'objet Gestion d'État sera appelée par chaque réplique. Une fois dans le Primaire du PBR pour récupérer la méthode d'action de capture d'état et une fois dans le Secondaire pour récupérer la méthode d'action de mise à jour d'état.

```
sche_function getFunction(int c){
  if(c==1){
    return std::bind(&GestEtat_GetSet_Ckpt::getState, this,
    ↪  std::placeholders::_1,std::placeholders::_2);
  }
  else if(c==2){
    return std::bind(&GestEtat_GetSet_Ckpt::setState, this,
    ↪  std::placeholders::_1,std::placeholders::_2);
  }
};
```

Listing 4.11 – Méthode de récupération de méthode si plusieurs actions

Nous avons maintenant tous les outils en main pour que nos FTM puissent fonctionner de manière nominale. Nous allons maintenant étudier les méthodes supplémentaires primordiales à la reconfiguration, l'adaptation et la composition.

3.3 Adaptation et Composition

Dans le Chapitre 2, nous avions distingué trois types de modification du graphe de composant :

1. **La reconfiguration :** Changement de l'état d'un composant (Le Secondaire du PBR qui devient Primaire suite à la défaillance de celui-ci) ;
2. **L'adaptation entre deux FTM :** Transition d'un FTM vers un autre en cas de changement de contexte applicatif (AC, RS)
3. **La composition de FTM :** Assemblage de deux FTM pour tolérer un nouveau modèle de fautes en plus de l'ancien.

Ces trois types de modification impliquaient trois méthodes distinctes, respectivement une manipulation des communications inter-répliques pour la reconfiguration, une substitution des composants *Before* et *After* d'un FTM par ceux d'un autre pour l'adaptation, et enfin la substitution du composant *Proceed* par un ensemble *P-BPA* pour la composition.

L'approche par objets ordonnançables va permettre d'uniformiser les différentes méthodes et va faire de la reconfiguration et de la composition des sous-ensembles de l'adaptation. Comme nous avions pu le voir sur la sous-machine d'état de transition (figure 4.8 page 122 de la section 1.5), l'adaptation consiste en trois opérations :

1. Suppression des *plugins* contenant les actions de SdF obsolètes ;
2. Instanciation des *plugins* valides ;
3. Mise à jour de la *map* contenant l'algorithme d'ordonnancement.

La reconfiguration et la composition reprennent les opérations 2 et 3 pour la composition et l'opération 3 pour la reconfiguration.

Ainsi, lors d'une demande d'adaptation du Moniteur après validation de l'Opérateur, l'*Adaptive Entity* va changer la valeur d'une variable (un flag) qui va permettre de rentrer dans l'état de transition. Nous n'avons pas implémenté la partie

de formatage de la liste d'objet et de transition mais avons directement prévu la modification du graphe dans le Gestionnaire d'Objets. Ainsi, l'*Adaptive Entity* va transmettre au Gestionnaire d'Objets le FTM à mettre en place et ce dernier va effectuer les modifications demandées.

Prenons trois scénario et analysons les méthodes utilisées dans ces trois scénarios puis construisons notre *Adaptation Entity* et complétons notre Gestionnaire d'objet à partir de ceux-ci.

Le premier scénario correspond à la perte du Primaire dans le cas du PBR. En fonctionnement nominal, le Primaire et le Secondaire sont identiques dans leur constitution. Le Primaire a cependant une table d'orchestration complète alors que le Secondaire n'effectue que l'action de mise à jour d'état sur réception d'un message de *checkpoint*. Le Primaire subit un crash de sa plate-forme matérielle, le *WatchDog* du Secondaire le détecte et envoie un message au composant de Recouvrement identique à la première approche. Le composant de Recouvrement envoie un message à l'*Adaptation Entity* du Secondaire. Ce dernier signale au Gestionnaire le besoin d'une reconfiguration, le Gestionnaire **reconstruit la *map*** du Secondaire identique à celle du Primaire puis l'Ordonnanceur la récupère.

Le deuxième scénario correspond à la composition d'un PBR et d'un TR. Le PBR est constitué des objets de communication et d'un Gestionnaire d'État avec *checkpoint*. Le TR est constitué d'un Gestionnaire d'État sans *checkpoint* et d'un Comparateur. Le Moniteur envoie un ordre de composition à l'*Adaptive Engine* qui sélectionne les mécanismes à composer. Ce dernier envoie à l'*Adaptation Entity* les mécanismes à mettre en place et sélectionne les objets du TR à ajouter au graphe d'action puis transmet la liste au Gestionnaire. Le Gestionnaire **instancie dynamiquement** les *plugins* du TR, **reconstruit la *map*** et la transmet à l'Ordonnanceur.

Le troisième scénario correspond à l'adaptation entre un PBR+TR et un LFR+TR. Pour effectuer cette adaptation, il faut **supprimer** le Gestionnaire d'État avec *checkpoint*, **instancier dynamiquement** les *plugins* de Synchronisation et **reconstruire la *map*** avant de la transmettre à l'Ordonnanceur comme l'illustre la figure 4.10.

Ainsi, les opérations utilisées pour l'adaptation sont réutilisées pour la composition et la reconfiguration. Il n'y a donc plus de différence significative concernant les méthodes des objets statiques du FTM entre ces trois modifications. Si nous voulons faire une reconfiguration, nous reconstruisons seulement la *map*. Si nous voulons faire une composition, nous effectuons instancions dynamiquement les *plugins* et nous reconstruisons la *map*.

Avec ces trois scénarios, nous avons défini les actions de l'*Adaptive Entity*. Il faut que celui-ci puisse recevoir des messages de la part du Moniteur qui sélectionne les mécanismes à mettre en place mais également du composant de Recouvrement dans le cas d'une détection de crash comme l'illustre la figure 4.11. De plus, il lui faut qu'il puisse sélectionner les *plugins* en fonction des mécanismes choisis et une action de transmission de la modification du graphe d'actions au Gestionnaire.

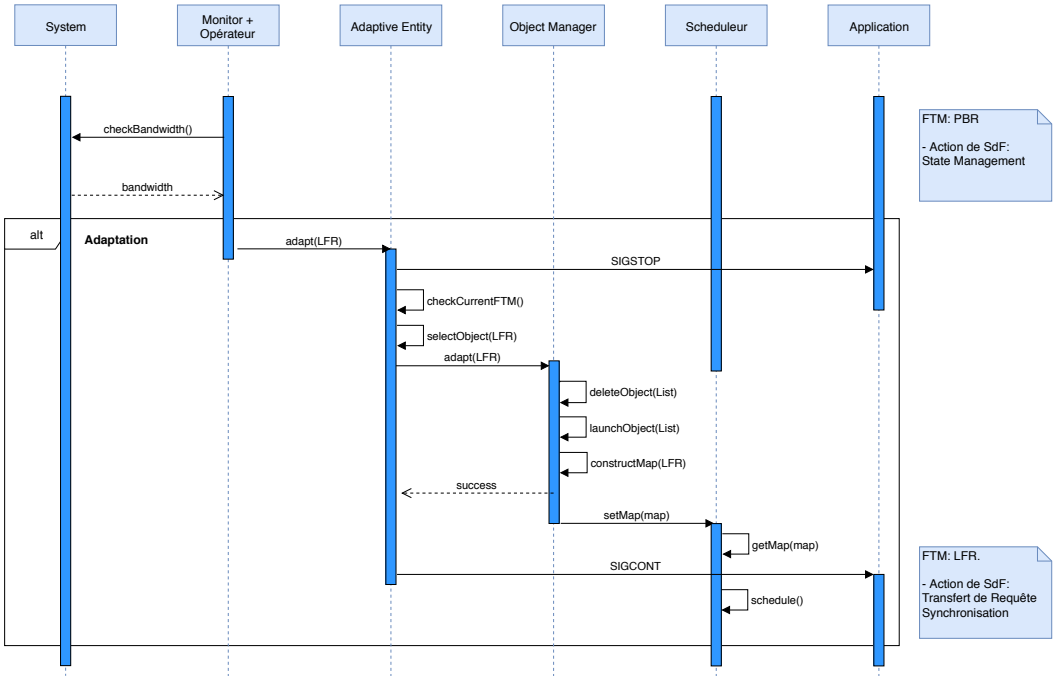


FIGURE 4.10 – Diagramme de séquence d'une adaptation

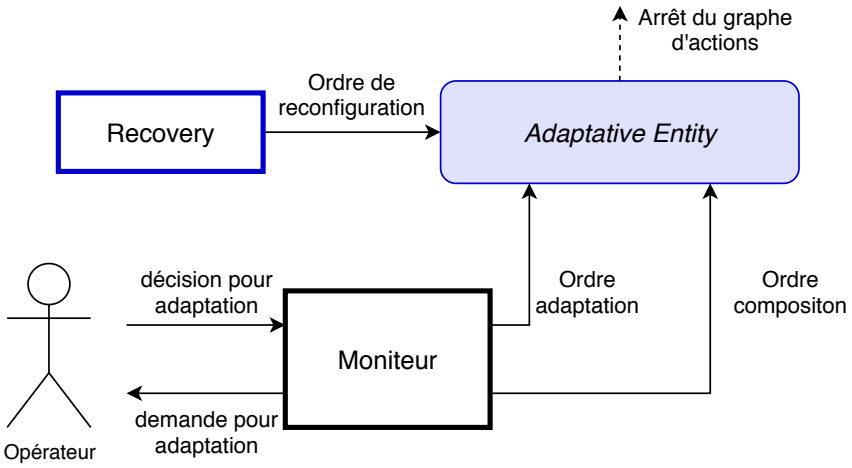


FIGURE 4.11 – Interaction entre les processus externes et l'*Adaptation Entity*

Nous allons maintenant comparer cette approche avec celle du Chapitre 2 et examiner notre réponse aux problèmes de ressources. De plus nous auront une analyse critique de cette approche pour en déterminer les défauts qui pourraient la rendre moins appropriée à l'utilisation que l'approche par composants substituables.

4 Analyse critique de la nouvelle approche

Dans cette section, nous allons évaluer si cette approche résout les problèmes soulevés par l'approche par composants substituables concernant les ressources utilisées. Nous avons déjà pu constater que cette approche intègre facilement et naturellement l'optimisation faite sur la réutilisation de code et l'affinage de la granularité du Chapitre 3. Il nous faudra également avoir un point de vue critique sur cette approche et analyser si cette approche ne soulève pas d'autres problèmes de conception ou d'implémentation.

Nous analyserons, en premier lieu, les différences fondamentales entre un nœud ROS et un plugin ROS. En second lieu nous comparerons les avantages conceptuels de cette approche par objet ordonnançable vis-à-vis de l'approche par composants substituables. Enfin, en troisième lieu, nous évaluerons les aspects négatifs de cette approche.

4.1 Avantages d'implémentation

Si nous devions catégoriser les avantages de la seconde approche face à la première, il se dégagerait une catégorie majeure, les avantages concernant l'implémentation. Plusieurs avantages se dégagent et concernent l'utilisation de *plugins* à la place des nœuds ROS.

Il est évident que les inconvénients relatifs à ROS sont toujours vrais. ROS a toujours un point de défaillance unique en la présence du *ROS Master* et ses nœuds restent actifs en attente d'une réception de message.

Concernant le premier avantage, l'approche par objets ordonnançables permet de se débarrasser des composants superflus regroupant les actions de SdF. En effet, dans l'approche par composants substituables, un PBR doit avoir un composant *Before* qui ne fait rien d'autre que transmettre la requête au *Proceed*. Cette obligation est née de la difficulté de ROS à insérer un nœud entre deux autres nœuds communiquant. Ici, la nouvelle approche permet de n'avoir que les objets dynamiques essentiels à la réalisation des actions de SdF. Les contraintes de ROS n'ont plus le même impact sur les choix d'implémentation et permettent une implémentation épurée de nos FTM.

Le second avantage concerne la manipulation des communications entrantes et sortantes. Ce problème n'est pas uniquement lié à ROS. En effet, la modification d'une communication demande une instrumentation lourde des composants. Cependant, dans cette approche, les communications sont gérées par des *plugins* et peuvent être simplement modifiées en changeant de *plugins*. Prenons un exemple, nous avons une application qui calcule la vitesse à partir d'un capteur fournissant l'accélération. Un FTM tolérant les fautes en crash est attaché à cette application. Nous souhaitons comparer la valeur de sortie de l'application avec une donnée externe issue d'une autre capteur. Il suffit de modifier le *plugin* de communication en entrée et d'accepter non pas une mais deux communications entrantes. La seconde valeur serait ensuite utilisée par un comparateur pour vérifier la vraisemblance de la

valeur de sortie de l'application. Ceci est très compliqué à faire avec l'approche par composants substituables car la modification du graphe des liens de communications n'est pas triviale.

Le troisième avantage est la capacité de rajouter facilement des *plugins* issus d'une librairie déjà existante. Seul le Gestionnaire d'objet aurait besoin d'être modifié pour manipuler le nouveau *plugin*, les méthodes de création de la *map* et du chargement de la librairie n'étant pas modifié. De plus la modification de la librairie et de ses *plugins* n'entraîneraient pas de conséquences concernant le chargement dynamique et l'instanciation des *plugins*.

Le quatrième avantage réside dans la *map* et surtout dans la structure associée à une clef. Dans les exemples que nous avons choisis, l'algorithme avait au maximum une divergence en deux branches (Au maximum deux méthodes peuvent succéder à celle en cours). Si nous avons une troisième possibilité, il suffit de rajouter une clef à la structure et de modifier la méthode d'orchestration de l'Ordonnanceur pour prendre cette possibilité en compte. Les valeurs décidant de la méthode à appeler ensuite ne seraient plus 0 et 1 mais, par exemple, 0, 1 et 2.

Nous allons maintenant nous intéresser à la conception des FTM et comparer les deux approches pour définir les avantages de celle-ci.

4.2 Avantages conceptuels

Une seconde catégorie pourrait regrouper, quant à elle, les avantages dûs à la conception. Cette catégorie regroupe les différences concernant les FTM eux-mêmes sans considérer le support d'exécution.

Le premier gros avantage de cette approche concerne la modification du graphe d'action. Contrairement à l'approche par composants substituables, l'ordonnancement des objets ne se fait pas par message mais par l'Ordonnanceur. En dehors des objets dynamiques servant aux communications avec l'application ou avec une réplique, les autres objets communiquent par variables partagées. Il n'y a donc pas de soucis de *dynamic binding* lors de la modification du FTM.

Le deuxième avantage concerne les communications. Les objets dynamiques utilisent des variables partagées pour se transmettre des informations ce qui permet une communication interne plus rapide sans avoir besoin de passer par des messages asynchrones. De plus, les objets dynamiques s'occupant de la communication peuvent être divers et ne nécessitent plus du duo *Proxy* - *Protocol*. Le *Proxy* peut être gardé pour conserver une similitude avec les architectures distribuées classiques mais le *Protocol* n'a plus de raison d'être.

Le troisième avantage concerne la composition. Nous avions défini la composition dans l'approche précédente par la substitution du *Proceed* par un ensemble complet *Protocol* - *Before* - *Proceed* - *After*. Cela nous permettait d'ajouter les composants constituant un FTM facilement. Ici, il suffit d'ajouter aux objets dynamiques déjà présents ceux essentiels à la tolérance d'un modèle de faute complexe et de modifier la *map*. C'est le cas dans le scénario d'un PBR auquel nous ajoutons un TR pour tolérer les fautes transitoires en valeur. Le PBR contient les objets de communica-

tion avec le Client et un Gestionnaire d'État avec *checkpoint*. Pour ajouter le TR, il suffit d'ajouter deux objets, un Comparateur et un Gestionnaire d'État avec état partagé, de modifier la *map* et de la transmettre à l'Ordonnanceur.

Nous allons maintenant nous intéresser aux inconvénients de l'approche par objets ordonnançables. Nous étudierons ainsi en comparant avec l'ancienne approche si nous avons ajouté plus de problèmes que nous en avons résolus.

4.3 Inconvénients de l'approche

Nous avons d'une part cherché à augmenter le niveau de réutilisation de code en nous intéressant aux actions de SdF plutôt qu'à la décomposition en *BPA* des FTM. Nous avons d'autre part changé d'approche pour résoudre le problème de ressource dû aux nœuds ROS qui restaient actifs pour contrôler la réception de messages tout en intégrant la première modification. Il nous reste maintenant à analyser les défauts de cette approche par objets ordonnançables.

L'inconvénient majeur de cette approche vient de la faible robustesse du FTM mono-processus en comparaison du FTM initial. Ce qu'on entend par robustesse c'est le comportement du FTM en cas de perte d'un processus. Pour le FTM multi-composants, la perte d'un composant ne signifiait pas la perte complète du mécanisme. Le composant peut être relancé rapidement et le FTM conserve son intégrité. Le FTM mono-processus n'ayant qu'un seul processus, la perte de celui-ci entraîne la perte complète du mécanisme et engendre une perte de disponibilité de l'application en présence de fautes dont la probabilité d'occurrence reste faible (10^{-4}/h pour une faute matérielle). Le mécanisme complet peut être relancé.

Le deuxième inconvénient important réside dans la conception des objets dynamiques. En effet, la création de ces objets et des classes génériques dont ils sont issus est plus fastidieuse. Il faut pendre en compte l'interface standard des méthodes, les méthodes de retour pour que le Gestionnaire puisse construire la *map* et l'enregistrement des objets comme *plugins*. La conception des composants est beaucoup plus simple et n'implique pas une gestion minutieuse des accès aux variables partagées.

On peut également expliciter un troisième inconvénient dans la complexité des objets statiques du mécanisme. Dans l'approche par composant substituable, les composants activent leur méthode sur réception de message. C'est une orchestration séquentielle basique qui est moins délicate que l'utilisation de la *map* et de l'Ordonnanceur.

Cette nouvelle approche peut être résumée en une phrase : Un FTM totalement configurable. Cependant cette approche n'est pas sans défaut et sacrifie la robustesse du FTM au profit d'une complète liberté de configuration. La robustesse et l'adaptabilité reposent sur un compromis. Les deux approches ont leur point fort et leur point faible et aucune n'est meilleure que l'autre et c'est à l'utilisateur de choisir celle qui lui semble la plus adaptée à la situation.

5 Synthèse

Ce quatrième chapitre nous a permis de concevoir nos mécanismes dans une optique d'épuration du superflu. Il nous a permis également d'illustrer une conception de FTM complètement configurable qui nous permet de réagir plus rapidement à l'apparition de perturbation en s'appuyant sur des *plugins*.

Ce qu'il faut retenir finalement de ce chapitre est le changement d'approche pour concevoir d'un FTM mono-processus. Cette conception s'appuie sur trois principes fondamentaux, la manipulation d'objets dynamiques, l'ordonnancement interne de ces objets et l'uniformisation des méthodes d'adaptation et de composition. Avec cette approche, la reconfiguration et la composition utilisent les mêmes méthodes que l'adaptation. Il n'y a donc pas trois techniques différentes pour mettre en œuvre l'AFT. De plus, l'implémentation de cette méthode repose sur les capacités de ROS à instancier dynamiquement des objets appelés *plugins*. Cette implémentation conserve cependant les mêmes défauts que la première concernant le point de défaillance unique du *ROS Master*.

Bien que nous ayons conçu et développé des FTM sans nous préoccuper des contraintes du logiciel embarqué automobile, nous allons maintenant nous y intéresser. En effet, le consortium AUTOSAR développe une nouvelle plate-forme pour permettre la mise à jour à distance des applications embarqués sur les différents ECU non-critiques dont ceux contenant les ADAS. Il faut donc analyser les contraintes du domaine pour évaluer la compatibilité entre nos approches et les standards automobiles.

Le prochain et dernier chapitre se concentrera sur la compatibilité de ces méthodes avec le domaine automobile. Nous évaluerons les possibilités de projection de ces méthodes sur AUTOSAR et sur Adaptive AUTOSAR. Il nous faudra analyser le niveau d'implémentation possible des deux approches sur ces deux plate-formes. Nous étudierons les capacités des deux standards et ferons une étude comparative avec ROS. Ce chapitre se divisera en trois sections. La première section se concentrera sur les capacités de la plate-forme *Classic* AUTOSAR pour implémenter nos deux approches. La deuxième se focalisera, quant à elle, sur la compatibilité entre la plate-forme *Adaptive* AUTOSAR et l'AFT. La troisième sera consacrée à une synthèse de la compatibilité entre les deux standards automobiles et l'*Adaptive Fault Tolerance*.

Chapitre 5

Application au Contexte Automobile

Sommaire

Introduction

Nos travaux se sont intéressés à un support d'exécution, ROS, sans tenir compte des contraintes liées à un domaine d'application précis. La raison de ce choix est liée à ses concepts et ses capacités de mise en œuvre de l'AFT. Nous avons donc pu théoriser, concevoir et implémenter deux approches différentes qui ont chacune leurs points forts et leurs points faibles. Nous allons maintenant nous intéresser au domaine de l'automobile et analyser la compatibilité entre nos approches et les deux standards AUTOSAR *Classic Platform* et AUTOSAR *Adaptive Platform*.

Nos deux approches ont des méthodes de conception différentes pour constituer les FTM. La première approche est une conception orientée composant. Les actions de SdF sont réunies dans deux composants en fonction de leur ordre d'exécution par rapport à l'exécution d'une fonction applicative. Toutes les actions de SdF s'exécutant en amont d'une fonction de l'application sont réunies dans le *Before*, celles s'exécutant en aval sont réunies dans l'*After*. La seconde approche se concentre sur les actions de SdF en elles-même de façon à identifier des rôles précis sous forme de méthodes implémentées dans des objets C++. Toutes les actions de SdF sont orchestrées au sein d'un même composant par un Ordonnanceur interne.

Dans ce chapitre, nous allons donc analyser la capacité des deux standards automobiles à implémenter nos approches. Pour effectuer cette analyse comparative entre les deux standards et nos approches et examiner leur compatibilité, nous devons définir les éléments équivalents entre notre support d'exécution ROS et les deux standards logiciels automobiles. Nous devons donc trouver pour chaque élément, composant, *plugin*, communication synchrone, ou encore communication asynchrone, son homologue sur les standards automobiles. Une fois tous ces éléments définis, nous pouvons examiner ce qui est identique ou adéquat, ce qui manque dans les standards pour déterminer s'ils sont compatibles avec nos approches.

La première section de ce chapitre se concentrera sur les besoins de l'industrie automobile pour faire évoluer les systèmes embarqués pendant la durée de leur vie opérationnelle. Nous discuterons des capacités de la plate-forme AUTOSAR *Classic Platform* à implémenter nos deux approches. Nous étudierons les motivations qui ont poussé les consortiums AUTOSAR à faire évoluer la plate-forme vers celle qui est en cours de développement, AUTOSAR *Adaptive Platform*.

La deuxième section se focalisera, quant à elle, sur la compatibilité entre la plate-forme AUTOSAR *Adaptive Platform* et l'AFT. Nous projeterons les éléments clefs de nos approches sur cette plate-forme faite pour l'adaptation des logiciels embarqués automobiles. Une analyse critique sera effectuée sur une étude partielle de la plate-forme, celle-ci n'étant pas encore développée totalement.

La troisième section sera consacrée à une synthèse de la compatibilité entre les deux standards automobiles et l'*Adaptive Fault Tolerance*. Nous en profiterons pour formuler notre point de vue sur les caractéristiques manquantes à la plate-forme pour réaliser l'AFT.

1 Évolutivité des systèmes automobiles

La mise à jour partielle dans les systèmes embarqués automobiles correspond à une étape importante afin, d'une part d'améliorer et faire évoluer les systèmes d'assistance à la conduite, mais aussi d'en proposer de nouveaux en utilisant le matériel existant (ADAS, système multimédia), et d'autre part de faciliter les modifications logicielles pour le constructeur automobile en cas de problèmes détectés (correction de bugs). Les mises à jour traitées ici se concentrent sur les mécanismes de tolérance aux fautes attachés aux systèmes embarqués plus ou moins critiques. Le but de notre démarche est donc d'améliorer la fiabilité et l'évolutivité des systèmes embarqués mais également la disponibilité de ses systèmes en présence de fautes.

Le principe de la plate-forme AUTOSAR *Classic Platform* est d'appliquer un niveau d'abstraction entre la couche matérielle et la couche logicielle pour les systèmes embarqués automobiles. AUTOSAR OS est basé sur l'OS temps réel dur OSEK/VDX. Un aperçu des caractéristiques techniques de AUTOSAR *Classic Platform* est disponible dans la section 5.2 du Chapitre 1. C'est un début de *Separation of Concern* qui permet de faire évoluer le logiciel sans se préoccuper de la plateforme matérielle.

1.1 Mise en œuvre de l'AFT avec AUTOSAR *Classic Platform*

Des travaux précédents ont analysé les capacités d'adaptation de AUTOSAR *Classic Platform* [Belaggoun 2016]. Une approche proposée dans [Martorell 2015] repose sur l'implantation de containers dans l'architecture afin d'aménager des espaces pour les futures mises à jour. En effet, après la génération du code, il est presque impossible de faire des modifications ultérieures, à moins de recharger intégralement le logiciel dans le calculateur. Il est possible, au niveau des *runnables* (unités fonctionnelles d'exécution) de les mettre à jour de manière différentielle. Deux mises à jours sont possibles, l'*upgrade*, le remplacement d'une fonctionnalité existante par une nouvelle version et l'*update*, l'ajout de fonctionnalité initialement absente au sein de l'application.

Une application sur AUTOSAR *Classic Platform* est un ensemble de composants, appelés *SoftWare Component* (SWC) qui sont composés de *runnables*. Les deux types de communications, synchrone et asynchrone, sont disponibles sur AUTOSAR *Classic Platform*. Bien que plusieurs éléments soient a priori disponibles pour projeter nos approches sur le standard, les travaux précédents ont montré le peu de capacité à implémenter et modifier des mécanismes de sûreté de fonctionnement dynamiquement. AUTOSAR *Classic Platform* est principalement utilisé pour embarquer des systèmes très critiques tels le contrôle moteur : **(1)** les mises à jour de ces systèmes sont très rares et **(2)** il n'est pas recommandé de faire de la mise à jour différentielle en-ligne sur les systèmes critiques automobiles selon les normes IEC 61508 [Bell 2006] et ISO 26262 [Birch 2013].

Nous allons quand même réfléchir à la possibilité d'implémenter nos deux approches sur le standard, pour les applications embarquées moins fermées et moins

critiques (P.ex. ASIL B). Notre analyse montre que les concepts proposés dans la thèse facilite la maintenance et l'évolution hors-ligne de FTM.

Notre première approche se base sur l'adaptation de composant qui sont substitués par d'autres composants équivalent. Nous pouvons donc construire nos mécanismes en projetant les composants définis dans la section 1.1 du Chapitre 2 sur des SWC. De plus, à l'aide des concepts d'actions de sûreté de fonctionnement (action de SdF) définis dans le Chapitre 3, nous pouvons projeter ces actions simples sur des *runnables* qui constituent nos SWC. Il est assez simple d'implémenter nos FTM à composants sur AUTOSAR *Classic Platform*. En ce qui concerne l'adaptation, ce n'est finalement qu'un enchaînement de modification de type *upgrade* pour passer d'un mécanisme à un autre. En effet, comme nos actions de SdF sont regroupées dans les composants *Before* et *After*, il suffit de modifier la "version" du composant pour passer d'un FTM à un autre. Il faut cependant prévoir à l'avance les communications propres à un mécanisme (envoie d'un *checkpoint* pour le PBR ou d'un message de synchronisation pour le LFR) et réserver ces espaces lors de la conception et de la génération du code. Une pré-réservation des espaces est strictement nécessaire a priori. En ce qui concerne la composition, il faut prévoir les *containers* pouvant accueillir les *runnables* qui seront rajoutés lors d'une *update*. Bien que ces modifications ne puissent pas être faites en ligne, notre première approche semble donc compatible avec AUTOSAR *Classic Platform*. Elle simplifie hors ligne la maintenance et l'évolutivité des logiciels de tolérance aux fautes.

La seconde approche est plus ardue à implémenter, notamment à cause de l'Ordonnanceur interne. En effet, les applications implémentées sur AUTOSAR *Classic Platform* demandent une prévision des *Worst Case Executation Time* (WCET), le calcul des périodicités et la prévision de l'ordre d'exécution des *runnables* et de leur communication. Dans notre première approche, nous concevons les composants avec peu d'actions élémentaires qui s'exécute avant ou après l'application. Dans la seconde approche, le FTM contient toutes les actions qui peuvent s'exécuter avant et après l'application, rendant le calcul des WCET plus difficile. Nous ne pouvons pas ainsi assurer que notre seconde approche est compatible avec AUTOSAR *Classic Platform*. Il faudrait une expérimentation sur cible complète pour la valider. Une analyse du WCET de la manipulation d'un graphe en ligne n'a pas été effectuée dans nos travaux.

Pour résumer, nous pouvons avancer que nos deux approches de mise en œuvre l'AFT semblent difficilement compatibles avec le standard automobile et ne sont pas recommandées par les normes de conceptions de systèmes critiques. Cependant, les systèmes embarqués de type ADAS considérés comme moins critiques sont amenés à évoluer et à être mis à jour. Il est impératif d'assurer la fiabilité de ces systèmes d'aide à la conduite dont la défaillance peut amener à des conséquences graves pour le conducteur [Finch 2009]. Notre approche apporte cependant une vision nouvelle pour la modification hors-ligne car elle favorise la *Separation of Concern* entre code applicatif et mécanismes de sûreté. En sacrifiant les transformations en-ligne, nous pouvons concevoir et implémenter au moins les FTM en suivant la première

approche.

1.2 Adaptation pour l'automobile : AUTOSAR *Adaptive Platform*

La mise à jour des systèmes moins critiques tels que les ADAS et le multimédia est devenue un enjeu économiques majeurs pour les constructeurs automobiles. L'arrivée de l'entreprise TESLA sur le marché automobile a poussé les autres constructeurs comme Renault-Nissan, Volkswagen ou encore BMW à proposer des services similaires à leur clients. Ainsi, le consortium AUTOSAR, à l'origine du premier standard, s'est penché sur la création d'un nouveau standard automobile permettant facilement la mise-à-jour des applications considérées moins critiques (ADAS, *Infotainment*).

1.2.1 Motivations

Les enjeux de la mise à jour à distance des systèmes est avant tout économique. Ils sont nombreux et nous nous attarderons sur deux qui concernent l'entreprise et le client. Même si les autres enjeux sont liés à la sécurité et la sûreté de fonctionnement, l'économie sur le plan financier reste toujours le moteur principal de cette technologie de mise-à-jour à distance (*Over-the-Air Update*).

Le premier enjeu économique est la possibilité d'ajouter des correctifs aux logiciels embarqués sans pour autant rappeler les voitures dans les garages. C'est un avantage économique non trivial à court, moyen et long terme. Il faut prendre en compte plusieurs aspects qui sont liés à l'image de marque de l'entreprise. En effet, le rappel de véhicule a un coût dû à la main d'œuvre nécessaire pour mettre à jour celui-ci mais la publicité négative qui entoure le rappel a un impact tout aussi important sur le chiffre d'affaire. Le coût pour Toyota en 2010 concernant le rappel massif concernant des problèmes d'accélération des véhicules est estimé à 2 milliards de dollars, somme incluant le coût de rappel et les ventes perdues suite au scandale [Times].

Le second enjeu économique concerne cette fois les clients des constructeurs automobiles. Comme l'explique Carlos Ghosn, ancien PDG du groupe Renault-Nissan-Mitsubishi au CES 2017, cet enjeu concerne les mises à jours à distance du logiciel embarqué qui sont une valeur ajoutée au véhicule à la revente. Au fur et à mesure qu'une voiture vieillit, sa valeur marchande baisse drastiquement. Cette baisse peut être compensée par l'ajout de fonctionnalité au véhicule. De plus, cela permet au client de faire évoluer sa voiture en fonction de ses goûts et de ses moyens.

Pour faciliter la mise à jour des systèmes embarqués, le consortium AUTOSAR s'est tourné vers une nouvelle approche non plus basée sur l'OS temps-réel dur OSEK/VDX mais sur des OS de type Linux.

1.2.2 La plate-forme AUTOSAR *Adaptive Platform*

Afin de continuer à répondre aux exigences définies par les nouvelles technologies et de répondre aux exigences du marché en constante évolution, le partenariat

AUTOSAR a développé un nouveau standard : la plate-forme AUTOSAR *Adaptive Platform* [Fürst 2016]. L'objectif était de réaliser des applications indépendantes les unes des autres d'une manière distribuée et adaptative, de renforcer l'interaction avec d'autres applications non-AUTOSAR, et de fournir la capacité de mettre à jour ces systèmes à distance.

L'architecture de la nouvelle plate-forme ressemble à celle de AUTOSAR *Classic Platform* dans le sens où il y a une séparation entre les applications et la couche basse contenant le système d'exploitation, les services ou encore les *drivers*. Cette architecture issue des travaux du consortium [Fürst 2016] et des travaux d'équipementiers [Oertel 2019] permet également une ségrégation entre l'architecture matérielle et logicielle.

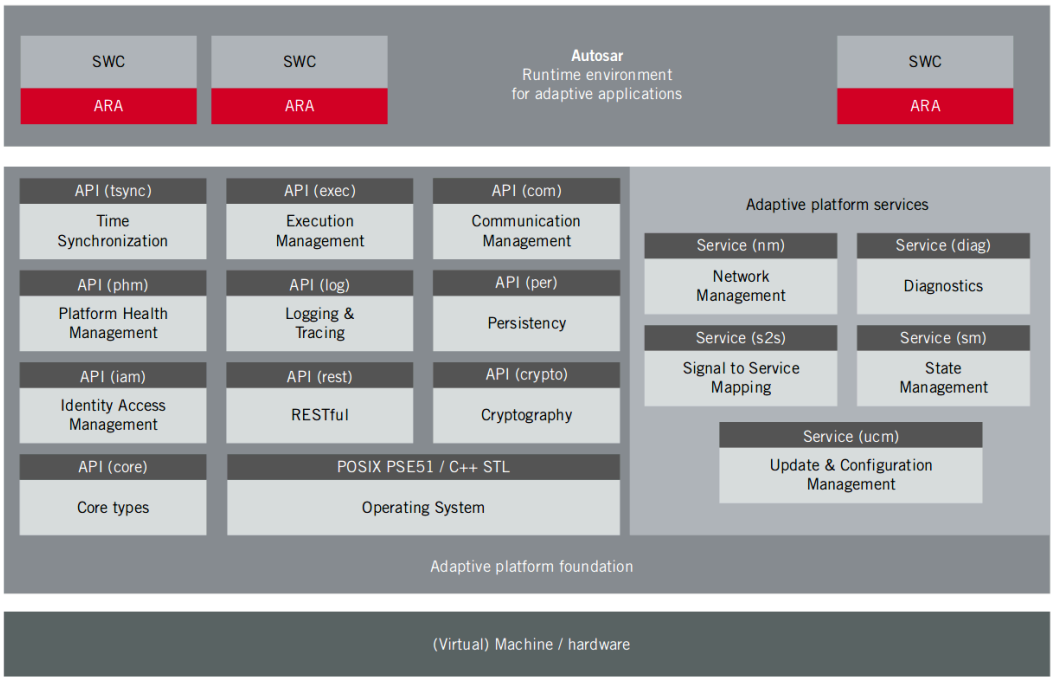


FIGURE 5.1 – Architecture de la plate-forme AUTOSAR *Adaptive Platform*

Comme l'illustre la figure 5.1, la plate-forme est divisée en trois parties. La couche applicative nommée ARA (*AUTOSAR Runtime for Adaptive Applications*) contient les applications réalisant les fonctions désirées sous forme de SWC. la couche basse se divise en deux sous ensembles, L'*Adaptive Plateform Foundation* qui contient les API, les fonctions élémentaires de la plate-forme (Gestion des communications, de l'exécution, observation de l'évolution du système), et L'*Adaptive Plateform Services* qui contient les services standards (Gestion de la configuration et des mises à jours, du réseau). Enfin, la couche matérielle qui peut être émulée à l'aide de machines virtuelles.

Comme nous pouvons le déduire de la couche basse, AUTOSAR *Adaptive Platform* a une architecture orientée service (SOA). C'est une architecture logicielle basée sur l'utilisation de services échangés entre les composants applicatifs et les

autres composants logiciels. Elle permet de réutiliser les services fournis par différentes applications, ce qui crée un moyen efficace et flexible d'interconnecter les systèmes. Cette approche permet d'augmenter l'évolutivité et la réutilisabilité du système et facilite simultanément la coexistence des différentes applications. De manière similaire à ROS, la plate-forme utilise un registre dans lequel les services sont enregistrés. Le client demande au registre la localisation du serveur puis transmet sa requête à ce dernier.

Les systèmes d'exploitation de cette plate-forme sont basés sur le standard IEEE POSIX [POS 2018] et plus précisément sur le profil PSE51 spécialement développé pour les systèmes embarqués temps-réels [POS 2004] et utilise un noyau Linux. Le standard vise à permettre la portabilité entre applications et la compatibilité entre différents systèmes d'exploitation. Ceci peut être réalisé en spécifiant des définitions communes sur les processus et la gestion des threads, la Communication Inter-Processus de base (IPC), et les gestionnaires de signaux et d'interruptions.

Enfin, AUTOSAR *Adaptive Platform* utilise plusieurs méthodes de communication, l'IPC [Rajkumar 1995] pour les communications locales, le DDS [Kang 2012] pour les communications avec le *cloud* et le SOME/IP [som] pour les communications inter-plate-forme. SOME/IP (*Scalable Service-Oriented Middleware over IP*) est un protocole de communication automobile qui prend en charge les appels de procédures à distance, les notifications d'événements et le format de sérialisation ainsi que de câblage sous-jacent. SOME/IP est spécifié par AUTOSAR pour s'adapter à des appareils divers utilisant différents systèmes d'exploitation. Ce protocole a pour but de répondre aux exigences des technologiques Ethernet dans l'industrie des systèmes embarqués automobiles. SOME/IP est basé sur un paradigme de communication orientée service (SOC) et permet aux applications d'être vues en tant que services sur le bus. Cela facilite la compatibilité des communications entre ECU en faisant abstraction de la plate-forme logicielle.

Nous avons pu appréhender les concepts de communication et les éléments constituants les différentes couches de la plate-forme AUTOSAR *Adaptive Platform*. Nous allons maintenant analyser le fonctionnement d'une application et de son adaptation sur cette plate-forme.

1.2.3 Adaptation fonctionnelle d'une application

Interessons nous d'abord aux différents services de la plate-forme nous permettant de mettre à jour une application. Nous allons donc analyser les fonctionnalités prévues pour ajouter une application et pour modifier cette application pendant l'exécution. Mais avant même de parler d'adaptation, il faut d'abord comprendre comment fonctionne le lancement et l'exécution d'une application sur AUTOSAR *Adaptive Platform*.

Une application est constituée de deux groupes d'éléments, les manifestes décrivant le graphe de composant et de communication (listing 5.1) et les exécutables associés. De plus, chaque ECU va posséder également un manifeste qui détermine les différentes applications qui doivent être lancées et définir ainsi la configura-

```
<SERVICE-INTERFACE>
  <SHORT-NAME>radar</SHORT-NAME>
  <EVENTS>
    <VARIABLE-DATA-PROTOTYPE>
      <SHORT-NAME>brakeEvent</SHORT-NAME>
      <TYPE-REF
      ↪  DEST="IMPLEMENTATION-DATA-TYPE">/events/RadarObjects</TYPE-REF>
    </VARIABLE-DATA-PROTOTYPE>
    <VARIABLE-DATA-PROTOTYPE>
      <SHORT-NAME>parkingBrakeEvent</SHORT-NAME>
      <TYPE-REF
      ↪  DEST="IMPLEMENTATION-DATA-TYPE">/events/RadarObjects</TYPE-REF>
    </VARIABLE-DATA-PROTOTYPE>
  </EVENTS>
  <FIELDS>
    <FIELD>
      <SHORT-NAME>UpdateRate</SHORT-NAME>
      <TYPE-REF DEST="IMPLEMENTATION-DATA-TYPE">/stdtypes/UInt32</TYPE-REF>
      <HAS-NOTIFIER>true</HAS-NOTIFIER>
      <HAS-GETTER>true</HAS-GETTER>
      <HAS-SETTER>true</HAS-SETTER>
    </FIELD>
  </FIELDS>
  <METHODES>
    <CLIENT-SERVER-OPERATION>
      <SHORT-NAME>Adjust</SHORT-NAME>
      <ARGUMENTS>
        <ARGUMENT-DATA-PROTOTYPE>
          <SHORT-NAME>target_position</SHORT-NAME>
          <TYPE-REF
          ↪  DEST="IMPLEMENTATION-DATA-TYPE">/methods/Position</TYPE-REF>
          <DIRECTION>IN</DIRECTION>
        </ARGUMENT-DATA-PROTOTYPE>
        <ARGUMENT-DATA-PROTOTYPE>
          <SHORT-NAME>success</SHORT-NAME>
          <TYPE-REF
          ↪  DEST="IMPLEMENTATION-DATA-TYPE">/stdtyoes/Boolean</TYPE-REF>
          <DIRECTION>OUT</DIRECTION>
        </ARGUMENT-DATA-PROTOTYPE>
      </ARGUMENTS>
    </CLIENT-SERVER-OPERATION>
  </METHODES>
</SERVICE-INTERFACE>
```

Listing 5.1 – Exemple de Manifeste définissant les Services d'un radar

tion de la machine. Les manifestes sont écrits en AUTOSAR XML (ARXML) [Sandmann 2012] et permettent à l'aide de balise de définir le type de service (Évènement, *Callback*) et le type de données utilisées. Sur l'exemple (listing 5.1), on

peut remarquer que le manifeste des services de l'application contient deux évènements, à savoir le freinage par la pédale (ligne 4 à 7) et le frein de parking (ligne 8 à 11), et au moins une méthode étant activée par message synchrone (ligne 26 à 35) contenant la position de la cible. La partie *Field* (ligne 14 à 21) définit le taux de rafraîchissement du radar.

À l'initialisation, une fois le système d'exploitation et les API de la couche basse lancés, l'API *Execution Management* (EM) [EM] va lire le manifeste de la machine, lire les manifestes des applications présentes et va lancer les différents exécutables liés au manifeste. Le rôle de cette API va donc être de manipuler le graphe de composant de son ECU en lançant et en arrêtant les différents composants d'une application. Si une application est distribuée sur plusieurs ECU, l'EM de chaque plate-forme va lancer ses composants. Pendant l'exécution, l'EM va consulter une autre API, l'API *Machine State Management* (MSM) qui contient le manifeste de la machine, et si la configuration logicielle de la machine a changé, l'EM va arrêter les SWC obsolètes et lancer ceux nécessaires à atteindre la nouvelle configuration. C'est donc le MSM qui définit à tout instant la configuration logicielle de l'ECU. La figure 5.2 illustre les interactions entre l'EM et le MSM et l'application. Ici, l'EM connaît trois états de l'application qui sont prédéfinis et qui vont être appliqués sur ordre du MSM.

Nous remarquons que l'EM et le MSM vont être les éléments importants pour mettre en exécution une adaptation du graphe de composant. Ainsi, si nous désirons procéder à une adaptation, il faut modifier la configuration logicielle de la machine contenue dans le MSM. Celui-ci va transmettre à l'EM le nouvel état et ainsi modifier le graphe de composant. Cette demande de modification de l'état de l'application peut-être déclenchée par une application de la plate-forme.

Lorsque le constructeur automobile veut installer une nouvelle application sur la plate-forme, mettre à jour une application déjà existante ou définir des états sûrs en cas de défaillance, celui-ci doit passer par le service *Update & Configuration Management* (UCM) qui a plusieurs rôle. En effet, l'UCM peut entre autre ajouter une application, mettre à jour les éléments de l'application, mettre à jour l'état de l'application.

Prenons un exemple simple avec un ADAS appelé *Traffic Jam Pilot*. Cet ADAS permet le suivi de ligne, le changement de ligne et la régulation de la vitesse dans une zone à fort trafic routier et à faible vitesse. Un constructeur automobile a mis au point un nouvel algorithme de suivi de ligne plus performant et moins coûteux en ressource. L'automobile connectée au serveur de mise à jour détecte ce changement, télécharge les paquets et l'UCM va gérer le remplacement des fichiers existants (manifestes et exécutables) par les nouveaux et définir les nouveaux états de l'application. Si maintenant, à cause d'une faute, le changement de ligne est défaillant, et cette défaillance est détectée par le moniteur, le MSM va transmettre à l'EM l'ordre de transition vers un état sûr prédéfini, ici la coupure de la fonction de changement de ligne ou l'arrêt complet de l'ADAS. Enfin, l'EM va arrêter les SWC qui doivent l'être.

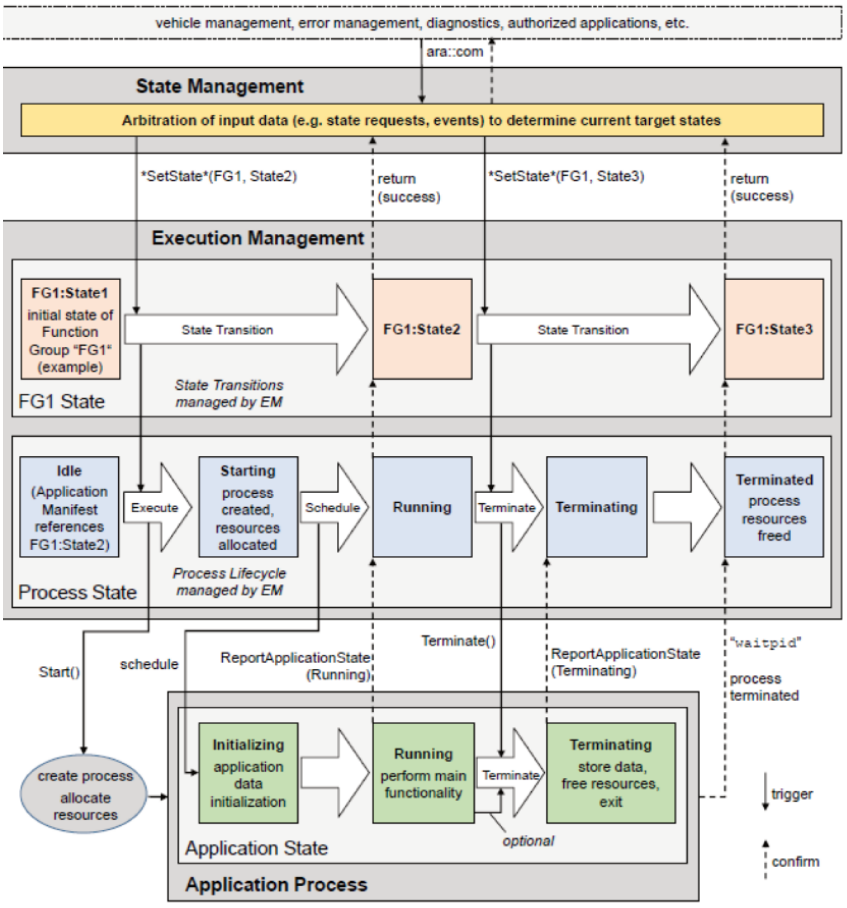


FIGURE 5.2 – Communications entre les services Adaptive AUTOSAR et l'application

Nous allons maintenant évaluer la compatibilité d'AUTOSAR *Adaptive Platform* avec l'AFT et analyser la faisabilité de projection de nos approches sur le standard.

2 Analyse de l'AFT sur AUTOSAR *Adaptive Platform*

Nous avons pu appréhender l'architecture et le fonctionnement de la nouvelle plate-forme AUTOSAR *Adaptive Platform* à travers l'exemple d'un manifeste et de l'architecture représentant les interactions entre le MSM, l'EM et une application. Ces documents que nous avons modifiés sont issus d'une formation de chez Renault Software Lab. Nous allons maintenant nous intéresser à la compatibilité de nos deux approches avec cette plate-forme et ainsi notre capacité à utiliser l'AFT avec les futurs systèmes embarqués automobiles évolutifs.

Actuellement, la plate-forme est toujours en développement, des éléments présentés précédemment vont peut être évoluer et tous les principes n'ont pas été complètement définis ou ne sont pas en version finale. Nous allons donc, au vu des documentations disponibles et des formations internes reçues, proposer la projection de nos deux approches sur AUTOSAR *Adaptive Platform*.

2.1 Projection de l'approche par composants substituables

2.1.1 Rappel de l'approche par composants substituables

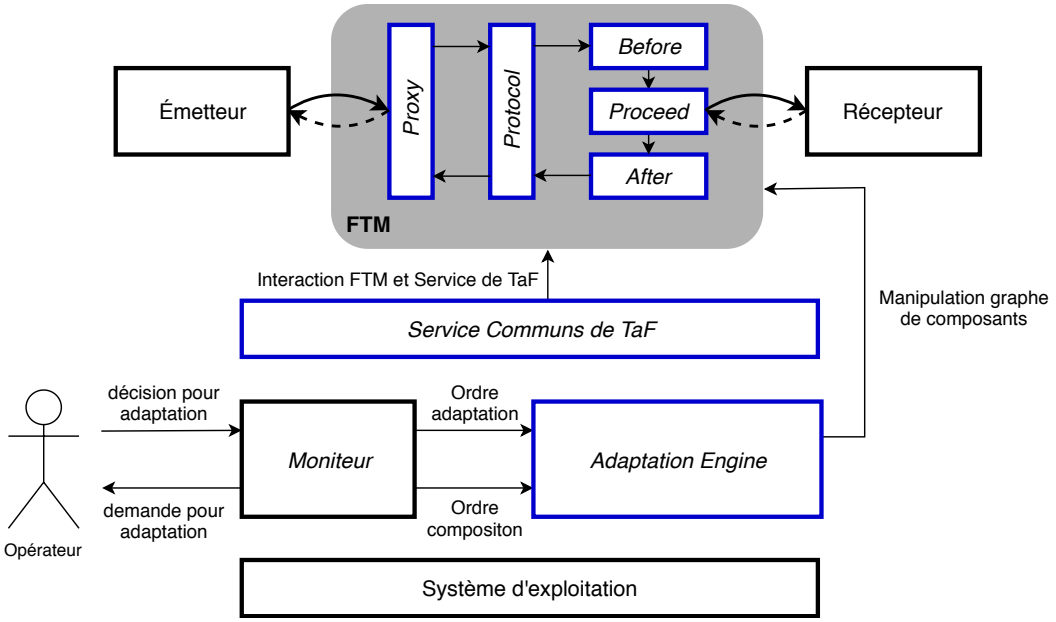


FIGURE 5.3 – Architecture globale d'un FTM à composant

Cette approche nous permet de concevoir nos FTM à l'aide de composants que nous pouvons manipuler sans impact sur l'application comme le rappelle la figure 5.3. Nous avons également défini, dans ce chapitre, trois types de transformations : **(1)** la mise à jour d'un composant (P.ex. Modifier l'état du Secondaire d'un PBR en cas de crash du primaire), **(2)** l'adaptation d'un mécanisme vers un autre (P.ex.

Transition d'un PBR vers un LFR) et **(3)** la composition de deux ou plusieurs mécanismes (P.ex. Transition d'un PBR vers un PBR + TR).

Pour réaliser cette approche, il faut effectuer un travail non trivial en amont pour valider les mécanismes mais aussi valider leur adaptation et leur composition. Une fois que les FTM ont été validés hors ligne en suivant les recommandations du standard de développement ISO26262, il faut être capable d'ajouter et de remplacer les composants installés à distance (P.ex. ajout d'un FTM non présent sur l'ECU), de changer l'état d'un composant en ligne **(1)**, de remplacer les composants en ligne **(2)** et de substituer un composant par un ensemble de composant **(3)** comme le montre la figure 5.4.

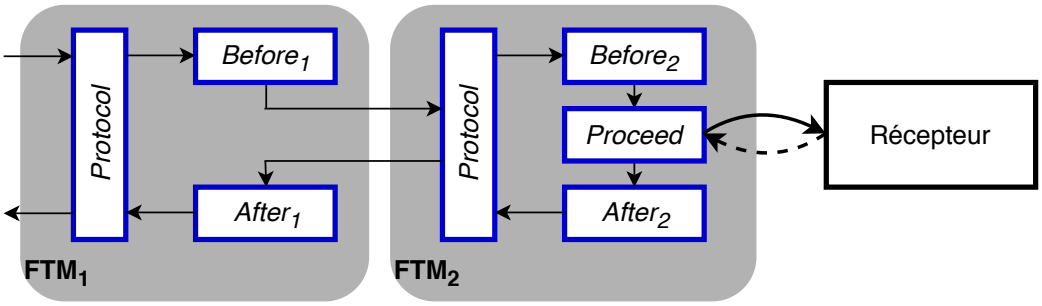


FIGURE 5.4 – Rappel de la composition de deux FTM

Sur ROS, nous avons dû instrumenter nos nœuds pour pouvoir modifier les communications lors d'une mise à jour d'un composant, par exemple, pour reconnecter le Client avec le Secondaire. De plus, nous avons un composant *Adaptation Engine* qui sert à manipuler le graphe de composants pour mettre en œuvre l'adaptation et la composition. Enfin, nous avons pu utiliser les fonctions de ROS pour pouvoir substituer un composant *Proceed* par un ensemble de composant *Protocol* - *Before* - *Proceed* - *After* et ainsi réaliser une composition.

2.1.2 Compatibilité de l'approche sur AUTOSAR *Adaptive Platform*

Pour analyser si cette approche est compatible avec AUTOSAR *Adaptive Platform*, il faut que le standard satisfasse des conditions fondamentales pour réaliser la projection :

1. **Contrôler l'état des composants :** Être capable de lancer, supprimer, arrêter et redémarrer nos composants.
2. **Manipuler les communications inter-composants :** Être capable de modifier, durant l'exécution, les canaux de communications entre les composants.
3. **Observer l'évolution du système :** Pouvoir détecter et analyser les modifications et les perturbations subies par le système.

Comme nous avons pu le remarquer dans la partie précédente, deux éléments sont au centre de la manipulation de l'état des ECU, l'API *Execution Management*

(EM) et le service *Machine State Management* (MSM). Les points **(1)** et **(2)** sont assurés par l'EM. En effet, ce composant est capable de contrôler l'état des SWC et de manipuler les communications à l'aide du middleware ARA avec la recherche dynamique de service. Le gros avantage d'AUTOSAR *Adaptive Platform* est la définition, à l'aide de manifestes, des composants formant une application et des communications entre les composants. Ainsi, on définit hors-ligne les transitions et les configurations logicielles pour assurer l'adaptation et la composition. Il est possible également de définir des configurations sûres de l'application en cas de perte complète du système.

La présence de l'EM et du MSM vont nous permettre de supprimer la différence entre la reconfiguration, l'adaptation et la composition grâce à la manipulation des états internes des SWC, du graphe de composants et de la dynamique des communications. Nous supposons la présence d'un moniteur pour observer le système et transmettre un ordre d'adaptation en cas de défaillance. Pour assurer l'observation du système **(3)**, l'API *Platform Health Management* (PHM) permet la supervision des applications (activité, échéance, logique de l'application), la vérification de la plate-forme (Test des RAM et ROM, surveillance de la tension et de la température). Idéalement, il faudrait associer notre *Adaptation Engine* avec le PHM ce qui n'est actuellement pas possible car le standard ne permet pas d'autres communications que celles prévues par le consortium.

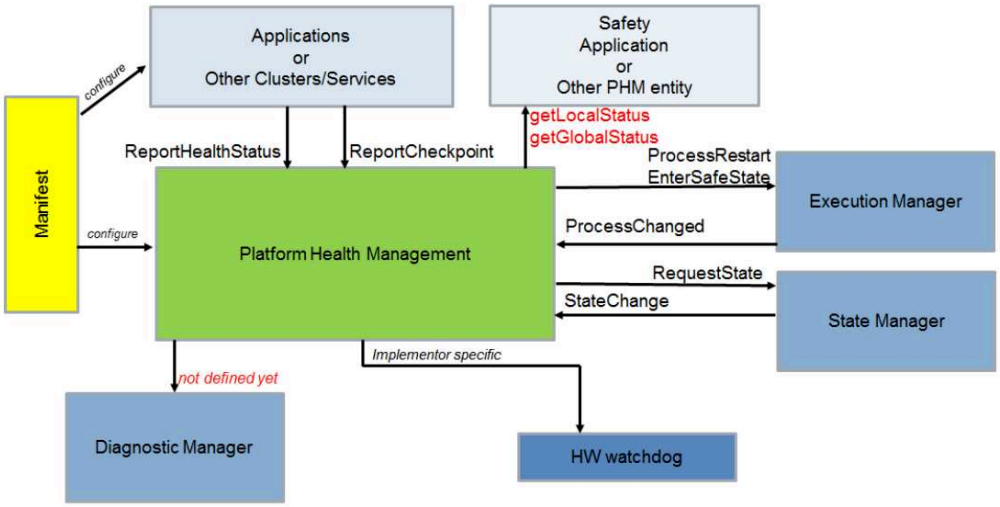


FIGURE 5.5 – Environnement du PHM

La figure 5.5 illustre l'environnement et les communications entre le PHM et les différentes API et services du standard. Toutes les communications et interactions entre le PHM et les autres API sont fixées. Nous ne pouvons donc pas déclencher une adaptation, à l'aide de l'*Adaptation Engine*, sur détection par le PHM d'un changement de contexte applicatif. Cependant, nous voyons que c'est le PHM qui émet la requête de changement de configuration vers le MSM. Il serait peut être possible d'inclure dans le PHM la sélection de mécanismes de l'*Adaptation Engine*.

La projection est donc théoriquement possible à condition de pouvoir transmettre un ordre du PHM vers notre *Adaptation Engine* pour déclencher une action d'adaptation ou de composition. Nous allons imaginer un scénario pour illustrer la mise en œuvre de l'approche. Reprenons un Client, le Serveur qui communiquent par message synchrone, le Primaire, et sa réplique, le Secondaire. À chaque Serveur est attaché un FTM, le *Primary Backup Replication*. Ce PBR est un FTM distribué sur deux ECU et chaque ECU contient un Protocol, un *Before*, un *Proceed* et un *After* et qui communiquent par l'intermédiaire d'un message asynchrone permettant l'envoi d'un checkpoint. De plus, deux *Watchdogs* surveillent l'état des ECU pour détecter le crash de leurs plate-formes respectives.

1er Scénario : L'ECU contenant le Primaire défaille par crash. Le *Watchdog* détecte la défaillance et appelle le service du MSM pour changer l'état du Secondaire et le connecter au Client. Le MSM utilise l'EM pour modifier l'état du Secondaire, état prévu à l'avance. Le Client fait appel au Registre des services pour localiser le Secondaire et la connexion est rétablie. Ici, les méthodes du composant de recouvrement *Recovery* sont distribuées entre l'EM, le MSM.

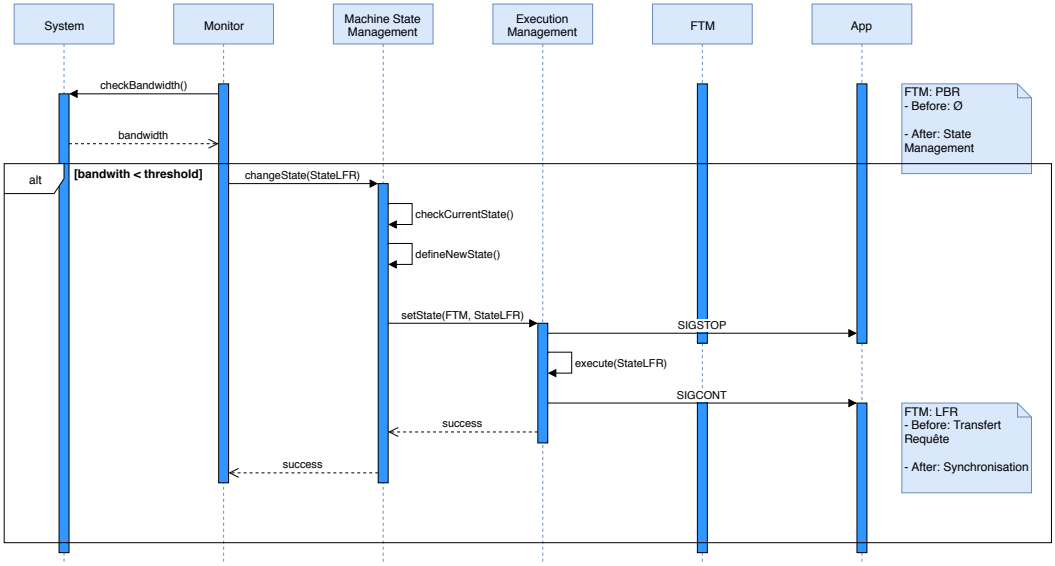


FIGURE 5.6 – Approche 1 : Diagramme de Séquence du 2ème Scénario

2ème Scénario : Une nouvelle application est activé sur l'ECU contenant le Primaire et diminue la bande-passante disponible pour le Serveur ce qui rend l'envoi d'un checkpoint impossible. Le Moniteur détecte la chute de bande-passante et envoie un ordre d'adaptation au MSM pour changer le graphe de composant du FTM. Le MSM utilise l'EM pour mettre en pause l'application, supprimer le *Before* et l'*After* du PBR et les remplacer par ceux du *Leader Follower Replication* (LFR). Ce scénario est représenté sur la figure 5.6.

3ème Scénario : L'ECU vieillissant sur lequel est installée l'application a des problèmes de *bitflip* intempestifs. Le Moniteur détecte le problème et signale au MSM pour changer le graphe de composant du FTM et composer le PBR avec un

Time Redundancy (TR). Le MSM utilise l'EM pour mettre en pause l'application, ajouter le *Before* et l'*After* du TR et modifier l'état interne des composants du PBR pour qu'ils communiquent avec les composants du TR. La différence avec le deuxième scénario va se situer au niveau de la définition de l'état transmis au MSM.

Ces trois scénarios sont théoriques et demanderaient une version finale de la plate-forme pour tester et valider l'approche sur le standard. Les méthodes de l'*Adaptation Engine* et du *Recovery* sont totalement distribuées sur les trois composants EM, MSM, PSM. Il est donc a priori possible de projeter notre approche par composants substituables sur AUTOSAR *Adaptive Platform*. Nous allons maintenant effectuer la même analyse avec l'approche du Chapitre 4, l'approche par actions ordonnançables.

2.2 Projection de l'approche par actions ordonnançables

2.2.1 Rappel de l'approche par composants substituables

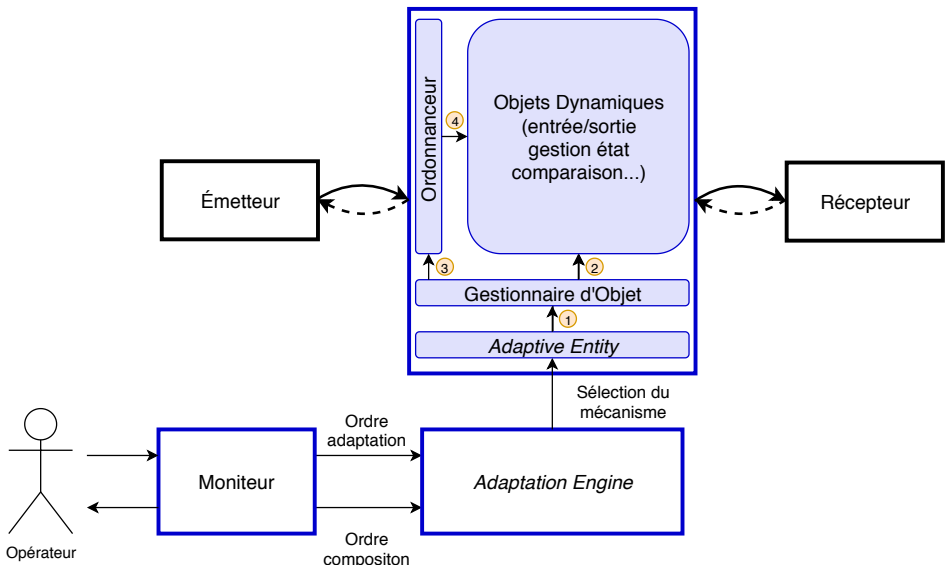


FIGURE 5.7 – Architecture globale d'un FTM à objets

Dans cette approche, il n'est plus question de manipuler un graphe de composant. Notre FTM est un composant unique constitué d'un graphe d'actions dynamiques qui sont manipulés et orchestrés en interne. Chacune de ses actions dynamiques contient une ou plusieurs actions de Sûreté de Fonctionnement (SdF). En parcourant le graphe d'action dans un certain ordre, on réalise le fonctionnement du FTM. En plus des objets dynamiques, il contient des objets statiques (qui ne sont pas compris dans le graphe d'actions) qui vont permettre de définir l'état du graphe d'actions (L'*Adaptive Entity*), son orchestration (l'Ordonnanceur) et sa manipulation (Le Gestionnaire d'Objet) comme le montre la figure 5.7.

Le travail à effectuer en amont pour mettre en œuvre cette approche est le même que pour l'approche précédente. Une différence majeure cependant réside dans la

granularité des manipulations. Ici, il faut pouvoir manipuler non plus des composants mais des actions. L'avantage de cette approche est une meilleure utilisation des ressources et une mise à jour différentielle à grain fin. Les inconvénients par rapport à l'autre approche restent la complexité de la mise en œuvre du graphe d'actions. Une synthèse de la comparaison entre nos deux approches est disponible dans la section 5 du Chapitre 4.

Sur ROS, nous avons utilisé les *plugins* pour projeter nos objets dynamiques. Un *plugin* peut être vu comme une extension des fonctionnalités d'une application que l'on peut instancier ou supprimer à volonté. Les *plugins* sont issus de classes mères abstraites contenant les fonctions virtuelles communes et sont contenus dans une bibliothèque. Par exemple, tous les *plugins* permettant la capture et la mise à jour de l'état sont issus d'une classe mère `GestionEtat` et sont regroupés dans une bibliothèque `libgestionetat`. L'avantage de l'utilisation des *plugins* est de ne charger dynamiquement en mémoire que les bibliothèques qui nous sont utiles et de pouvoir instancier n'importe quel *plugin* à partir de cette bibliothèque. Cela nous permet de modifier et compiler la bibliothèque contenant nos actions de SdF sans avoir à recompiler le reste (sauf en cas de modification des interfaces des méthodes abstraites et surchargées). Notre FTM n'est finalement qu'un conteneur comprenant les objets *Adaptive Entity*, le Gestionnaire d'Objet et l'Ordonnanceur et des plugins constituant le graphe d'actions.

Nous allons maintenant nous intéresser à la projection de cette approche sur AUTOSAR *Adaptive Platform* et comme précédemment définir les conditions essentielles à la mise en œuvre de l'approche. De plus, nous illustrerons cette projection avec les mêmes scenarii théoriques précédents.

2.2.2 Projection de l'approche sur AUTOSAR *Adaptive Platform*

Pour analyser si cette approche est compatible avec AUTOSAR *Adaptive Platform*, en plus de contrôler l'état des composants, les communications et d'observer l'évolution du système (*monitoring*), il faut que le standard satisfasse une condition supplémentaire pour réaliser la projection, **manipuler le graphe d'actions** ; c'est-à-dire être capable d'instancier et de supprimer dynamiquement les actions d'un composant.

Notre projection de l'approche par actions ordonnançables va dépendre de la version finale d'AUTOSAR *Adaptive Platform* et de la capacité à créer un graphe d'actions dans un *SoftWare Component* (SWC). Le standard utilise le C++ comme langage de programmation ce qui nous permet de conserver la création d'objets à partir de nos classes d'action de SdF. De plus, il est possible d'instancier nos objets issus de ces classes et de les associer à des *threads* [Barney 2009]. Il est donc a priori possible de créer un FTM contenant un graphe d'actions.

Néanmoins, un problème va se présenter au niveau de l'adaptation à grain fin. Nous n'avons pas la même liberté de manipulation des objets que sur ROS. Cependant, une solution permettant une adaptation limité est possible. AUTOSAR *Adaptive Platform* est un standard pour les systèmes embarqués automobiles qui

ont des ressources limitées. A l'inverse de ROS où nous ne nous soucions pas de l'allocation de ressources, ici il est recommandé de connaître et de limiter à l'avance les ressources allouées et donc de ne pas dynamiquement créer des *threads*. Cependant, nous pouvons définir à l'avance un nombre limité de *threads* auxquels nous attacherions les méthodes d'un certain nombre d'objet contenant les actions de SdF (notion de *thread pool* de CORBA [Pyarali 2001]. Il ne nous reste plus qu'à modifier le Gestionnaire d'Objets pour pouvoir changer les méthodes associées aux *threads*. Ainsi lors d'un changement d'état définit par le MSM, le Gestionnaire d'Objets peut modifier les *threads* et ainsi manipuler le graphe d'actions.

La projection est donc possible même si plus restreinte par rapport à son implémentation sur ROS. En effet, nous avons un nombre prédéterminé de *thread*. Nous ne pouvons donc pas instancier autant d'objets dynamiques que nous le souhaitons. Nous allons reprendre les deux premiers scenarii pour illustrer la mise en œuvre de l'approche. Reprenons le même exemple, à savoir un Client et un Serveur qui communiquent par message synchrone, le Primaire du PBR, et sa réplique, le Secondaire du PBR. Il contient un objet dynamique de Gestion d'État avec transmission de l'état par envoi de *checkpoint* et des objets de Synchronisation avec l'application. De plus, chaque ECU contient un *watchdog* pour détecter une défaillance par crash.

1er Scénario : L'ECU contenant le Primaire crash. Le *Watchdog* détecte la défaillance et appelle le service du MSM pour changer l'état du Secondaire et le connecter au Client. Le MSM utilise l'EM pour modifier l'état du Secondaire, ainsi ce ne sont plus les méthodes de réception de *checkpoint* et de mise à jour de l'état qui sont associées au *thread* mais les méthodes de capture d'état et de création de *checkpoint*. Ainsi le Secondaire devient le Primaire et le Client fait appel au Registre des services pour localiser le nouveau Primaire et la connexion est rétablie.

2nd Scénario : On remarque sur le diagramme de séquence de la figure 5.8 que la principale différence entre les deux approches va être l'activité du FTM. En effet, il faut que le FTM reste actif pour manipuler son graphe d'actions. L'utilisation des services du Moniteur est identique. Il faut cependant noter qu'ici, l'état de la machine au niveau graphe de SWC n'est pas modifié, seulement le graphe de communications l'est. De plus, l'EM communique avec la FTM. L'*Adaptive Entity* (Objet statique du FTM) récupère le nouvel état du graphe d'action qui est modifié par l'intermédiaire du Gestionnaire d'Objet. Ce dernier recrée la table d'orchestration puis l'Ordonnanceur la récupère.

Cette approche est a priori possible mais demanderait une implémentation rigoureuse des concepts que nous avons conçus pour valider complètement la projection de cette approche sur AUTOSAR *Adaptive Platform*. La difficulté ici est de savoir si nous pouvons facilement modifier les objets dynamique et ainsi le graphe d'actions. De plus, nous devons étudier la possibilité de définir au niveau du MSM et de l'EM les différentes configurations logicielles, les états des SWC, ainsi que les transitions.

Une autre possibilité est une solution mixte entre nos deux approches. Nous sacrifions l'adaptation à grain fin mais gardons le principe du FTM contenant un graphe d'actions. Chaque FTM serait constitué d'un graphe d'actions statique dé-

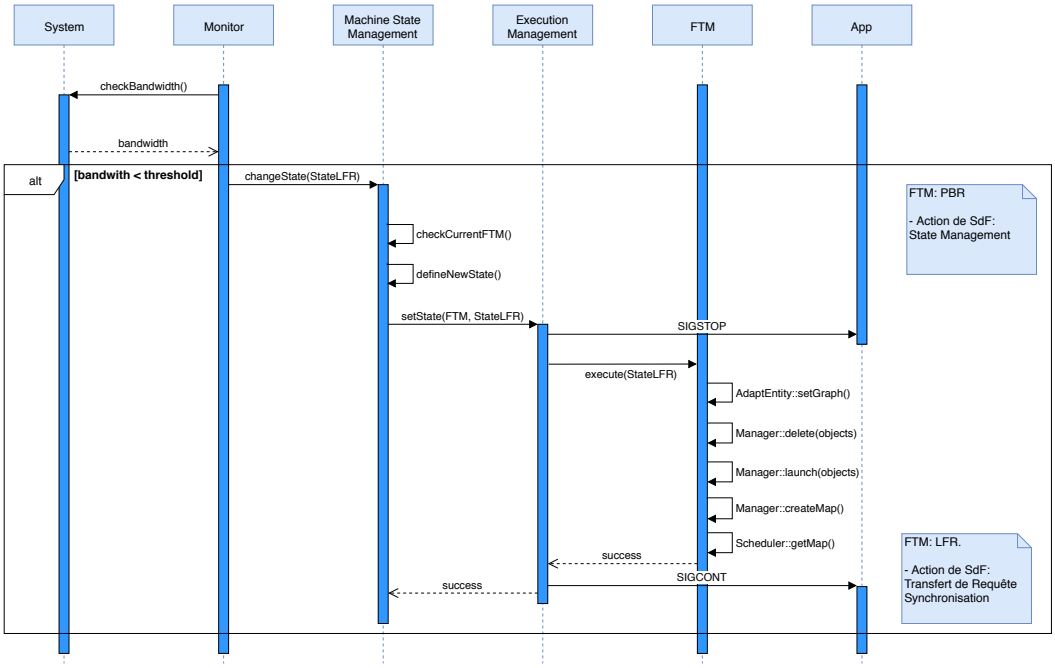


FIGURE 5.8 – Approche 2 : Diagramme de Séquence du 2nd Scénario

finit à l'avance. L'adaptation consisterait à substituer ce composant par celui du nouvel FTM à mettre en place. Par exemple, un PBR serait toujours constitué de deux composants, le Primaire et le Secondaire. Pour passer du Primaire au LFR, nous ne modifierions plus le graphe d'actions des deux composants mais les substituerions avec les deux composants du LFR. C'est néanmoins un sacrifice majeur du *Design for Adaptation* car nous perdons et l'architecture modulaire de la première approche et la mise-à-jour différentielle à grain fin de la seconde.

Nous avons à présent étudié les projections de nos approches à la fois sur AUTOSAR *Classic Platform* et AUTOSAR *Adaptive Platform*. En Août 2019, au vu des prévisions du consortium, la plate-forme n'était qu'à environ 60% de son développement final. Il sera intéressant, une fois la plate-forme finie et validée, d'expérimenter nos deux approches et de vérifier à l'aide de nos scenarii si les réserves que nous avons émises sont justifiées ou non. Nous allons conclure ce chapitre en effectuant une synthèse récapitulative des éléments clefs des deux standards et les comparer aux éléments nécessaires à la mise en œuvre des deux approches.

3 Synthèse et perspective pour l'automobile

Ce cinquième et dernier chapitre s'est concentré sur l'application de nos deux approches aux systèmes embarqués automobiles sur les deux standards du consortium AUTOSAR. Nous nous sommes intéressés aux motivations des constructeurs automobile concernant l'évolutivité de leurs systèmes et amenant à la conception et au développement du standard AUTOSAR *Adaptive Platform*. Nous avons pu également analyser l'architecture logicielle actuelle du nouveau standard et analyser les éléments qui sont utiles pour mettre en œuvre l'*Adaptive Fault Tolerance*.

	AUTOSAR *Classic Platform*	**AUTOSAR *Adaptive Platform***	**ROS**
Système d'exploitation	OSEK/VDX	POSIX PSE51	Ubuntu 18.1
Unité d'exécution	Tâche	Processus UNIX	Processus Unix
Lancement unité d'exécuction	Génération de code Model-based design	Manifeste XML	Commande Shell Manifeste XML
Gestion état des applications	*ECU State Management*	*Machine State Management* *Execution Management*	*ROS Master*
Type de Comminucation	CAN, FlexRay...	SOME/IP, DDS, IPC	TCP/IP

TABLE 5.1 – Comparaison des supports d'exécution

Le tableau 5.1 met en comparaison les éléments des couches basses qui vont être essentiels à la réalisation de l'AFT. On remarque beaucoup de similitude entre AUTOSAR *Adaptive Platform* et ROS concernant les unités d'exécution et leur gestion. Les deux supports utilisent des manifestes XML (fichier json pour le premier, fichier launch pour le second). Nous avons également rappelé les éléments de AUTOSAR *Classic Platform* mais, pour les raisons évoquées précédemment, nous ne l'utiliserons pas pour le comparatif des supports d'exécution concernant la projection de nos deux approches.

	Adaptive AUTOSAR	**ROS**
Projection d'un FTM	Ensemble de SWC	Ensemble de nœuds
Projection d'un composant	Processus UNIX	Processus Unix
Communication inter-composant	Service AUTOSAR Synchrone et Asynchrone	Topic/Service
Ordonnancement des composants	Ordonnanceur de l'OS Évènement par message	Ordonnanceur de l'OS Évènement par message
Adaptation & Composition	Service Machine State Management Service Execution Management Signaux UNIX	signaux UNIX commande shell roskill & roslaunch Adaptation Engine
Mise à jour à distance	Service Update and Configuration Management	Modification des exécutables & des Manifestes manuelle

TABLE 5.2 – Approche 1 : Comparaison entre ROS et Adaptive AUTOSAR

Le tableau 5.2 montre une comparaison des communications, de l'utilisation des services et des API utilisées dans ROS, dans AUTOSAR *Adaptive Platform* ainsi que les éléments que nous avons créés pour implémenter l'approche par composants substituables. Il est intéressant de noter que cette approche implémentée sous ROS

trouve facilement un parallèle sur le nouveau standard. L'API *Execution Manager* utilise également les signaux UNIX (SIGSTOP, SIGCONT, SIGTERM) pour gérer l'état des composants du système. Ce tableau met finalement en lumière la forte compatibilité de l'implémentation de cette approche pour le contexte automobile.

	AUTOSAR *Adaptive Platform*	**ROS**
Projection d'un FTM	Un SWC, ensemble de thread (nombre prédéterminé)	Un nœud, ensemble de plugins (nombre non-prédéterminé)
Projection d'une action de SdF	thread	plugin ROS
Communication inter-composant	Service AUTOSAR	Topic/Service
Communication intra-composant	Variable partagée	Variable partagée
Ordonnancement des composants	Ordonnanceur de l'OS + Ordonnanceur interne	Évènement par message + Ordonnanceur interne
Adaptation **& Composition**	Service Machine State Management Service Execution Management Adaptive Entity Gestionnaire d'Objet	signaux UNIX Adaptive Entity Gestionnaire d'Objet
Mise à jour à distance	Service Update and Configuration Management	Modification des exécutables & des Manifestes manuelle

TABLE 5.3 – Approche 2 : Comparaison entre ROS et Adaptive AUTOSAR

Le tableau 5.3 montre une comparaison des communications, de l'utilisation des services et des API utilisées dans ROS et dans AUTOSAR *Adaptive Platform* concernant l'implémentation de l'approche par actions ordonnançables. La principale difficulté dans l'approche est l'implémentation des objets contenant des actions de SdF. Dans ROS, elles sont implémentés sur des *plugins*, des extensions de fonctionnalité que l'on peut manipuler facilement. Sur AUTOSAR *Adaptive Platform*, il faut instancier les objets et les associer à des *threads*. Pour l'instant, nous ne pouvons pas tester la manipulation de ces *threads* sur une plate-forme complète. S'il est possible de modifier les objets liés aux *threads* pendant l'exécution pour adapter et composer nos FTM alors l'implémentation de l'approche est possible.

Nous avons montré dans ce chapitre qu'AUTOSAR *Adaptive Platform* semble cependant être un standard intéressant pour l'implémentation de l'AFT pour les systèmes embarqués automobiles évolutifs. Les API et services proposés pour la modification du graphe de composants et des communications, et leur mise à jour à distance est une avancée majeure face à ce que proposait le standard AUTOSAR *Classic Platform*. Beaucoup de travail reste cependant à effectuer pour valider l'implémentation de nos approches sur ce support, mais cela doit se faire dans un cadre industriel et sur une plate-forme et une étude de cas opérationnelles représentatives.

Conclusion

Avec l'arrivée des voitures connectées et de la voiture autonome, assurer la fiabilité, la maintenabilité et la disponibilité est devenu un enjeu majeur pour les constructeurs automobiles en dépit de l'évolution permanente des systèmes à bord. La Tolérance aux Fautes Adaptative (AFT) est un des moyens pour permettre à un système d'être résilient fasse à des perturbations, à savoir des changements non prévus dans le contexte applicatif.

Dans cette thèse, deux approches avec des granularités différentes, ont été proposées pour assurer la résilience du système à travers l'adaptation des mécanismes de tolérance aux fautes (FTM). Bien que l'AFT fasse partie des solutions possibles depuis quelque temps déjà, les solutions existantes ne permettent pas une modification en ligne des FTM. Cela n'est pas satisfaisant pour les systèmes à longue durée de vie qui fonctionnent dans des environnements très dynamiques, car il est impossible de prévoir tous les changements qui pourraient survenir pendant leur durée de vie. Par conséquent, nous nous sommes appuyé dans un premier temps sur une approche par composants substituables, projetant les FTM sur un ensemble de trois composants (*Before* - *Proceed* - *After* ou BPA et une approche par actions ordonnançables, projetant les FTM sur un ensemble d'objets dynamiques contenant les actions de SdF qui sont manipulés et ordonnancés à l'intérieur du FTM. Enfin, nous avons fait une analyse critique du standard automobile AUTOSAR *Adaptive Platform*, en cours de développement à ce jour, et qui est censé faciliter la mise à jour et l'adaptation des systèmes embarqués automobiles.

La première approche conçoit les FTM comme des graphes de composants et propose des méthodes pour manipuler se graphe pendant l'exécution. Cette approche repose sur le principe de *Separation of Concern* (SoC), séparant l'application fonctionnelle des FTM. Les FTM sont choisis en fonction d'hypothèses sur le contexte applicatif, à savoir le modèle de fautes FT, les caractéristiques applicatives AC, et les besoins en ressources RS. Le FTM est ensuite projeté sur trois composants, le *Before* pour les actions de SdF en amont des actions fonctionnelles, le *Proceed* qui sert d'interface avec l'application, l'*After* pour les actions de SdF en aval des actions fonctionnelles et le *Protocol* qui sert de râtelier à ces trois composants. L'avantage de ce schéma de conception est de pouvoir transiter d'un mécanisme à un autre en ne substituant que les composants nécessaires. Ce schéma est appelé le *Design for Adaptation* (D4A) permet une manipulation à grain fin des composants. De plus nous avons conçu l'*Adaptation Engine*, un composant qui manipule le graphe de composant lorsqu'une transition (adaptation, composition ou reconfiguration) est nécessaire et que le FTM en place n'est plus valide dû à un changement de contexte applicatif et/ou de ses spécifications non fonctionnelles. Nous avons implémenté notre approche sur ROS (*Robot Operating System*), un *middleware* fonctionnant sur une distribution Linux. Ce *middleware* permet l'implémentation de nos FTM et l'application des méthodes pour assurer l'AFT. Il n'est pas parfait car il n'a

pas d'API dédiée à la manipulation dynamique des communications en ligne et a un point de défaillance unique, le *ROS Master*. Ce *ROS Master* est un processus unique qui connaît le graphe de composants et qui gère la liaison des communications inter-composants. Il n'est donc pas envisageable pour une industrialisation de notre approche mais reste un bon support d'exécution pour appréhender l'AFT. Il est en outre possible de rendre le *ROS Master* redondant en utilisant des approches à base de *checkpointing* de processus UNIX comme les travaux de Gene Cooperman à NWU Boston [Ansel 2009]

Le problème avec cette approche est que les composants sont projetés à l'exécution sur des unité d'exécutions (P.ex. processus UNIX) ce qui rend les FTM gourmands en ressource. De plus la granularité ne semble pas assez fine car une adaptation signifie la substitution d'un composant complet contenant plusieurs actions de SdF. Entre autre, des composants utilisé dans certains FTM pourraient être réutilisé dans d'autres. Nous avons fait une analyse critique de plusieurs stratégies de tolérance aux fautes et des implémentations FTM pour en sortir des Catégories d'actions de SdF qui sont spécifiées en sous-catégorie en fonction de choix de conception et d'implémentation.

Notre seconde approche conçoit les FTM comme des graphes de actions de SdF réuni à l'intérieur d'un composant et propose des méthodes pour ordonnancer ces actions et manipuler le graphe pendant l'exécution. Cette approche repose également sur le principe de *Separation of Concern* (SoC), séparant l'application fonctionnelle des FTM. Les FTM sont toujours choisis en fonction d'hypothèses sur le contexte applicatif. Cependant, une fois le FTM sélectionné, les sous-catégorie d'actions de SdF sont projetés sur des objets dynamiques. Ce objets dynamiques sont ordonnancés par un *Scheduler* interne au FTM qui contient une table d'orchestration regroupant les méthodes des objets. Ces objets sont manipulés grâce à un *Object Manager*. L'avantage de cette approche est d'affiner la granularité de nos transitions en ne substituant plus des composants contenant plusieurs actions mais en modifiant les objets dynamiques et en modifiant l'ordonnancement des méthodes qu'ils contiennent. Pour modifier un action de sélection de donné (moyenne, comparaison, vote), nous pouvons juste modifier l'objet qui contient ces méthodes et de modifier la table d'orchestration du *Scheduler*. Ici, une transition ne signifie plus substituer un composant mais substituer un objet dynamique et mettre à jour la table d'orchestration. Nous avons implémenté notre approche sur ROS (*Robot Operating System*), et utilisé les *plugins*, des objets qui peuvent être chargés et instanciés pendant l'exécution. Nous avons, grâce à cette méthode, affiné le niveau des transitions entre FTM, diminué la consommation des ressources et augmenté la réutilisation de code.

Enfin, nous avons fait une analyse critique du standard AUTOSAR *Adaptive Platform*. Nous avons étudié la compatibilité de nos approches sur ce standard et comparé les points communs et divergents avec leur implémentation sur ROS. AUTOSAR *Adaptive Platform* est intéressant pour l'implémentation de l'AFT pour les systèmes embarqués automobiles évolutifs. Les API et services proposés pour la

modification du graphe de composants et des communications, et leur mise à jour à distance est une avancée majeure face à ce que proposait le standard AUTOSAR *Classic Platform*. Beaucoup de travail reste cependant à effectuer pour valider l'implémentation de nos approches sur ce support, mais cela doit se faire dans un cadre industriel et sur une plate-forme et une étude de cas opérationnelles représentatives.

Bibliographie

[Aeberhard 2015] MICHAEL Aeberhard, T Kühbeck, B Seidl, M Friedl, J Thomas et O Scheickl. *Automated Driving with ROS at BMW*. ROSCon 2015 Hamburg, Germany, 2015. (Cité en pages 25 et 53.)

[Ansel 2009] Jason Ansel, Kapil Arya et Gene Cooperman. *DMTCP : Transparent checkpointing for cluster computations and the desktop*. Dans 2009 IEEE International Symposium on Parallel & Distributed Processing, pages 1–12. IEEE, 2009. (Cité en pages 73, 83 et 166.)

[Armoush 2010] Ashraf Armoush. *Design patterns for safety-critical embedded systems*. PhD thesis, RWTH Aachen University, 2010. (Cité en page 88.)

[AUTOSAR] AUTOSAR. *Consortium AUTOSAR*. `https://www.autosar.org/`. Accessed : 2019-03-15. (Cité en page 1.)

[Avizienis 1995] Algirdas Avizienis. *The methodology of n-version programming*. Software fault tolerance, vol. 3, pages 23–46, 1995. (Cité en page 95.)

[Avizienis 2004] Algirdas Avizienis, J-C Laprie, Brian Randell et Carl Landwehr. *Basic concepts and taxonomy of dependable and secure computing*. IEEE transactions on dependable and secure computing, vol. 1, no. 1, pages 11–33, 2004. (Cité en page 8.)

[Baldessari 2007] Roberto Baldessari, Bert Bödekker, Matthias Deegener, Andreas Festag, Walter Franz, C Christopher Kellum, Timo Kosch, Andras Kovacs, Massimiliano Lenardi, Cornelius Menig*et al.* *Car-2-car communication consortium-manifesto*. 2007. (Cité en page 30.)

[Barney 2009] Blaise Barney. *POSIX threads programming*. National Laboratory. Disponível em :< https ://computing. llnl. gov/tutorials/pthreads/> Acesso em, vol. 5, page 46, 2009. (Cité en page 160.)

[Bechennec 2006] Jean-Luc Bechennec, Mikael Briday, Sébastien Faucou et Yvon Trinquet. *Trampoline an open source implementation of the osek/vdx rtos specification*. Dans 2006 IEEE Conference on Emerging Technologies and Factory Automation, pages 62–69. IEEE, 2006. (Cité en page 34.)

[Belaggoun 2016] Amel Belaggoun et Valerie Issarny. *Towards adaptive autosar : a system-level approach*. 2016. (Cité en page 147.)

[Bell 2006] Ron Bell. *Introduction to IEC 61508*. Dans Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55, pages 3–12. Australian Computer Society, Inc., 2006. (Cité en pages 27 et 147.)

[Birch 2013] John Birch, Roger Rivett, Ibrahim Habli, Ben Bradshaw, John Botham, Dave Higham, Peter Jesty, Helen Monkhouse et Robert Palin. *Safety cases and their role in ISO 26262 functional safety assessment*. Dans International Conference on Computer Safety, Reliability, and Security, pages 154–165. Springer, 2013. (Cité en page 147.)

[Blair 2004] Gordon Blair, Geoff Coulson, Jo Ueyama, Kevin Lee et Ackbar Joolia. *Opencom v2 : A component model for building systems software*. IASTED software engineering and applications, 2004. (Cité en page 18.)

[Blair 2009] Gordon Blair, Thierry Coupaye et Jean-Bernard Stefani. *Component-based architecture : the Fractal initiative*. annals of telecommunications-annales des telecommunications, vol. 64, no. 1-2, pages 1–4, 2009. (Cité en page 18.)

[Bo 2010] Huang Bo, Dong Hui, Wang Dafang et Zhao Guifan. *Basic concepts on AUTOSAR development*. Dans 2010 International Conference on Intelligent Computation Technology and Automation, volume 1, pages 871–873. IEEE, 2010. (Cité en page 32.)

[Boudreau 1971] Paul E Boudreau et Robert F Steen. *Cyclic redundancy checking by program*. Dans Proceedings of the November 16-18, 1971, fall joint computer conference, pages 9–15. ACM, 1971. (Cité en page 90.)

[Braden 1989] Robert Braden, David Borman et Craig Partridge. *Computing the internet checksum*. ACM SIGCOMM Computer Communication Review, vol. 19, no. 2, pages 86–94, 1989. (Cité en page 90.)

[Broy 2005] Manfred Broy. *Automotive software and systems engineering*. Dans Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2005. MEMOCODE'05., pages 143–149. IEEE, 2005. (Cité en page 30.)

[Broy 2006] Manfred Broy. *Challenges in automotive software engineering*. Dans Proceedings of the 28th international conference on Software engineering, pages 33–42. ACM, 2006. (Cité en page 30.)

[Bunzel 2011] Stefan Bunzel. *Autosar–the standardized software architecture*. Informatik-Spektrum, vol. 34, no. 1, pages 79–83, 2011. (Cité en page 32.)

[Chérèque 1992] Marc Chérèque, David Powell, Philippe Reynier, J-L Richier et Jacques Voiron. *Active replication in Delta-4*. Dans [1992] Digest of Papers. FTCS-22 : The Twenty-Second International Symposium on Fault-Tolerant Computing, pages 28–37. IEEE, 1992. (Cité en page 89.)

[Coekaerts 2008] Wim A Coekaerts. *Heartbeat mechanism for cluster systems*, novembre 11 2008. US Patent 7,451,359. (Cité en page 86.)

[Crnkovic 2011] Ivica Crnkovic, Severine Sentilles, Aneta Vulgarakis et Michel RV Chaudron. *A classification framework for software component models*. IEEE Transactions on Software Engineering, vol. 37, no. 5, pages 593–615, 2011. (Cité en page 18.)

[Cui 2019] Ang Cui. *Embedded systems monitoring systems and methods*, juin 4 2019. US Patent 10,311,232. (Cité en page 47.)

[Defago 1998] Xavier Defago, Andre Schiper et Nicole Sergent. *Semi-passive replication*. Dans Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems (Cat. No. 98CB36281), pages 43–50. IEEE, 1998. (Cité en page 90.)

[Delgado 1999] Joaquin Delgado et Naohiro Ishii. *Memory-based weighted majority prediction*. Dans SIGIR Workshop Recomm. Syst. Citeseer, 1999. (Cité en page 85.)

[Dijkstra 1982] Edsger W Dijkstra. *On the role of scientific thought*. Dans Selected writings on computing : a personal perspective, pages 60–66. Springer, 1982. (Cité en page 12.)

[Douglass 1999] Bruce Powel Douglass. Doing hard time : developing real-time systems with uml, objects, frameworks, and patterns, volume 1. Addison-Wesley Professional, 1999. (Cité en page 88.)

[Douglass 2003] Bruce Powel Douglass. Real-time design patterns : robust scalable architecture for real-time systems. Addison-Wesley Professional, 2003. (Cité en page 88.)

[Elrad 2001] Tzilla Elrad, Robert E Filman et Atef Bader. *Aspect-oriented programming : Introduction*. Communications of the ACM, vol. 44, no. 10, pages 29–32, 2001. (Cité en page 17.)

[EM] *Specification on SOME/IP Transport Protocol*. `https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-10/AUTOSAR_SWS_ExecutionManagement.pdf`. Accessed : 2019-10-01. (Cité en page 153.)

[Excoffon 2016] William Excoffon, Jean-Charles Fabre et Michael Lauer. *Towards modelling adaptive fault tolerance for resilient computing analysis*. Dans International Conference on Computer Safety, Reliability, and Security, pages 159–171. Springer, 2016. (Cité en pages 14 et 44.)

[Fabre 1995] Jean-Charles Fabre, Vincent Nicomette, Tanguy Pérennou, Robert Stroud et Zhixue Wu. *Implementing fault-tolerant applications using reflective object-oriented programming*. Dans Predictably Dependable Computing Systems, pages 189–208. Springer, 1995. (Cité en page 90.)

[Finch 2009] Joel Finch. *Toyota sudden acceleration : a case study of the national highway traffic safety administration-recalls for change*. Loy. Consumer L. Rev., vol. 22, page 472, 2009. (Cité en page 148.)

[Fürst 2009] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa et Klaus Lange. *AUTOSAR–A Worldwide Standard is on the Road*. Dans 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, volume 62, page 5, 2009. (Cité en page 31.)

[Fürst 2010] Simon Fürst. *Challenges in the design of automotive software*. Dans Proceedings of the Conference on Design, Automation and Test in Europe, pages 256–258. European Design and Automation Association, 2010. (Cité en page 32.)

[Fürst 2016] S. Fürst et M. Bechter. *AUTOSAR for Connected and Autonomous Vehicles : The AUTOSAR Adaptive Platform*. Dans 2016 46th Annual

IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), pages 215–217, June 2016. (Cité en pages 2 et 150.)

[Gallager 1962] Robert Gallager. *Low-density parity-check codes.* IRE Transactions on information theory, vol. 8, no. 1, pages 21–28, 1962. (Cité en page 90.)

[Geib 1997] Jean Marc Geib. Corba des concepts a la pratique. Ed. Techniques Ingénieur, 1997. (Cité en page 18.)

[Goel 1979] Amrit L Goel. *Software error detection model with applications.* Journal of Systems and Software, vol. 1, pages 243–249, 1979. (Cité en page 85.)

[Griffin 1990] Robert W Griffin et James P Emmond. *Exactly-once semantics in a TP queuing system*, août 14 1990. US Patent 4,949,251. (Cité en page 115.)

[Guettier 2016] C Guettier, B Bradai, F Hochart, P Resende, J Yelloz et A Garnault. *Standardization of generic architecture for autonomous driving : A reality check.* Dans Energy Consumption and Autonomous Driving, pages 57–68. Springer, 2016. (Cité en page 25.)

[Hanmer 2013] Robert S Hanmer. Patterns for fault tolerant software. John Wiley & Sons, 2013. (Cité en page 88.)

[Harding 2014] John Harding, Gregory Powell, Rebecca Yoon, Joshua Fikentscher, Charlene Doyle, Dana Sade, Mike Lukuc, Jim Simons, Jing Wang*et al. Vehicle-to-vehicle communications : readiness of V2V technology for application.* Rapport technique, United States. National Highway Traffic Safety Administration, 2014. (Cité en page 30.)

[Hatton 2007] Les Hatton. *Language subsetting in an industrial context : A comparison of MISRA C 1998 and MISRA C 2004.* Information and Software Technology, vol. 49, no. 5, pages 475–482, 2007. (Cité en page 27.)

[Heineman 2001] George T Heineman et William T Councill. *Component-based software engineering.* Putting the pieces together, addison-westley, page 5, 2001. (Cité en page 18.)

[Irwin 1997] John Irwin, Jean-Marc Loingtier, John R Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar et Tatiana Shpeisman. *Aspect-oriented programming of sparse matrix code.* Dans International Conference on Computing in Object-Oriented Parallel Environments, pages 249–256. Springer, 1997. (Cité en page 17.)

[Kang 2012] Woochul Kang, Krasimira Kapitanova et Sang Hyuk Son. *RDDS : A real-time data distribution service for cyber-physical systems.* IEEE Transactions on Industrial Informatics, vol. 8, no. 2, pages 393–405, 2012. (Cité en page 151.)

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier et John Irwin. Ecoop'97 — object-oriented programming : 11th european conference jyväskylä, finland, june 9–13, 1997 proceedings, chapitre Aspect-oriented programming, pages 220–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. (Cité en page 17.)

[Kim 1990] KH Kim et Thomas F Lawrence. *Adaptive fault tolerance : Issues and approaches.* Dans Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of, pages 38–46. IEEE, 1990. (Cité en page 12.)

[Kinder 2012] Stephen Joseph Kinder et James Irwin Knutson. *Method for accessing internal states of objects in object oriented programming*, mars 20 2012. US Patent 8,141,035. (Cité en page 83.)

[Laprie 1990] J-C Laprie, Jean Arlat, Christian Beounes et Karama Kanoun. *Definition and analysis of hardware-and software-fault-tolerant architectures.* Computer, vol. 23, no. 7, pages 39–51, 1990. (Cité en page 96.)

[Laprie 1995] JC Laprie, J Arlat, JP Blanquart, A Costes, Y Crouzet, Y Deswarte, JC Fabre, H Guillermain, M Kaâniche, K Kanoun*et al. Guide de la Sûreté de Fonctionnement, 324 p.* Cépaduès Editions, Toulouse, France, 1995. (Cité en pages 6 et 8.)

[Laprie 2008] Jean-Claude Laprie. *From dependability to resilience.* Dans 38th IEEE/IFIP Int. Conf. On Dependable Systems and Networks, pages G8–G9. Citeseer, 2008. (Cité en pages 10 et 11.)

[Li 2008] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V Adve, Vikram S Adve et Yuanyuan Zhou. *Understanding the propagation of hard errors to software and implications for resilient system design.* Dans ACM SIGARCH Computer Architecture News, volume 36, pages 265–276. ACM, 2008. (Cité en page 11.)

[Long 2009] Rongshen Long, Hong Li, Wei Peng, Yi Zhang et Minde Zhao. *An approach to optimize intra-ecu communication based on mapping of autosar runnable entities.* Dans 2009 International Conference on Embedded Software and Systems, pages 138–143. IEEE, 2009. (Cité en page 32.)

[Lu 2009] Caroline Lu, Jean-Charles Fabre et Marc-Olivier Killijian. *An approach for improving fault-tolerance in automotive modular embedded software.* 2009. (Cité en page 93.)

[Lyons 1962] Robert E Lyons et Wouter Vanderkulk. *The use of triple-modular redundancy to improve computer reliability.* IBM journal of research and development, vol. 6, no. 2, pages 200–209, 1962. (Cité en page 93.)

[Marder-Eppstein 2010] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian Gerkey et Kurt Konolige. *The office marathon : Robust navigation in an indoor office environment.* Dans 2010 IEEE international conference on robotics and automation, pages 300–307. IEEE, 2010. (Cité en page 25.)

[Marin 2001] Olivier Marin, Pierre Sens, Jean-Pierre Briot et Zahia Guessoum. *Towards adaptive fault tolerance for distributed multi-agent systems.* Proceedings of ERSADS, pages 195–201, 2001. (Cité en page 11.)

[Martorell 2014] Hélène Martorell. *Architectures et processus de développement permettant la mise à jour dynamique de systèmes embarqués automobiles.* PhD thesis, 2014. (Cité en page 35.)

[Martorell 2015] H. Martorell, J. C. Fabre, M. Lauer, M. Roy et R. Valentin. *Partial Updates of AUTOSAR Embedded Applications – To What Extent ?* Dans Dependable Computing Conference (EDCC), 2015 Eleventh European, pages 73–84, Sept 2015. (Cité en page 147.)

[Mera 2009] Edison Mera, Pedro Lopez-García et Manuel Hermenegildo. *Integrating software testing and run-time checking in an assertion verification framework*. Dans International Conference on Logic Programming, pages 281–295. Springer, 2009. (Cité en page 85.)

[NREC] NREC. *Description of the Crusher system*. `https://www.nrec.ri.cmu.edu/nrec/solutions/defense/other-projects/crusher.html`. Accessed : 2019-03-15. (Cité en page 25.)

[Oertel 2019] Markus Oertel et Bastian Zimmer. *More Performance with Autosar Adaptive*. ATZelectronics worldwide, vol. 14, no. 5, pages 36–39, May 2019. (Cité en page 150.)

[Parnas 1972] David Lorge Parnas. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, vol. 15, no. 12, pages 1053–1058, 1972. (Cité en page 18.)

[Plank 1994] James S Plank, Micah Beck, Gerry Kingsley et Kai Li. Libckpt : Transparent checkpointing under unix. Computer Science Department, 1994. (Cité en page 85.)

[Plank 1998] James S Plank, Kai Li et Michael A Puening. *Diskless checkpointing*. IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 10, pages 972–986, 1998. (Cité en page 85.)

[POS 2004] *IEEE Standard for Information Technology- Standardized Application Environment Profile (AEP)-POSIX Realtime and Embedded Application Support*. IEEE Std 1003.13-2003 (Revision of IEEE Std 1003.13-1998), pages i–164, 2004. (Cité en page 151.)

[POS 2018] *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7*. IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), pages 1–3951, Jan 2018. (Cité en page 151.)

[Powell 1991] D Powell. *A Generic Architecture for Dependable Distributed Computing*, 1991. (Cité en page 90.)

[Pyarali 2001] Irfan Pyarali, Marina Spivak, Ron Cytron et Douglas C Schmidt. *Evaluating and optimizing thread pool strategies for real-time CORBA*. ACM SIGPLAN Notices, vol. 36, no. 8, pages 214–222, 2001. (Cité en page 161.)

[Quigley 2009] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler et Andrew Y Ng. *ROS : an open-source Robot Operating System*. Dans ICRA workshop on open source software, volume 3, page 5, 2009. (Cité en pages 18 et 19.)

[Quigley 2015] Morgan Quigley, Brian Gerkey et William D Smart. Programming robots with ros : a practical introduction to the robot operating system. " O'Reilly Media, Inc.", 2015. (Cité en page 25.)

[Rajkumar 1995] Ragunathan Rajkumar, Michael Gagliardi et Lui Sha. *The real-time publisher/subscriber inter-process communication model for distributed real-time systems : design and implementation*. Dans Proceedings Real-Time Technology and Applications Symposium, pages 66–75. IEEE, 1995. (Cité en page 151.)

[ROS a] ROS. *ROS-industrial, supported hardware*. `http://wiki.ros.org/Industrial/supported_hardware`. Accessed : 2019-03-15. (Cité en page 26.)

[ROS b] ROS. *ROS on the ISS*. `http://www.ros.org/news/2014/09/ros-running-on-iss.html`. Accessed : 2019-03-15. (Cité en page 26.)

[ROS c] ROS. *ROS on the ISS*. `http://rosindustrial.org/the-challenge/`. Accessed : 2019-03-15. (Cité en page 26.)

[Rousselin 2017] Thomas Rousselin, Guillaume Hubert, Didier Régis, Marc Gatti et Alain Bensoussan. *Impact of aging on the soft error rate of 6T SRAM for planar and bulk technologies*. Microelectronics Reliability, vol. 76, pages 159–163, 2017. (Cité en page 49.)

[Rubinger 2010] Andrew Lee Rubinger et Bill Burke. Enterprise javabeans 3.1 : Developing enterprise java components. " O'Reilly Media, Inc.", 2010. (Cité en page 18.)

[Sandmann 2012] Guido Sandmann et Michael Seibt. *AUTOSAR-Compliant Development Workflows : From Architecture to Implementation-Tool Interoperability for Round-Trip Engineering and Verification and Validation*. Rapport technique, SAE Technical Paper, 2012. (Cité en page 152.)

[Seinturier 2009] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni et Jean-Bernard Stefani. *Reconfigurable sca applications with the frascati platform*. Dans 2009 IEEE International Conference on Services Computing, pages 268–275. IEEE, 2009. (Cité en page 16.)

[Sherlock 2000] Terence Sherlock, Gene Cronin et Terence P Sherlock. Com beyond microsoft : Designing and implementing com servers on compaq platforms. Digital Press, 2000. (Cité en page 18.)

[Sinha 2011] Purnendu Sinha. *Architectural design and reliability analysis of a fail-operational brake-by-wire system from ISO 26262 perspectives*. Reliability Engineering & System Safety, vol. 96, no. 10, pages 1349–1359, 2011. (Cité en page 1.)

[som] *Specification on SOME/IP Transport Protocol*. `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SOMEIPTransportProtocol.pdf`. Accessed : 2019-10-01. (Cité en page 151.)

[Stoicescu 2011] Miruna Stoicescu, Jean-Charles Fabre et Matthieu Roy. *Towards a System Architecture for Resilient Computing*. pages 17–19, 05 2011. (Cité en page 12.)

[Stoicescu 2012a] Miruna Stoicescu, Jean-Charles Fabre et Matthieu Roy. *From Design for Adaptation to Component-Based Resilient Computing*. Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC, pages 1–10, 11 2012. (Cité en page 9.)

[Stoicescu 2012b] Miruna Stoicescu, Jean-Charles Fabre et Matthieu Roy. *From design for adaptation to component-based resilient computing*. Dans Dependable Computing (PRDC), 2012 IEEE 18th Pacific Rim International Symposium on, pages 1–10. IEEE, 2012. (Cité en page 14.)

[Stoicescu 2013] Miruna Stoicescu. *Architecting Resilient Computing Systems : a Component-Based Approach*. PhD thesis, Institut National Polytechnique de Toulouse, 2013. 2013INPT0120. (Cité en pages 13, 14, 16 et 43.)

[Szyperski 1999] Clemens Szyperski, Jan Bosch et Wolfgang Weck. *Component-oriented programming*. Dans Object-oriented technology ecoop'99 workshop reader, pages 184–192. Springer, 1999. (Cité en page 18.)

[Szyperski 2002] Clemens Szyperski. Component software : Beyond object-oriented programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd édition, 2002. (Cité en page 18.)

[Times] Times. *Toyota says recalls will cost the company about $2 billion*. `https://www.latimes.com/archives/la-xpm-2010-feb-04-la-fi-toyota-profit5-2010feb05-story.html`. Accessed : 2019-09-30. (Cité en page 149.)

[Welsh 1999] Matt Welsh, Mathhias Kalle Dalheimer et Lar Kaufman. Running linux. O'Reilly & Associates, Inc., 1999. (Cité en page 19.)

[Xu 1996] Jie Xu et Brian Randell. *Roll-forward error recovery in embedded real-time systems*. Dans Proceedings of 1996 International Conference on Parallel and distributed systems, pages 414–421. IEEE, 1996. (Cité en page 90.)

[Zahir 1998] Andree Zahir et Patrick Palmieri. *OSEK/VDX-operating systems for automotive applications*. 1998. (Cité en page 34.)

Publications

Michaël Lauer, Matthieu Amy, William Excoffon, Matthieu Roy, Miruna Stoicescu. Towards Adaptive Fault Tolerance : From a Component-Based Approach to ROS. Dans CARS 2015 - Critical Automotive applications : Robustness & Safety, Sep 2015, Paris, France.

Jean-Charles Fabre, Michaël Lauer, Matthieu Roy, Matthieu Amy, William Excoffon et al. Towards Resilient Computing on ROS for Embedded Applications. Dans 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Jan 2016, TOULOUSE, France

Michaël Lauer, Matthieu Amy, Jean-Charles Fabre, Matthieu Roy, William Excoffon et al. Engineering Adaptive Fault-Tolerance Mechanisms for Resilient Computing on ROS. Dans HASE 2016 — IEEE 17th International Symposium on High Assurance Systems Engineering Symposium, Jan 2016, Orlando, FL, United States. pp.94-101.

Matthieu Amy. Adaptive Fault Tolerance : Is ROS a Relevant Executive Support ? Dans Student Forum of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Jun 2016, Toulouse, France.

Matthieu Amy, Jean-Charles Fabre, Michaël Lauer. Towards Adaptive Fault Tolerance on ROS for Advanced Driver Assistance Systems. Dans 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), Jun 2017, Denver, United States. 7p.

Michaël Lauer, Matthieu Amy, Jean-Charles Fabre, Matthieu Roy, William Excoffon et al. Resilient Computing on ROS using Adaptive Fault Tolerance. Dans Journal of Software : Evolution and Process, John Wiley & Sons, Ltd., 2018, 30 (3), pp.e1917

Résumé : A l'instar du téléphone mobile évoluant en smartphone, la voiture s'est transformée petit à petit en smartcar. Les aides à la conduite, l'infotainment ou encore la personnalisation du véhicule sont les points clefs de l'attractivité auprès des consommateurs. L'apparition des véhicules automobiles connectés a permis aux constructeurs de mettre à jour à distance les logiciels embarqués, favorisant leur maintenabilité et l'ajout a posteriori de fonctionnalités. Dans ce contexte, le consortium AUTOSAR, un regroupement de constructeurs automobiles majeurs, a conçu une nouvelle plate-forme logicielle facilitant la mise à jour à distance et la modification en ligne de ces systèmes embarqués. Cependant, avec de plus en plus de complexité dans ces logiciels, il est devenu essentiel de pouvoir assurer un service sûr de fonctionnement malgré des changements imprévus. Ainsi, les mécanismes de sûreté de fonctionnement doivent eux aussi s'adapter et être mis à jour pour assurer la résilience du système, à savoir, la persistance de la sûreté de fonctionnement face à des changements. Les mécanismes de tolérance aux fautes (Fault Tolerance Mechanisms - FTM) assurant un service nominal ou dégradé en présence de fautes doivent également s'adapter face à un changement de contexte applicatif (changement du modèle de faute, des caractéristiques de l'application ou des ressources disponibles). Cette capacité à adapter les FTM est appelée Tolérance aux Fautes Adaptative (Adaptive Fault Tolerance – AFT). C'est dans ce contexte d'évolution et d'adaptativité que s'inscrivent nos travaux de thèse. Dans cette thèse, nous présentons des approches pour développer des systèmes sûrs de fonctionnement dont les FTM peuvent s'adapter à l'exécution par des modifications plus ou moins à grain fin pour minimiser l'impact sur l'exécution de l'application.

Nous proposons une première solution basée sur une approche par composants substituables, nous décomposons nos FTM selon un schéma de conception Before-Proceed-After regroupant respectivement les actions de sûreté de fonctionnement s'exécutant avant une action l'application, la communication avec l'application et celles s'exécutant après une action de l'application. Nous implémentons cette approche sur ROS (Robot Operating System), un intergiciel pour la robotique permettant de créer des applications sous forme de graphe de composants.

Nous proposons ensuite une seconde solution dans laquelle nous affinons la granularité des composants de nos FTM et nous catégorisons, dans un premier temps, les actions de sûreté de fonctionnement qu'ils contiennent. Cela permet non plus de substituer un composant mais une action élémentaire. Ainsi, nous pallions à un problème de ressource apparu dans l'approche par composants substituables. Un composant étant projeté sur un processus, nos FTM utilisent inutilement des ressources déjà limitées sur les plate-formes embarqués. Pour ce faire, nous proposons une solution basé sur une approche par objets ordonnançables. Les FTM passent d'une conception par graphe de composants à une conception par graphe d'objets. Les actions de sûreté de fonctionnement sont projetés sur des objets qui sont ordonnancés à l'intérieur du FTM. Cette seconde solution est aussi mise en œuvre sur ROS.

Enfin, nous faisons une analyse critique des deux supports d'exécution logiciel

pour l'automobile, à savoir, AUTOSAR Classic Plateform, et AUTOSAR Adaptive Platform, qui est en cours de développement encore actuellement. Nous examinons, dans une dernière étape la compatibilité entre ces deux supports et nos approches pour concevoir des systèmes résilients embarqués basés sur de la tolérance aux fautes adaptative.

Mots clés : Système embarqués, Sûreté de fonctionnement, Programmation adaptative, Systèmes automobiles, Programmation orientée composants, Programmation orientée objets