Résilience des systèmes informatiques adaptatifs : modélisation, analyse et quantification

William Excoffon

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Sureté de Logiciel et Calcul à Haute Performance

Présentée et soutenue par :

M. WILLIAM EXCOFFON

le vendredi 8 juin 2018

Titre :

Résilience des systèmes informatiques adaptatifs: modélisation, analyse et quantification

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Laboratoire d'Analyse et d'Architecture des Systèmes (L.A.A.S.)

Directeur(s) de Thèse :

M. JEAN-CHARLES FABRE

M. MICHAEL LAUER

Rapporteurs :

M. FRANCOIS TAIANI, UNIVERSITE RENNES 1

Mme SARA BOUCHENAK, INSA LYON

Membre(s) du jury :

M. LIONEL SEINTURIER, UNIVERSITE LILLE 1, Président

M. JEAN-CHARLES FABRE, INP TOULOUSE, Membre

M. JEAN-PAUL BLANQUART, AIRBUS FRANCE, Membre

M. MARC GATTI, THALES AVIONICS, Membre

M. MICHAEL LAUER, UNIVERSITE TOULOUSE 3, Membre

Remerciements

Une thèse n'est jamais un long fleuve tranquille, afin d'arriver à bon port il faut passer de nombreux obstacles. Cette aventure dans laquelle je me suis embarqué n'aurait sans doute pas était rendu possible sans les nombreuses personnes qui m'ont épaulé durant ces années.

Je tiens tout particulièrement à remercier mes directeurs de thèse Jean-Charles Fabre et Michaël Lauer, pour leur soutient autant scientifique que moral. Vous m'avez offert cette chance et avez crû en moi plus que je ne l'ai moi-même fait et je vous en suis reconnaissant.

Je remercie également Madame Sarah Bouchenak et Monsieur François Taiani de m'avoir fait l'honneur d'être les rapporteurs de cette thèse.

J'adresse également mes remerciements à Messieurs les membres du jury Lionel Seinturier, Jean-Paul Blanquart et Marc Gatti de m'avoir accordé de leur temps. Les discussions que nous avons pu avoir m'ont permis d'envisager mon travail sous un autre angle.

Cette thèse a été conduite au sein de l'équipe TSF, c'est pourquoi je remercie les permanents pour leurs conseils et leur bienveillance. Merci également aux doctorants passés, présents et futurs qui m'ont permis de garder le sourire lors des moments difficiles.

Enfin je remercie ma famille pour leur soutien indéfectible, ils sont pour moi un modèle et m'ont permis de garder courage et espoir. Mes derniers remerciements vont à ma compagne qui a toujours été là pour me supporter et me soutenir dans tout ce que j'ai entrepris.

Résumé

Titre: Résilience des systèmes informatiques adaptatifs : Modélisation, analyse et quantification

On appelle résilient un système capable de conserver ses propriétés de sûreté de fonctionnement en dépit des changements (nouvelles menaces, mise-à-jour,…). Les évolutions rapides des systèmes, y compris des systèmes embarqués, implique des modifications des applications et des configurations des systèmes, en particulier au niveau logiciel. De tels changements peuvent avoir un impact sur la sûreté de fonctionnement et plus précisément sur les hypothèses des mécanismes de tolérance aux fautes. Un système est donc résilient si de pareils changements n'invalident pas les mécanismes de sûreté de fonctionnement, c'est-à-dire, si les mécanismes déjà en place restent cohérents malgré les changements ou dont les incohérences peuvent être rapidement résolues.

Nous proposons tout d'abord dans cette thèse un modèle pour les systèmes résilients. Grace à ce modèle nous pourrons évaluer les capacités d'un ensemble de mécanismes de tolérance aux fautes à assurer les propriétés de sûreté issues des spécifications non fonctionnelles. Cette modélisation nous permettra également de définir un ensemble de mesures afin de quantifier la résilience d'un système. Enfin nous discuterons dans le dernier chapitre de la possibilité d'inclure la résilience comme un des objectifs du processus de développement.

Title: Resilience of Adaptive Computer Systems: Modelling, Analysis and Quantification

A system that remains dependable when facing changes (new threats, updates) is called resilient. The fast evolution of systems, including embedded systems, implies modifications of applications and system configuration, in particular at software level. Such changes may have an impact on the dependability of the system, in particular on the assumptions of the fault tolerance mechanisms. A system is resilient when such changes do not invalidate its dependability mechanisms, said in a different way, current dependability mechanisms remain appropriate despite changes or whose inconsistencies can be rapidly solved.

We propose in this thesis a model for resilient computing systems. Using this model we propose a way to evaluate if a set of fault tolerance mechanisms is able to ensure dependability properties from non-functional specifications. The proposed model is the used to quantify the resilience of a system using a set of specific measures. In the last chapter we discuss the possibility of including resilience as a goal in development process.

Mots-clés: Tolérance aux fautes, Résilience, Métriques d'évaluation

Discipline: Systèmes informatiques critiques

Intitulé et adresse du laboratoire: CNRS ; LAAS ; 7 avenue du colonel Roche, 31077 Toulouse, France

Table des matières

Liste des Figures

Liste des Tableaux

Introduction

« Intelligence is the ability to adapt to change. »
Stephen Hawking (1942-2018) – A briefer history of time

Nous vivons dans un monde où les systèmes informatiques sont devenus indissociables de notre vie quotidienne. Adaptés à nos modes de vie, ces systèmes se doivent être à la fois sûrs de fonctionnement et hautement évolutif. Nous sommes en effet très sensibles aux capacités d'évolution et d'adaptation de nos « compagnons électroniques » tels que les téléphones mobiles. Nous sommes aussi très sensibles à leur disponibilité, nous en sommes peut-être même victimes tant il est difficile de s'en séparer, tout au moins de se séparer de leurs services. Nous sommes tous les jours confrontées à la mise à jour à distance de leurs fonctionnalités, un mal nécessaire pour pouvoir bénéficier des services qu'ils offrent. C'est une forme de maintenance à distance pour le fournisseur de services, mais plus encore, une façon de conserver les fonctionnalités au niveau de leur dernière version ce qui donne l'impression à l'utilisateur que son appareil est dernier cri ! Bien entendu, cela repose sur un matériel qui n'évolue pas et c'est donc la dimension logicielle qui est le cœur du problème.

Les systèmes embarqués dont nous venons de parler n'ont pas un niveau de criticité élevé, mais l'impact commercial d'une non stabilité suite à des évolutions fréquentes est important. Ce n'est pas l'impact économique qui nous intéresse dans cette thèse, mais l'impact de ces évolutions sur les propriétés de sûreté des systèmes embarqués critiques. En effet, on parle aujourd'hui de systèmes embarqués critiques tels que les véhicules autonomes qui nécessitent des capacités d'adaptations inenvisageables il y a quelques années. Ces systèmes sont donc soumis à de nombreux changements, qu'ils soient environnementaux, matériels, ou bien fonctionnels. C'est bien cette évolutivité qui est un défi pour la sûreté de fonctionnement et qu'il faut réussir à surmonter. Ce qui change la donne aujourd'hui c'est que les systèmes embarqués critiques qui sont habituellement des systèmes fermés deviennent de plus en plus ouverts. On parle d'*over-the-air updates* dans l'automobile, les véhicules du futurs auront donc la capacité de se mettre à jour sans passer par le garagiste. Dans l'avionique la tendance est au *single pilot operation*, ce qui signifie que le copilote sera tôt ou tard remplacé par un système informatique. Toutes ces technologies reposent souvent sur des accès à des services distants, le plus souvent hébergés dans le *Cloud*.

D'une manière générale, et quelle que soit l'origine de ces changements il n'y a que deux issues possibles : le système peut ou ne peut pas s'adapter pour absorber ces évolutions. L'objectif est donc de garantir un service sûr à tout moment de l'utilisation de ces systèmes. D'un point de vue technique même si une partie des évolutions qui affectent le système peuvent être prévues il existera toujours une partie dont on ne connait ni la nature ni la date à laquelle elles vont survenir. Et c'est en cela que réside le défi de la résilience informatique: comment peut-on garantir qu'un système reste sûr de fonctionnement et ce quelques soient les changements qu'il subit ?

Assurer la sûreté de fonctionnement d'un système est en soi une contrainte forte à un coût non-négligeable qui peut devenir prohibitif pour certaines applications industrielles. Le coût de l'utilisation de techniques de sûreté est acceptable à condition qu'elles soient efficaces lorsqu'un problème survient. Supporter ce coût, y compris à l'exécution, sans pouvoir se prémunir des défaillances lorsque elles surviennent est bien entendu inacceptable. Très clairement, assurer la sûreté nécessite l'utilisation de techniques coûteuses à la fois en termes de développement et en termes de ressources nécessaires à leur implémentation. Ces solutions représentent une deuxième couche dite non-fonctionnelle qui se superpose à la couche fonctionnelle. Elles ont pour but de garantir des propriétés telles que la fiabilité, la disponibilité, la sûreté (*safety*), la sécurité (*security*),… Nous nous intéresserons dans cette thèse aux solutions de sûreté de fonctionnement et plus spécifiquement aux mécanismes de tolérance aux fautes qui sont un moyen d'assurer en ligne un service nominal ou dégradé conforme aux spécifications du système, en dépit de l'occurrence de fautes. Il est à noter que lorsque l'on souhaite rendre un système sûr de fonctionnement, non seulement il faut ajouter une couche de protection mais il faut également modifier le processus de développement de ces systèmes. On parle ici de normes (par exemple l'ISO26262 dans l'automobile, le DO178C dans l'avionique) dans lesquelles il est imposé un ensemble de procédés tels que les Analyse des Modes de Défaillances, de leurs Effets et de leur Criticité (AMDEC), du test, des analyse de pires temps d'exécutions, etc…

Jusqu'à présent les notions de modularité et de flexibilité étaient l'apanage de la couche fonctionnelle, c'est-à-dire celle qui correspond aux fonctionnalités proposées à l'utilisateur. Des technologies de plus en plus répandues permettent d'avoir des systèmes très évolutifs, à la fois en tant que plateforme comme par exemple les architectures basées sur des composants, mais aussi du point de vue fonctionnel avec l'introduction de mises-à-jour *Over-The-Air*, c'est-à-dire automatique et à distance, qui viennent ajouter ou modifier des fonctionnalités déjà en place. Toutes ces modifications introduisent la nécessité d'adapter les parties non-fonctionnelles des systèmes pour garantir les propriétés de sûreté de fonctionnement et ce quelques soient la nature des modifications subies.

Les travaux présentés dans ce manuscrit proposent une approche pour modéliser et quantifier la capacité d'un système à garantir sa sûreté de fonctionnement et ce malgré les changements qu'il subira lors de sa vie opérationnelle et quelle que soit leur origine. Une des approches pour garantir ces propriétés non fonctionnelles au cours du temps est d'être capable d'adapter les mécanismes de tolérance aux fautes et ce d'une façon la plus légère possible. C'est ce que l'on appelle la tolérance aux fautes adaptative ou *Adaptive Fault Tolerance*. Or cette approche ne peut être réalisée que si l'on arrive à répondre aux deux questions suivantes :

-Quels sont les paramètres affectés par les changements qui ont un impact sur la configuration du système (c'est-à-dire la combinaison entre les applications et leurs mécanismes de protections) ?

-Comment rendre le système le plus résilient possible, c'est-à-dire moins sensible aux effets des changements et à un coût acceptable pour l'application dans son contexte industriel?

Après une introduction plus détaillée de la problématique et du positionnement de cette dernière dans le chapitre 1 nous proposons une approche en plusieurs étapes afin de répondre à ces questions. Le travail de cette thèse repose sur notre capacité à modéliser les hypothèses de fonctionnement liant les applications (couche fonctionnelle) et les mécanismes de tolérance aux fautes (couche non-fonctionnelle).

C'est cette modélisation que nous présentons dans le chapitre 2. Ce chapitre permettra de définir également les principes clés de la résilience d'un système à travers les propriétés de compatibilité, d'adéquation et de cohérence entre applications et mécanismes.

Le chapitre 3 propose une approche analytique de la résilience. Nous mettrons en avant dans ce chapitre les contraintes de sélection des mécanismes de tolérance aux fautes en proposant un premier estimateur de la résilience d'un système que l'on appellera *Ratio de Cohérence*. Ce ratio permettra de quantifier la capacité d'un ensemble de mécanismes à tolérer les changements d'une application. Grace à des analyses de sensibilité nous mettrons en avant la possibilité d'améliorer la résilience des systèmes en minimisant les efforts de développement nécessaires.

Tout comme pour la sûreté de fonctionnement, il est nécessaire lorsque l'on parle de résilience de travailler avec un ensemble de mesures qui permettent de quantifier les effets des changements sur le système. Dans le chapitre 4, nous pousserons l'analogie afin d'obtenir un ensemble de mesures classiques de la sûreté de fonctionnement qui se veulent être une extension des mesures telles quel le temps moyen entre défaillances (Mean Time Between Failure), le temps moyen de réparation (Mean Time To Repair), etc.

Enfin le dernier chapitre propose une méthode d'inclusion des mesures précédentes dans un processus de développement. Comme dit précédemment, la sûreté de fonctionnement est une contrainte qui impose des efforts à chaque étape du processus de développement et des coûts. Nous proposons donc des outils tels que des analyses de changements et leurs effets sur la criticité afin d'étendre la sûreté de fonctionnement pour garantir également la résilience d'un système dès les débuts de sa conception.

Nous énoncerons enfin quelques perspectives avant de conclure.

Chapitre 1 – Contexte et positionnement du problème

“For me context is the key - from that comes the understanding of everything. “
Kenneth Noland (1924–2010)

1-1 Introduction

Dans ce chapitre nous présentons le contexte de cette thèse ainsi que les principes fondamentaux sur lesquels reposent ces travaux de recherche. Dans un contexte de systèmes de plus en plus évolutifs, nous cherchons à mieux comprendre et analyser les risques que posent ces évolutions du point de vue de la sûreté de fonctionnement.

Nous commencerons donc par situer le problème technologique sous-jacent, puis nous utiliserons un court exemple afin de mettre en lumière les verrous technologiques et scientifiques qu'il nous faudra lever. Enfin nous présenterons un état de l'art afin de positionner ce travail de recherche par rapport aux travaux déjà existants.

1-2 De la sûreté de fonctionnement à la résilience

1.2.1 Sûreté de fonctionnement

Aujourd'hui, les systèmes devant être sûrs de fonctionnement sont partout : automobiles, avions, serveurs, … On parle de sûreté de fonctionnement lorsque l'on s'intéresse à des systèmes critiques dont la défaillance est lourde de conséquences sur le plan humain et/ou économique.

Cette sûreté de fonctionnement englobe de nombreux concepts dont la taxonomie et les définitions sont clairement établies depuis plusieurs décennies [23]. Ainsi le groupe IFIP 10.4 *Working Group on Dependable Computing and Fault Tolerance* défini la sûreté de fonctionnement (*dependability*) d'un système comme son aptitude à fournir un service avec un niveau de confiance justifiée. Autrement dit, un système est dit sûr de fonctionnement s'il est démontrable qu'il évite des défaillances trop fréquentes ou trop graves. Cette définition englobe plusieurs concepts :

-La disponibilité
-La fiabilité
-l'intégrité
-la maintenabilité
-la sûreté de fonctionnement (*safety*)

Chacune de ces propriétés peut être mesurée d'après des indicateurs précis (MTTF, MTBF, Mean Time To Repair,…). La disponibilité mesure la probabilité d'un système à fournir un service à un instant t. Elle se calcule comme le rapport entre le temps moyen de bon fonctionnement jusqu'à la première panne (MTTF et le temps moyen entre panne (MTBF). La fiabilité quant à elle mesure la capacité d'avoir un service continu dans un intervalle de temps donné. L'intégrité mesure l'absence d'altération du service rendu par le système et la

maintenabilité qui quantifie notre capacité à modifier et/ou réparer un système durant sa vie opérationnelle en utilisant les temps moyens d'intervention (MTTR).

Il est à noter que le terme de sûreté de fonctionnement est source de confusion en français. En effet il est à la fois la traduction de *dependability* et de *safety*. Or la *dependability* englobe la *safety,* puisqu'un système *safe* se caractérise par l'absence de défaillance catastrophique alors qu'un système *dependable* sera non seulement *safe* mais devra également être disponible, fiable, intègre et maintenable.

Les menaces pour un système critiques sont de trois sortes : les défaillances, les erreurs et les fautes. Ce sont les défaillances que l'on veut empêcher d'arriver. En effet on considère qu'il y a défaillance lorsque le service fournit par le système n'est plus celui que l'on attend. La défaillance c'est donc un évènement à partir duquel le service observé n'est plus conforme aux spécifications formulées au court de sa conception.

Une défaillance est donc une conséquence de la déviation du comportement du système qui peut être composé de plusieurs sous-systèmes. Cette déviation est appelée une erreur. Elle apparait en court de fonctionnement lorsqu'un ou plusieurs sous-systèmes se retrouvent dans un état inattendu. Une erreur est la conséquence de l'activation d'une faute.

Une faute est un défaut du système, ce défaut peut être natif comme une section de code instable ou il peut apparaitre avec le temps comme dans le cas de l'usure de la plateforme matérielle. Ainsi un système peut contenir des fautes mais c'est l'activation de celle-ci qui génère des erreurs qui peuvent conduire le système tout entier à défaillir via un mécanisme de propagation.

Ainsi dans un système composite on peut représenter cette chaine de propagation par le schéma suivant :

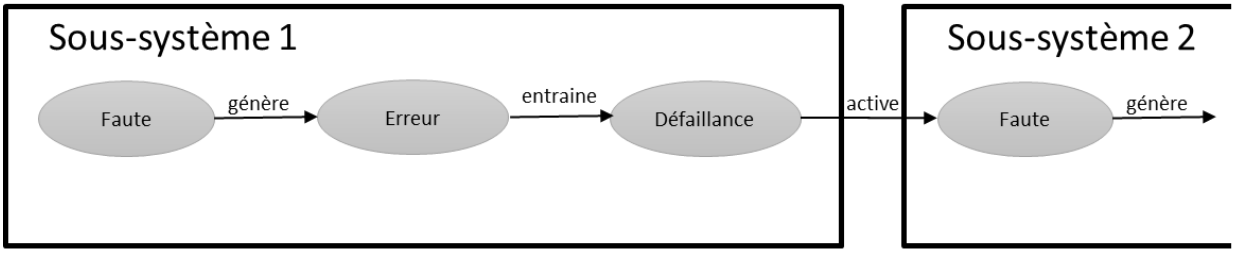


Figure 1- Chaine de défaillance

L'objectif d'un concepteur de système est donc de limiter les défaillances. Or compte tenu du lien de causalité entre fautes et défaillances, s'attaquer aux défaillances d'un système c'est avant tout limiter les fautes. Ainsi on dispose de 4 moyens comme défini dans [1]:

-La *prévention des fautes*, c'est-à-dire l'ensemble des actions qui visent à prévenir l'occurrence ou l'introduction de fautes (ex : processus de développement certifiés).

-La *suppression de fautes*, une fois le système conçu on s'attache à éliminer les fautes par exemple en utilisant des méthodes de test.

-La *prévision de fautes*, pour laquelle on cherche à estimer le nombre de fautes ainsi que leurs probables conséquences sur le système (ex : analyse des modes de défaillances AMDEC).

-La *tolérance aux fautes*, c'est-à-dire la capacité à ce qu'une faute résiduelle ne produise pas de défaillance (ex : mécanisme de redondance).

Toutes ces définitions et ces mesures pour la sûreté de fonctionnement d'un système reposent sur un ensemble d'hypothèses faîtes lors de leurs évaluations. Mais que se passe-t-il lorsque le système évolue ? Que reste-t-il de la sûreté de fonctionnement ?

1.2.2 Résilience

La résilience est avant tout un concept qui, selon le domaine dans lequel il est utilisé, peut revêtir plusieurs significations. En écologie, la résilience est la capacité d'un écosystème à se remettre de perturbations. En science sociale un individu sera résilient s'il est capable de s'adapter pour surmonter l'adversité… Cependant on note un point commun à toutes les définitions : un système résilient est capable d'absorber un évènement limitant ainsi son impact et quand bien même il serait impacté, le système est capable de retourner à un état stable.

Comme l'a défini J.C. Laprie dans [2], la résilience c'est la persistance de la sûreté de fonctionnement (*dependability*) lors des changements (environnementaux, fonctionnel, …). Ainsi un système résilient est avant tout un système sûr de fonctionnement dont les propriétés de sûreté sont toujours vraies malgré les changements. Et tout comme la sûreté de fonctionnement, la résilience englobe différents aspects et différentes problématiques.

Tout d'abord il nous faut classifier les différents types de changements selon leur nature, leur prédictibilité et leur temporalité. Ces changements peuvent concerner autant le système en lui-même que les menaces qu'il peut être amené à rencontrer au cours de sa vie opérationnelle. La classification suivante proposée par J.C Laprie dans [2] permet d'avoir une vue d'ensemble des problématiques que soulève la résilience :

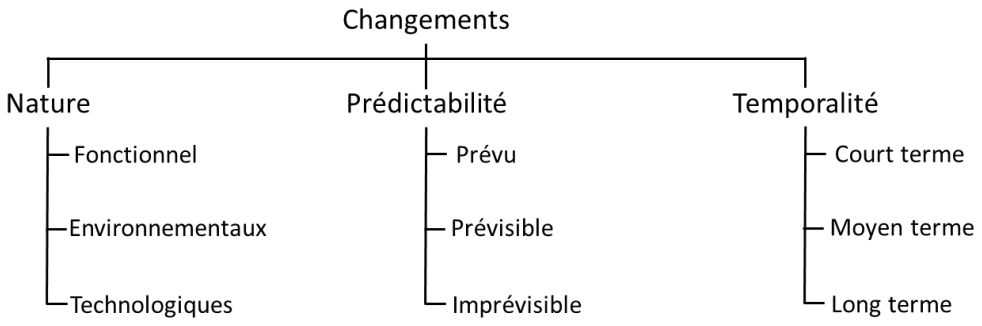


Figure 2 - Classification des changements

C'est la nature des changements qui va déterminer leur impact sur les différentes composantes du système et donc les conséquences en termes de risque de défaillances. Les aspects temporels et prédictifs des changements influent directement sur notre capacité à les absorber. En effet un changement prévu à long terme autorise un temps de réflexion et de prise de décision généralement suffisant pour que des mesures soient prises (maintenance prédictive par exemple). En revanche, un changement imprévisible et quasi immédiat sera plus difficile à absorber pour le système.

Pour illustrer ces changements on peut citer les travaux de O.Martin, M.Bertier et P.Sens [24] qui proposent d'observer les défaillances des plateformes matérielles afin d'ajuster les stratégies de redondances en fonction de ces défaillances. Ainsi, si le matériel devient de plus en plus défaillant on augmente le nombre de répliques de l'application à protéger pour s'adapter.

Dans [25] est présentée une méthode d'analyse des défaillances des plateformes matérielles. Ces travaux considère les changements de fréquences de défaillance afin d'utiliser des mécanismes duplex de redondance froide, tiède ou chaude. La réactivité des mécanismes utilisés est donc en corrélation directe avec la fréquence de défaillance susceptible de changer.

1.2.3 Devenir résilient

Tout comme pour la sûreté de fonctionnement, il existe des concepts généraux afin de rendre un système résilient. Ces concepts sont au nombre de quatre : l'évolutivité, l'évaluabilité, l'utilisabilité et la diversité. On retrouve une description plus précise de ce que chacun d'entre eux signifie dans [2]. Dans notre cas c'est la notion d'adaptabilité sous-jacente à l'évolvabilité qui nous intéressera. Un système adaptatif est un système capable d'évoluer lors de son exécution afin d'absorber les changements.

Ces techniques sont à mettre en parallèle avec les moyens de la sûreté de fonctionnement tels que la prévision, la prévention, la tolérance et la suppression de faute. Toutefois, les moyens de la sûreté de fonctionnement sont généralement définis hors-ligne et uniquement lors de la phase de conception. Afin de déterminer si un système est sûr de fonctionnement on utilisera des mesures comme le taux de défaillance, le temps moyen entre pannes,… Toutes ces mesures sont calculées sur des hypothèses supposées constantes (logiciel identique tout au long de la vie du système, taux de défaillance de la plateforme matérielle constant, etc…)

Or un système résilient se doit d'être adaptable. Cette adaptabilité des mécanismes de tolérance aux fautes est ce qui permet à un système d'absorber un changement, ou bien de se préparer au mieux au suivant. K. H. Kim et Thomas F. Lawrence [3] définissent ainsi *l'Adaptive Fault Tolerance* (Tolérance aux Fautes Adaptative) : « c'est une approche qui cherche à remplir les exigences dynamiques et volatiles de la sûreté de fonctionnement et utilisant de manière adaptive et efficaces des ressources d'exécutions changeantes et

limitées. On considère aujourd'hui que la tolérance aux fautes adaptative est la clé qui permettra de rendre un jour les systèmes résilient.

Ce que nous présentons dans ces travaux de thèse n'est non pas une méthode d'adaptation mais un ensemble d'outils théoriques permettant de mesurer l'impact de l'utilisation de la tolérance aux fautes adaptative. Comme nous le verrons dans les chapitres suivants, nous souhaitons proposer des outils de modélisation ainsi que des mesures de la résilience d'un système. Pour mieux appréhender ces notions d'adaptabilité et de résilience la section suivante introduira un court exemple illustratif.

1.3 Exemple de scénario d'adaptation

Supposons que nous étudions un système embarqué automobile. De nos jours les véhicules contiennent une grande capacité d'exécution logicielle répartie sur plusieurs calculateurs. Puisque le logiciel a remplacé de nombreuses fonctions qui autrefois étaient principalement réalisées par des composants matériels, il est normal de voir apparaitre de plus en plus la nécessité de mettre à jour ce logiciel au cours de la vie du système.

Pour illustrer le lien entre une application et son mécanisme de tolérance aux fautes associé ainsi que l'impact des mises à jour de l'application on considère à l'instant t_0 une application *App* dont la version est v_0. Cette application possède un ensemble de caractéristiques telles que le déterminisme (i.e. pour des entrées identiques le résultat sera toujours le même) et l'accès à son état (état de la mémoire, données sauvegardées,…). Du point de vue du modèle de fautes on considère que cette application risque de subir des fautes par crash, c'est-à-dire une interruption de durée indéfinie. Les exigences de sûreté de fonctionnement nécessitent que cette application soit protégée contre ce type de fautes.

Dans un premier temps on décide d'associer à cette application un mécanisme de tolérance aux fautes de type *Primary Backup Replication* (PBR). Ce mécanisme duplex que l'on appelle également redondance froide ou passive, permet d'envoyer à intervalles réguliers des points de sauvegardes de l'application primaire à une copie secondaire hébergée sur une plateforme matérielle différente de la première. En cas de défaillance de la copie primaire, on redémarre à froid une nouvelle copie en partant du dernier point de sauvegarde enregistré par la copie secondaire.

Pour mettre en place un tel mécanisme il est nécessaire d'avoir accès à l'état de l'application afin de mettre en place des points de sauvegardes. En revanche, l'application ne doit pas nécessairement être déterministe puisque ce mécanisme n'exige pas que la copie secondaire exécute également la requête, les deux copies ne sont donc pas nécessairement identiques. Ce mécanisme permet de tolérer les fautes pas crash et donc de satisfaire les exigences de sûreté de fonctionnement.

Supposons à présent qu'une mise à jour arrive à l'instant t_1 (cf figure 3). La nouvelle version de l'application v_1 est *non déterministe* mais conserve la possibilité d'accéder à son

état. Dans ce cas, application et mécanisme PBR sont encore compatibles. Il n'y a donc pas d'impact vis-à-vis de la sûreté de fonctionnement puisque l'exigence de tolérance aux fautes par crash est toujours satisfaite : le système est résilient à ce changement.

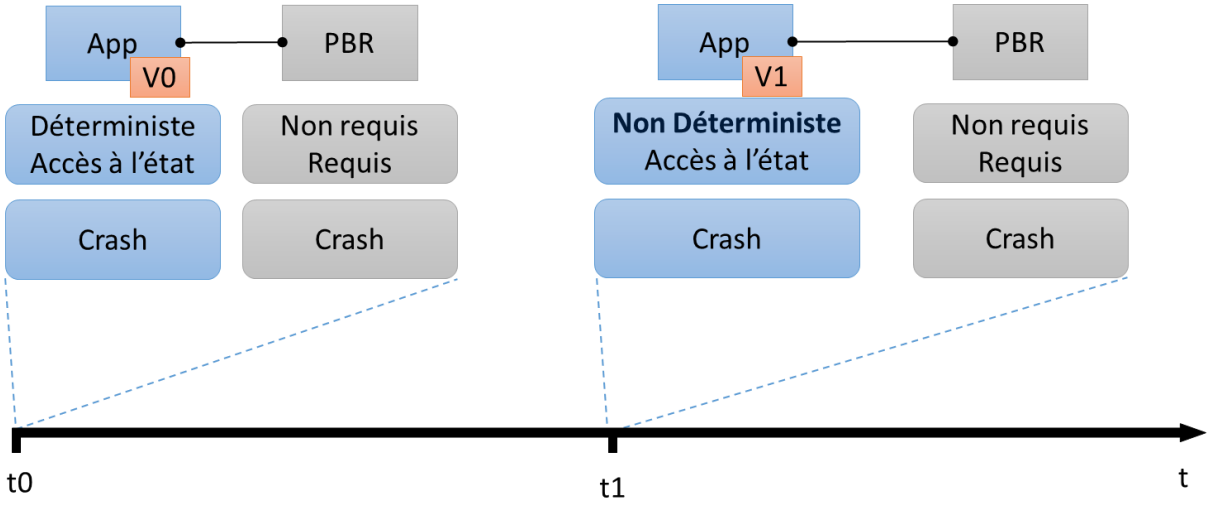


Figure 3 - Cas d'un changement sans impact

Supposons à présent qu'à l'instant suivant t_2, une nouvelle mise à jour arrive. Cette fois-ci l'application redevient déterministe mais on perd l'accès à son état. Ce changement a pour conséquence de rendre incompatible le mécanisme PBR avec l'application. Cette dernière n'est alors plus protégée contre les fautes qu'elle devrait être capable de tolérer. Dans ce cas le système n'est pas résilient.

La solution proposée par la tolérance aux fautes adaptative est de changer le mécanisme devenu incompatible en un temps minimal que nous appellerons dt_2 voir figure 4. Il s'agira dans notre cas de remplacer le mécanisme PBR par un mécanisme *Leader Follower Replication* (LFR) de type redondance chaude (ou semi-acitve). Ce mécanisme fonctionne en exécutant parallèlement deux copies parfaitement identiques de l'application sur deux plateformes matérielles. La première est appelée *Leader* la seconde *Follower*. Les deux copies exécutent les même requêtes, cette stratégie duplex nécessite donc une exécution déterministe de l'application afin de garantir qu'en cas de reprise du *Follower* après un crash du *Leader* les sorties soient identiques. LFR a pour avantage de ne pas nécessiter de point de sauvegarde et donc de ne pas nécessiter l'accès à l'état de l'application ce qui le rend compatible avec la version v_2 de l'application.

Dans cet exemple il existe donc une période dite de *fragilité* lorsque l'exigence de tolérance aux fautes par crash n'est plus respectée. On note également que dans ce scénario un évènement sur deux a introduit une incohérence entre l'application et son mécanisme de tolérance aux fautes. Toutes ces informations sont autant d'indicateurs de la résilience du système que nous définirons précisément et étendrons dans cette thèse.

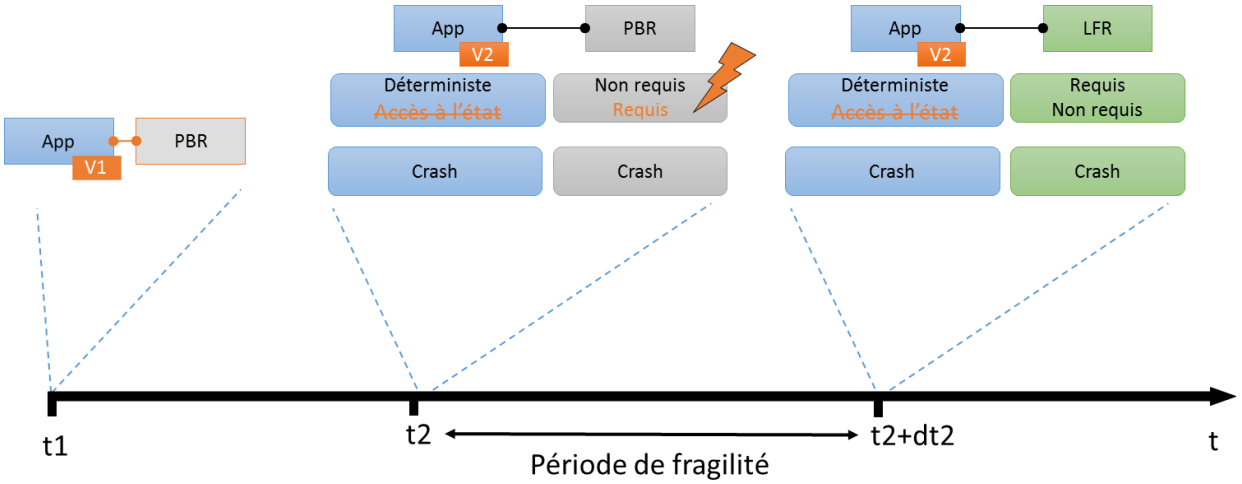


Figure 4 - Cas d'une application nécessitant l'adaptation de son mécanisme de tolérance aux fautes

Cet exemple montre succinctement les conséquences sur la sûreté de fonctionnement de l'évolution des systèmes. Bien entendu, les sources de changements peuvent être variées (environnement, usure…) et affecter les ressources et le modèle de fautes en plus des caractéristiques applicatives. Dans un monde où les systèmes critiques sont de plus en plus présents et interconnectés non seulement entre eux mais avec nos vies, la résilience est aujourd'hui un défi majeur qu'il nous faudra relever.

1.4 Positionnement du problème

Ces travaux de recherches s'incluent dans un contexte où les systèmes embarqués sont de plus en plus autonomes et où la généricité des plateformes matérielles implique une multiplication et une complexification du logiciel.

Ces systèmes d'aujourd'hui et de demain sont soumis à des modifications fréquentes et rapides, que ce soit de par l'environnement dans lequel ils évoluent où les modulations des fonctionnalités qu'ils offrent.

Prenons le cas de l'automobile par exemple. Les perspectives d'avenir pour ce domaine sont variées et nous sommes aujourd'hui aux balbutiements de nouveaux paradigmes. Le véhicule de demain sera autonome, connecté et sans doute vendu comme un service. L'autonomie impose une grande connaissance des environnements (villes, autoroutes, campagnes …) pour lesquels un ensemble de fonctionnalités sont disponibles. Ces mêmes fonctionnalités sont déjà vendues comme des options, modulables au gré du client (comme l'option du pilote automatique chez Tesla Motors). Cette modularité, nécessite de pouvoir s'adapter à chaque configuration, et ce généralement uniquement à l'aide de support logiciel.

Dans ce cadre on parle également de *over-the-air update*, ce qui signifie que le véhicule, comme tout système informatique, doit pouvoir être mis à jour régulièrement sans retour chez le concessionnaire. Ces mises à jour peuvent inclure de nouvelles fonctionnalités

ou modifier celles qui existent déjà, elles peuvent également inclure des correctifs de sécurité ou de sûreté, … Elles sont de plus en plus fréquentes comme on peut le constater dans [4] : un véhicule de la marque Tesla a était mis à jour 117 fois sur la période du 22/06/2012 au 26/02/2016 soit tous les 11,5 jours en moyenne.

Dans ce même rapport [4] on trouve que les véhicules étaient sujet aux mêmes maux que les systèmes informatiques de type PC. Ainsi on a constaté des défaillances de l'affichage, des arrêts intempestifs et la nécessité de redémarrer. Ce qui pour un ordinateur constitue un simple désagrément pour l'utilisateur peut-être un problème critique dans un système embarqué. C'est pourquoi la sûreté de fonctionnement doit s'adapter à ces systèmes changeants pour les rendre résilients.

1.4.1 Modélisation des changements

Comme nous l'avons introduit dans l'exemple précèdent, il est nécessaire de prendre en compte un certain nombre de changements lors de la vie d'un système. On peut diviser les paramètres affectés par ces changements en trois catégories :

- Les caractéristiques applicatives (AC) ;

-Le modèle de fautes (FM) ;

-Les ressources disponibles (R).

Tout changement affectant l'une de ces catégories peut avoir un impact sur les propriétés de sûreté de fonctionnement du système dans sa globalité et plus précisément sur sa capacité à tolérer les fautes. C'est cet ensemble de paramètre que l'on appelle *modèle de changement* et qui nous permet de quantifier les modifications que subit un système. L'impact quant à lui peut être nul ou bien trop important au point de nécessiter le remplacement d'un ou plusieurs mécanismes de tolérance aux fautes.

Une modification des mécanismes de tolérance aux fautes peut donc être due à l'évolution de caractéristiques applicatives, rendant l'application incompatible avec le mécanisme déjà en place. Elle peut également être la conséquence d'une variation du modèle de fautes auquel cas le mécanisme n'est plus adéquat puisqu'il ne permet pas de tolérer les fautes de l'application. Enfin une variation dans les ressources disponibles entraînera l'impossibilité physique de continuer à faire fonctionner un mécanisme déjà en place.

Dans le contexte de ce travail on considère que les caractéristiques applicatives (AC) sont toutes les paramètres intrinsèques aux applications tels que leurs variations peuvent influer sur la compatibilité de l'application avec des mécanismes de tolérance aux fautes. Le modèle de fautes quant à lui regroupe les différents types de fautes qu'il faudra être capable de tolérer si l'on souhaite garantir les propriétés de sûreté de fonctionnement de l'application modélisée. Enfin, les ressources représentent la surconsommation qu'implique le fonctionnement des mécanismes de tolérance aux fautes.

Ainsi, pour choisir un mécanisme cohérent avec une application il nous faut considérer dans un premier temps les mécanismes adéquats avec le modèle de fautes. Puis il faut vérifier leur compatibilité avec les caractéristiques applicatives avant de s'assurer de la capacité du système à fournir les ressources nécessaires à son bon fonctionnement.

Toute variation dans ces paramètres peut invalider le choix initial de mécanisme de tolérance aux fautes. Dans le cas où on relève une incohérence il faudra déclencher le processus d'adaptation afin de garantir au plus vite les propriétés de sûreté de fonctionnement.

1.4.2 Cas des ressources

Des trois catégories énoncées précédemment on peut séparer les ressources des caractéristiques applicatives et du modèle de fautes. On appelle ressources des paramètres tels que la consommation de mémoire vive, de bande passante, de temps d'exécution alloué sur le processeur, …

Chaque fois que l'on modifie une des applications qui composent le système cela affecte les ressources disponibles pour toutes les autres applications. En d'autres termes, alors que le modèle de fautes et les caractéristiques applicatives sont intrinsèques à une application, les ressources sont communes au système. Or, il y a deux façons de considérer les ressources d'un système. Soit ces ressources sont une source de changement, soit elles sont un critère de plus pour sélectionner les mécanismes de tolérance aux fautes.

Dans le cadre de cette thèse, nous considèrerons que les ressources sont le critère final que l'on utilisera pour sélectionner et mettre en place un mécanisme de tolérance aux fautes. Devant la complexité de modéliser l'ensemble des ressources d'un système, qui sera souvent composé de plusieurs entités d'exécution pour mettre en place des stratégies duplex par exemple, nous avons convenu de ne pas traiter cette dimension de changement dans ce travail.

Cette thèse se focalisera donc sur l'étude d'application n'ayant pas de dépendances directes entre elles en ce qui concerne les hypothèses nécessaires à la mise en place de mécanisme de tolérance aux fautes. Pour autant l'ensemble de ce qui sera dit par la suite est valide et extensible si l'on dispose d'un modèle capable d'exprimer les interconnexions entre applications d'un même système.

1.4.3 Problématique globale

Le but de la science est de produire une description la plus complète et exhaustive possible d'un ensemble de phénomènes qui serait fondée sur un ensemble de principes. Pour ce faire, le scientifique dispose d'un ensemble de méthodes qui permettent cette description de la réalité. Ces méthodes varient en fonction de l'approche choisie mais également en fonction des résultats attendus.

D'ordinaire en ingénierie informatique deux approches s'offrent à nous : l'approche descendante et l'approche ascendante. Dans le premier cas il s'agit de partir d'un système complet que l'on raffine en sous-système de plus en plus petit jusqu'à un niveau atomique réalisant ainsi une mise à plat analytique de l'objet étudié. L'approche ascendante quant à elle effectue le chemin inverse, il s'agit donc d'agglomérer un ensemble de sous-système afin de synthétiser un système.

Dans notre cas les deux approches sont nécessaires. Comme nous l'avons décrit précédemment la sûreté de fonctionnement est le principe global et fondateur duquel nous partons. La tolérance aux fautes en est une de ses composantes et nous nous intéressons aux interactions entre applications et mécanismes de tolérance aux fautes lors des changements qui peuvent survenir au cours de la vie opérationnelle de cette application. Afin de passer de la sûreté de fonctionnement à la résilience il nous faut considérer que ces mécanismes ne sont plus définis statiquement lors du développement du système mais qu'ils peuvent être amené à être modifier, composer, réutiliser, etc… Cela nécessite donc un certain nombre d'outils qui nous permettent de développer des applications et des mécanismes de tolérance aux fautes composables. C'est ce besoin de modularité qui nous pousse à orienter nos recherches sur des plateformes basée sur une architecture à composants reposant sur les principes de Component-Based Software Engineering (CBSE). De telle plateforme nécessite une approche ascendante puisque l'on part des composants atomiques qui composent un système afin de construire celui-ci.

On citera par exemple le middleware ROS, qui bien que développé initialement pour la robotique tend aujourd'hui à s'étendre à de nombreux domaines de l'informatique embarqué. C'est ce type de plateforme qui permet la plus grande liberté quant aux possibilités de moduler les composants tant lorsque le système est hors-ligne que lorsqu'il est en cours de fonctionnement. Dans les travaux de [26] on démontre qu'il est notamment possible d'utiliser une décomposition des mécanismes de tolérance aux fautes en trois sous-composants (*Before, Proceed, After)* afin de permettre leur mise à jour de manière optimale. Un exemple de modélisation en utilisant l'architecture ROS est donné dans la figure 5. Sur ce schéma sont représentées les interactions entre un client et un serveur, ce dernier étant protégé par un mécanisme de redondance temporelle noté TR et composé des trois blocs *Before, Proceed* et *After*. Les communications sont gérées par un composant *Proxy* côté client et par le composant *Protocol* côté serveur.

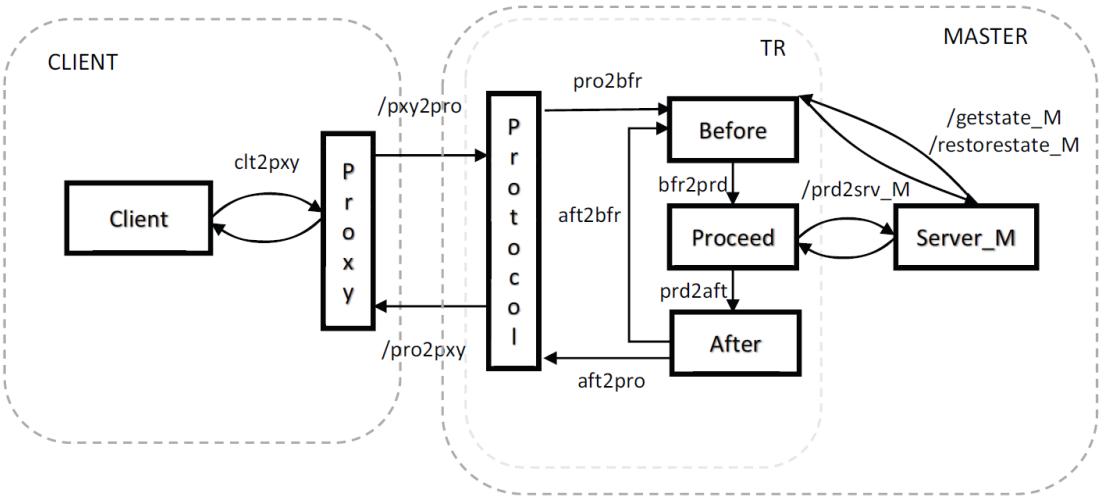


Figure 5 - Réalisation d'un mécanisme de redondance temporelle sous ROS

La modification de l'architecture du logiciel dépend en partie des protocoles de communications qui les relient. Dans le cas de ROS, les composants communiques via des liens (appelés *topics)* sur lesquels sont définis un *publisher* qui publie les messages et des *subscribers* qui sont connecté au *topic* afin de recevoir ces messages. Changer un composant c'est donc s'assurer de déconnecter l'ancien et de reconnecter le nouveau en lieu et place de la version précédente. Il faut également s'assurer que durant cette opération il n'y a pas de perte de message ni de conflit entre les deux versions.

On notera que l'approche que nous choisissons permet de garantir le principe de *Separation of Concern*, c'est-à-dire que la partie fonctionnelle (i.e. l'application) est indépendante de la partie non fonctionnelle assurant la sûreté de fonctionnement (i.e. les mécanismes de tolérance aux fautes). Certes, cette approche présente l'inconvénient de forcer la généricité des mécanismes de tolérance aux fautes empêchant de fait de mettre en œuvre certaines solutions. Toutefois, l'important pour nous est de garantir que le système est capable de changer de mécanisme en fonction des changements qui peuvent survenir dans sa vie opérationnelle.

L'intérêt de cette approche est de garder une bonne vision d'ensemble de la problématique tout en s'assurant que le travail effectué est portable sur des systèmes réels via des solutions technologiques viables et déjà existantes.

1.5 Approche

Comme nous l'avons introduit dans la sous-section précédente, le principe de la *Separation of Concern* tel qu'énoncé par Dijkstra [27] est essentiel à notre approche. Ce principe garantie la modularité des fonctions du logiciel en séparant le développement des applications et le développement des composants garantissant la sûreté de fonctionnement, mais aussi la sûreté de fonctionnement, la qualité de service, …

On suppose également que le système est non seulement bâti suivant ce principe mais qu'en plus nous avons la capacité d'observer un certain nombre de paramètres dont les variations auront un potentiel effet du point de vue de la sûreté de fonctionnement [24] [25]. En cas de changement le système doit également être capable de se moduler pour absorber ces changements.

Dans notre cas ces changements peuvent être de deux types :

- Soit il n'y a pas de changement du point de vue de l'application mais le modèle de fautes varie due à des perturbations environnementales (radiations, humidité, vibrations…) ou des causes internes (vieillissement de semi-conducteur, perte de performances, …)
- Soit ces changements proviennent directement de la partie applicative via des mises à jour.

Dans les deux cas la question que l'on se pose est la suivante : le système est-il résilient à ces changements ? Et pour répondre à cette question il nous faudra passer par différente étapes. Tout d'abord il est nécessaire d'obtenir un modèle fiable qui réunisse l'ensembles des paramètres (caractéristiques applicatives et modèle de fautes) qui donnent à tout instant le profil de l'application que l'on veut rendre sûr de fonctionnement.

Pour ce faire nous partirons d'une classification des mécanismes de tolérance aux fautes afin de proposer une méthode de construction de modèle qui permet de réunir les paramètres nécessaires à la mise en place d'observateur et de déclencheurs de l'adaptation. Cette classification sera nécessaire à la compréhension de la philosophie que nous essayons de développer au travers de cette thèse.

Ensuite, il faudra définir un certain nombre d'outils capables non seulement de rendre possible l'adaptation mais également de mesurer son impact en termes de résilience. Et pour se faire il est nécessaire de développer des métriques spécifiques à la résilience. Ces métriques s'inspirent des concepts forts de la sûreté de fonctionnement tels que la fiabilité, la disponibilité afin de garantir que les résultats obtenus soient interprétables comme la continuité des travaux déjà réalisés dans ce domaine.

En effet, nous proposons dans cette thèse une extension des concepts de la sûreté de fonctionnement afin de faire correspondre ce domaine aux nécessités qui sont celles de systèmes de plus en plus évolutifs et autonomes. Cette extension se traduit par des mesures complémentaires à celle du taux de défaillance, des temps moyens entre défaillance, des taux de réparations,…

Enfin, l'objectif est d'inscrire ce travail dans un processus de développement de système embarqué. Nous souhaitons mettre en évidence le fait que rendre un système résilient a un impact sur sa sûreté de fonctionnement, mais également que d'inscrire cette réflexion dans les premières étapes de développement permet d'obtenir des systèmes capables d'absorber les changements, même imprévus, qui peuvent survenir durant sa vie opérationnelle.

1.6 Etat de l'art

Le travail effectué lors de cette thèse se situe à l'intersection de plusieurs domaines scientifiques. Ainsi on traitera de :

-Tolérance aux fautes et de mécanismes de tolérance aux fautes.

-Tolérance aux fautes adaptative

-Programmation orientée composant

-Informatique Autonome

-Systèmes reconfigurables

Dans cette section nous ferons le tour des avancées techniques de chacun de ces domaines afin de positionner au mieux nos travaux.

1.6.1 Tolérance aux fautes et mécanismes de tolérances aux fautes

La tolérance aux fautes et les mécanismes qui existent pour la garantir est un domaine de recherches complet bénéficiant de nombreuses publications scientifiques. Ainsi nous proposons ici une revue des publications fondatrices sans pour autant prétendre à l'exhaustivité.

C'est au début des années 1950 que les premiers mécanismes de tolérance aux fautes ont commencé à être mis en place. En effet, l'informatique naissante, on s'est aperçu que la faible fiabilité des composants matériels de l'époque provoquait un nombre non-négligeable de défaillances. Afin d'augmenter la fiabilité des systèmes il a était décidé d'utiliser des unités redondantes afin de tolérer un certain nombre de fautes. C'est le cas par exemple de l'UNIVAC 1 [5] ou du moins célèbre calculateur SAPO développé à l'université de Prague qui fut le premier ordinateur conçu pour être tolérant aux fautes [6].

La notion de tolérance aux fautes est donc indissociable de l'informatique pourtant il faudra attendre les années 1990 pour que commencent à se formaliser les premiers principes de la sûreté de fonctionnement. Parmi les pionniers de ce domaine nous citerons les travaux de P. Lee et T. Anderson [7] et de JC. Laprie, A. Avižienis et H. Kopetz [8] dont est issue toute la terminologie relative au domaine de la sûreté de fonctionnement et dont une partie des concepts a été détaillée en section 1.2.1.

Concernant les mécanismes de tolérance aux fautes les plus courants, on en retrouve une description détaillée dans le manuscrit de thèse de A. Armoush [9]. Ce travail propose une analyse des mécanismes de tolérances aux fautes à travers leurs modèles de conceptions. On y retrouve un catalogue des techniques tant matérielles que logicielles permettant leur classification en fonction d'exigences telles que la redondance matérielle, la diversification du logiciel, la disponibilité, leur niveau de couverture… Dans la mesure où nous souhaitons

travailler avec des mécanismes de tolérances aux fautes les plus génériques possibles, nous utiliserons principalement ces mécanismes qui couvrent un champ d'application large puisqu'ils sont issus d'une approche généraliste de la sûreté de fonctionnement.

1.6.2 Tolérance aux Fautes Adaptative

Les avancés en matière de tolérance aux fautes s'orientent donc à présent vers de nouveaux paradigmes tels que la résilience et il en découle de nouveaux défis techniques et donc des nouvelles solutions telles que la tolérance aux fautes adaptative.

La tolérance aux fautes adaptative cherche à combler les lacunes d'une tolérance aux fautes statique. En effet, lorsque l'on parle de tolérance aux fautes statique il est nécessaire d'avoir pleine connaissance et contrôle du système que l'on souhaite protégé. Il faut donc être capable d'envisager toutes les évolutions du logiciel, du matériel, de l'environnement, des utilisateurs… Or dans un monde où les systèmes sont de plus en plus évolutifs et autonomes cela devient impossible. C'est pourquoi ne nombreuse études ont été faîtes concernant les possibilités de s'adapter pour mieux tolérer les fautes.

La première définition du terme *Adaptive Fault Tolerance* (AFT) a été présentée par K.H. Kim et T.F. Lawrence dans leurs recherches [3]. Dans ces travaux l'AFT, est vu comme un moyen de dépasser les concepts de tolérance aux fautes mais représente également un challenge technique qui semblait à l'époque hors d'atteinte. Bien entendu la définition citée en section 1.2.3 ne pouvait à l'époque pas prendre en compte la diversité et la puissance des systèmes d'aujourd'hui. Les systèmes actuels sont non-seulement distribués mais également hyper connectés et ce même pour des applications simples. Avec l'essor de la domotique et de l'IoT (*Internet of Things*) nous disposons de systèmes évolutifs (avec ajout de nouveaux éléments et de nouvelles fonctionnalités) dont on souhaite pouvoir garantir un niveau de fonctionnement acceptable.

Le concept de l'AFT dépasse donc la sphère des systèmes embarqués critiques et nécessite pour fonctionner de pouvoir mettre en œuvre des architectures spécialement développées pour. O. Marin décrit dans [10] une architecture distribuée tolérante aux fautes basée sur un ensemble de systèmes distribués décentralisés. L'objectif est de pouvoir tolérer les changements de configurations, d'environnement et d'exigences de sûreté de fonctionnement afin de garantir la sûreté de fonctionnement du réseau. L'exemple donné dans ce document concerne des systèmes multi-acteurs tels qu'une équipe de policiers, pompiers,… devant gérer un état de crise. Bien entendu, chaque agent ayant son propre rôle d'importance différente il faut pouvoir considérer la criticité de chacun de ces rôles laquelle peut varier en fonction des différentes phases de l'intervention. Afin de diminuer les coûts de fonctionnement chaque mécanisme de tolérance aux fautes doit être adapté à chaque agent et à son rôle. Les auteurs proposent l'architecture DARX (basée sur Java) pour concevoir de tels systèmes. Un des points important est la nécessité d'avoir un *Manager* capable de prendre les décisions qui s'imposent notamment concernant les stratégies de réplications qu'il faut mettre en place pour chaque groupe d'agents. Cette entité est nécessaire lorsque l'on

parle d'AFT puisqu'il faut toujours avoir un système dont le niveau d'abstraction est suffisant pour observer les changements (*Monitoring*) et prendre des décisions pour garantir la sûreté de fonctionnement.

1.6.3 Programmation orientée composant

Une des avancées techniques qui permet la mise en place de l'adaptabilité des mécanismes de tolérance aux fautes est l'utilisation d'architecture orientée composant. Cette philosophie de développement est imaginée en 1968 par D. Mcllroy qui la décrit pour la première fois lors d'une conférence de l'OTAN. Il formalisera ce concept dans un livre en 1986 : Object Oriented Programming – An evolutionary approach [11].

Depuis, de nombreuses solutions ont vu le jour parmi lesquelles Entreprise JavaBeans (de Sun Microsystems), .NET Remoting (de Microsoft), CORBA (de Object Management Group) mais également des technologies plus grand public telles que Unity (de Unity Technologies) ou Unreal Engine (de Epic Games).

Les architectures à composants sont donc partout, des systèmes embarqués aux technologies de divertissement grand public. Dans tous les cas, la définition la plus couramment acceptée est celle donnée par Szyperski [12] : « Un composant logiciel est une unité de composition dont les interfaces sont spécifiées et les dépendances contextuelles sont explicites. Un composant peut être déployé seul ou mis en composition avec d'autres composants. » Bien entendu chaque architecture propose ses spécificités permettant de travailler avec des composants à granularité variables, avec ou sans spécifications de contraintes non-fonctionnelles, …

Dans cette thèse nous ne traiterons pas d'une technologie en particulier. Il sera considéré que compte tenu de la diversité des solutions nous travaillerons dans l'optique d'intégrer notre démarche pour des systèmes dont la conception est orientée composant.

1.6.4 Informatique Autonome

L'informatique autonome correspond à la capacité d'un système à gérer lui-même un ensemble de ressources distribuées dont il dispose, à s'adapter à des changements imprévisibles tout en veillant à ce que cela reste transparent pour l'utilisateur. Ce concept a était définit par IBM en 2001 [13] dans le but de réussir à créer des systèmes autonomes capables de gérer des situations dont la complexité devient de plus en plus importante.

Ces systèmes se doivent d'être :

-Automatique : le système doit pouvoir contrôler ses fonctions internes et doit donc être capable de démarrer et de s'arrêter seul ainsi que de prendre des décisions sans interventions externes.

-Adaptatif : Le système doit être capable d'adapter ses fonctionnalités, sa configuration et son état en fonction de l'évolution de son environnement mais également des ressources dont il dispose.

-Conscient : Le système doit être capable d'observer son contexte opérationnel afin de déclencher automatiquement les adaptations nécessaires.

L'objectif est de déléguer un ensemble de décisions dont la complexité est exponentielle au système lui-même. Ce faisant on s'assure de diminuer les coûts de développement et d'opération comme le décrit R. Sterrit [14]. L'informatique autonome englobe également tous les concepts *Self-** tels que le Self Healing qui est la propriété pour un système de se réparer ou encore le Self Learning qui est la capacité pour un système à apprendre à réagir en fonction d'un ensemble de situations déjà apparues.

1.6.5 Systèmes reconfigurables

Enfin, un système adaptable doit être capable de se reconfigurer comme nous l'avons dit dans la section précédente. Ainsi on voit apparaitre de plus en plus d'architectures telles que celle décrite dans les travaux d'A. Boudjadar [15] qui présente une architecture capable de modéliser et d'analyser l'ordonnancement d'un système. L'objectif est de partir d'un système formé de composant applicatif et d'étudier en ligne la faisabilité de l'exécution de ces composants sur les calculateurs disponibles. Comme nous l'avons exprimé précédemment, afin de pouvoir s'adapter, un système doit être capable d'être modéliser. Ils utilisent l'outil UPAAL [16] qui permet de modéliser les politiques d'ordonnancement des composants, les fréquences des CPU et les comportements des composants. Ces paramètres sont autant de variables sur lesquelles le système est capables d'agir afin d'optimiser les coûts d'utilisation tout en garantissant les propriétés de temps réels.

On note également qu'il existe des systèmes dont même la couche matérielle peut être modifiée en ligne. C'est le cas en particulier des FPGA (*Field Programmable Gate Array*), qui sont des circuits intégrés dont la configuration peut être changée afin de convenir au besoin de l'utilisateur. Les progrès technologiques en matière de FPGA permettent aujourd'hui de reconfigurer dynamiquement les plateformes matérielles afin d'adapter leurs fonctionnements pour répondre à des évènements ponctuels [17].

1.6.6 Conclusion

Les avancées technologiques permettent aujourd'hui de proposer des systèmes capables de s'adapter aux besoins de leurs utilisateurs et ce quel que soit leur domaine d'application. Comme nous l'avons dit précédemment, un système doit pouvoir s'adapter, mais pour se faire il est nécessaire de mettre en place des modèles et des mesures afin d'orienter au mieux les modifications imposées par des changements (environnement, nouvelles fonctionnalités, mises à jour, …). C'est ce que nous proposerons dans le chapitre suivant.

Chapitre 2 – Modélisation des systèmes résilients

"All models are wrong but some are useful"
George Box (1919-2013) - Science and Statistics.

2.1 Introduction

Ce chapitre aborde la question de la modélisation d'un système résilient. Le modèle proposé répond à des besoins spécifiques dont l'objectif est de permettre de mesurer la résilience d'un système. Il s'agit donc de modéliser les composants fonctionnels applicatifs que nous appellerons tout simplement applications et non fonctionnels (ici des mécanismes de tolérance aux fautes) ainsi que leurs interactions.

Ce chapitre n'a pas pour objectif de produire un modèle exhaustif à utiliser sur n'importe quel système mais une méthode de modélisation qui permet d'adapter le modèle à chaque système. Le but est de fournir des principes et des outils de modélisation afin de permettre aux développeurs de créer le modèle qui conviendra le plus à leurs besoins. Plus la connaissance du système sera précise et plus le modèle produit avec ces outils pourra être complet.

Le premier jalon de cette modélisation est de réussir à représenter avec justesse les mécanismes de tolérance aux fautes. Pour ce faire nous partirons d'une classification basique de ces mécanismes. Ensuite, nous verrons comment passer de cette classification à un modèle utilisable pour les applications. Enfin, nous traiterons de la modélisation des interactions entre les applications et les mécanismes de tolérance aux fautes.

2.2 De la classification des mécanismes de tolérance aux fautes

Comme dit précédemment, nous allons nous appuyer sur une classification des mécanismes de tolérance aux fautes. En effet, classifier ces mécanismes c'est avant tout repérer quelles sont les différences fondamentales entre eux. Or ces différences sont autant de paramètres qu'il nous faudra prendre en compte dans notre modèle. Nous choisirons ici l'approche de de classification proposée dans les travaux de M. Stoicescu [20]. Nous allons à présent détailler la démarche associée à cette méthode de classification.

Le développement d'un système sûr de fonctionnement implique une analyse de sûreté qui conduit à l'identification de mécanismes de défense. Les solutions possibles pour rendre le système sûr de fonctionnement consistent alors à mettre en place un ou plusieurs mécanismes de tolérance aux fautes. Ce choix d'un mécanisme parmi d'autres s'effectue selon plusieurs critères qui seront autant de paramètres à faire figurer dans le modèle que nous cherchons à produire.

L'analyse de sûreté de fonctionnement afin d'identifier tous les types de fautes auxquels notre système devra faire face. Ces analyses de sûreté (Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité) conduites sur une application particulière nous permettent de lister tous les modes de défaillances et leurs causes (i.e. les fautes subies) et leurs effets sur les autres composants du système.

Pour l'exemple, prenons le cas d'une application dont on suppose qu'elle est soumise à des fautes par crash uniquement. Cela signifie que lorsque l'application est fautive elle cesse de produire un service. Une application qui n'est sujette qu'à des fautes par crash vérifie donc la propriété suivante :

« Le service fournit par l'application est correct (la valeur renvoyée est correcte et dans un intervalle de temps acceptable) ou bien (XOR) l'application ne répond pas ».

Une fois l'ensemble des fautes identifiées pour un système, on peut s'attacher à regarder comment traiter ces fautes, c'est-à-dire quels mécanismes de tolérance aux fautes doivent être installés pour garantir que notre application soit sûre de fonctionnement. Chaque mécanisme permet de traiter un ou plusieurs types de fautes, et chacun d'eux impose de faire des hypothèses plus ou moins fortes sur l'application qu'il doit protéger.

Reprenons le cas des fautes par crash, pour tolérer ce type de faute nous utilisons un ensemble de stratégies duplex. Ces stratégies ont en commun l'idée qu'il faut avoir au moins une copie redondante de l'application à protéger pour qu'en cas de défaillance de l'application primaire la copie (appelée secondaire) puisse prendre le relais.

Ces stratégies duplex, peuvent se spécialiser en fonction des besoins et de leurs implémentations. Afin d'illustrer ceci nous traiterons de deux types de mécanismes duplex classiques : la « redondance chaude » et la « redondance froide ».

La redondance chaude signifie que les deux copies identiques en tout point d'une même application sont exécutées en parallèle. En cas de crash du primaire, le secondaire prend presque immédiatement le relais car il est la copie exacte du primaire (mêmes entrées et sorties, même état interne). Ce type de redondance est utilisé dans le cadre de système ou la disponibilité doit être très élevée. Cette stratégie porte aussi l'appellation LFR pour Leader-Follower Replication.

A contrario la redondance froide n'oblige pas d'exécuter en simultané deux copies d'une même application. La stratégie est d'envoyer régulièrement des checkpoints (des instantanés de l'état de l'application) à la copie pour qu'en cas de crash celle-ci puisse redémarrer à partir du dernier checkpoint. Ce type de redondance est particulièrement utilisé dans les systèmes dont la consommation de ressources énergétiques est critique (e.g. satellites, capteurs autonomes,…). Cette famille de mécanismes est aussi appelé PBR pour Primary Back-up Replication.

Nous l'avons vu, ces deux stratégies permettent de tolérer le même type de fautes (i.e. les crashs), pourtant leurs spécificités propres impliquent que l'on fasse certaines hypothèses sur la nature de l'application à protéger.

Reprenons le cas des mécanismes LFR. Comme dit précédemment les copies doivent être identiques même si elles sont exécutées sur deux environnements matériels différents. En conséquence le code de la dite application ne peut être indéterministe. L'indéterminisme signifie que les sorties (valides) peuvent être différentes pour les mêmes entrées. Cela implique que l'historique des états des répliques est différent. Un observateur lié à la première réplique ne pourra pas comprendre le comportement du système après défaillance de celle-

ci puisque les sorties après basculement d'une réplique à l'autre ne seront pas cohérentes avec celles perçues avant la défaillance de la première réplique.

On voit donc ici que l'hypothèse « l'application est déterministe » joue un rôle fondamental dans le choix d'un mécanisme LFR lorsqu'une stratégie duplex est envisagée pour tolérer les fautes par crash.

De manière analogue, l'utilisation d'un mécanisme de type PBR requiert la mise en place de checkpoint, c'est-à-dire que nous devons être capables de capturer l'état d'une application si celui-ci existe. Ainsi l'hypothèse « l'application à un état et il est accessibles » doit pouvoir être vérifiées dans le modèle que nous cherchons à produire.

C'est cette réflexion que l'on peut résumer par le tableau suivant extrait des travaux de Mirùna Stoicescu [20]:

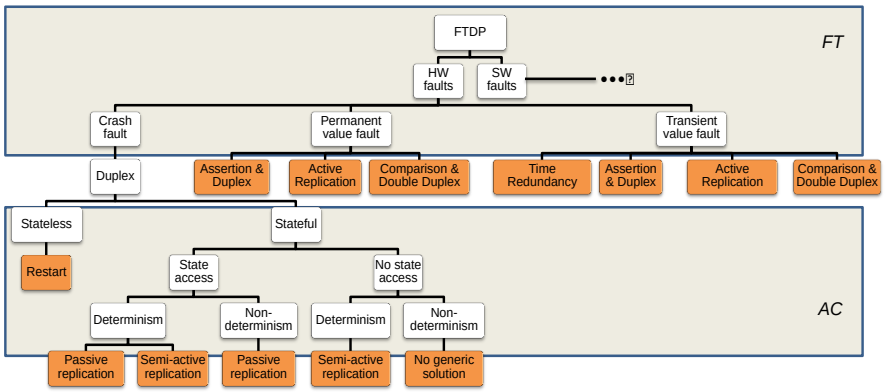


Figure 6 - Classification des mécanismes de tolérance aux fautes

Cette première approche est un arbre qui permet de déterminer que lorsque l'on veut tolérer des fautes matérielles, notamment des fautes par crash il faut utiliser des mécanismes de redondance de type duplex. Dans le cas de mécanisme ayant un état mémoire et si celui-ci est accessible on peut utiliser au choix des mécanismes duplex actif ou semi actif si l'application est déterministe, et si ce n'est pas le cas il faut utiliser des techniques de réplication passives.

Cette classification nous permet d'obtenir une première base de paramètres pour notre modèle mais ce dernier pourra être augmenté à l'aide d'autres analyses. On ajoutera ainsi de nouveaux modèles de fautes et donc de nouvelles caractéristiques applicatives.

Cela signifie que dans le modèle doit non seulement figurer le modèle de faute (que nous noterons FM) mais également les caractéristiques applicatives (notées AC) sur lesquelles reposent les hypothèses nécessaires à la mise en place des mécanismes de tolérance aux fautes.

Tout cela décrit une réflexion théorique qui permet de définir deux ensembles de propriétés : FM et AC. Cette classification considère au départ un modèle de fautes (crash, valeur, …) et le raffine pour trouver in fine des solutions de tolérance aux fautes. Dans ce raffinement on voit apparaitre des caractéristiques sur les applications (déterminisme, accès à l'état, …). Les feuilles orange de cet arbre sont une solution possible pour chaque branche.

2.3 De la classification au modèle

On peut résumer cette approche par la figure 7 :

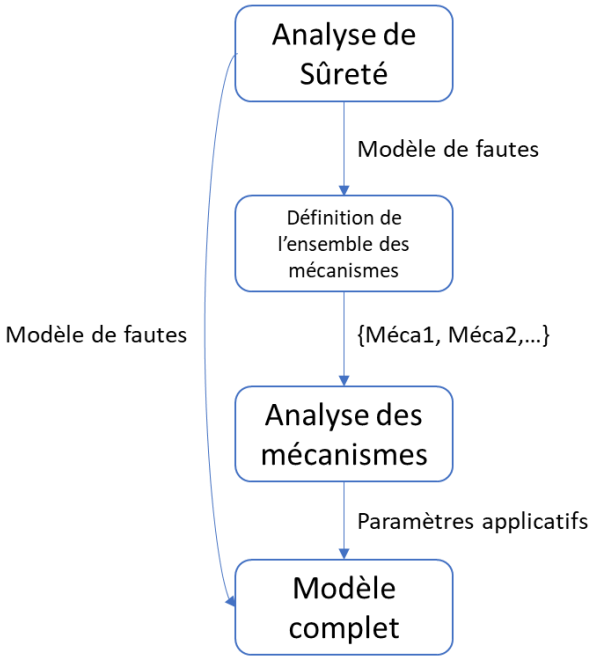


Figure 7 - Principe de création du modèle

Il faut donc partir d'un ensemble de type de fautes afin de pouvoir choisir les mécanismes que nous souhaitons mettre en place. De ces mécanismes on déduit un ensemble de paramètres applicatifs essentiels au fonctionnement des mécanismes. Le modèle est quant à lui battit à partir du modèle de fautes et des paramètres applicatifs.

2.3.1- Le modèle de fautes

L'approche vue précédemment permet de mieux comprendre les choix de modélisation qui ont été fait dans ce travail. Les deux types de paramètres du modèle ayant été identifiés il faut à présent bâtir concrètement un modèle avec lequel nous pourrons représenter les applications et les mécanismes de tolérance aux fautes.

Pour cette étude nous nous concentrerons sur trois types de faute :

-les fautes par crash.

-les fautes en valeurs.

-les fautes par omission.

Partons d'un ensemble de trois mécanismes de tolérance aux fautes. Ces mécanismes sont génériques mais représentatifs des principales techniques de tolérance aux fautes. Les deux premiers ont déjà été abordés précédemment, il s'agit des mécanismes implémentant

les stratégies de redondance chaude et froide, respectivement LFR et PBR. Le troisième mécanisme est appelé mécanisme de redondance temporelle ou TR pour Time Redundancy.

TR permet de tolérer les fautes en valeurs (appelées également fautes transitoires) en exécutant deux fois la même application puis en comparant les résultats. En cas d'égalité la réponse est validée, en cas de désaccord plusieurs stratégies sont possibles (une troisième itération, un reset de l'application, un abandon de la requête,…).

On considère donc que ce mécanisme permet de tolérer les fautes en valeurs. De plus, puisqu'il nécessite d'exécuter deux fois une requête on considère qu'il est capable de tolérer les fautes par omission. En effet, si un des deux résultats est omis alors le mécanisme détecte qu'aucune comparaison ne peut être effectuée et qu'il y pas par conséquent une faute.

Nous limiterons notre étude à ces trois types de fautes. Cependant, lors d'une application sur cas d'étude il faudra extraire d'une analyse de sûreté de fonctionnement l'ensemble de toutes les fautes que peut subir une application. Une fois ceci fait nous pouvons passer à la recherche des caractéristiques applicatives qui seront pertinentes pour notre modèle à partir des mécanismes de tolérance aux fautes disponibles.

2.3.2 Les caractéristiques applicatives

Il nous faut à présent extraire de l'ensemble des FTMs toutes les caractéristiques applicatives qui sont nécessaires dans la formulation des hypothèses relatives à la mise en place de ces FTMs.

Commençons par l'étude du mécanisme LFR. L'analyse que nous avons conduite dans la section précédente nous a permis de déterminer que si l'on veut protéger une application avec ce mécanisme il faut à minima que l'application soit déterministe. On note également que pour détecter le crash d'une application il faut que celle-ci soit silencieuse sur défaillance. En effet, en cas de crash, si l'application continue de fournir des preuves de vie rien ne permet de vérifier qu'elle a crashé.

En toute rigueur tout mécanisme qui ne considère que le crash dans son modèle de fautes impose l'hypothèse d'une auto testabilité parfaite du composant applicatif, c'est-à-dire une couverture de 100% des mécanismes de détection internes à ce composant. On peut représenter le mécanisme LFR de la manière suivante :

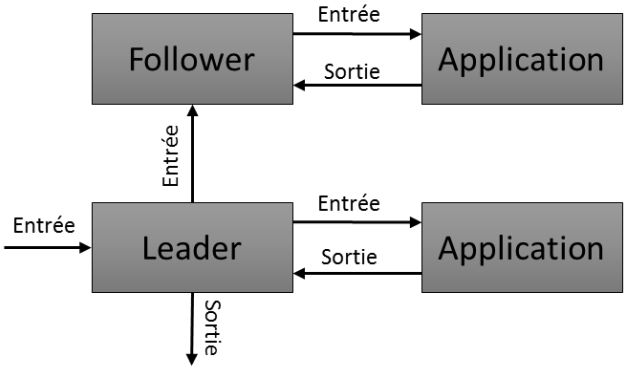


Figure 8 - Modèle de redondance semi-active (Leader Follower Replication)

La copie *Leader* reçoit l'entrée et la transmet au *Follower*. Les deux envoient ensuite cette requête aux composants applicatifs qui se chargent de l'exécution (ici en parallèle) avant de retourner une valeur de sortie. Seule la copie *leader* transmet une réponse au client tant qu'il fonctionne. En cas de défaillance c'est au *follower* de s'en charger.

Concernant le mécanisme PBR, il nécessite l'accès à l'état de l'application afin de pouvoir capturer des états de reprise du composant (notion de checkpoint). Bien entendu, comme pour LFR, l'application associée à ce mécanisme doit être silencieuse sur défaillance pour les même raisons que celles évoquées précédemment pour tout mécanisme duplex tolérant le crash. Le fonctionnement est similaire au mécanisme LFR si ce n'est que la copie dite *secondaire* se contente de sauvegarder les états de la copie primaire comme on peut le voir sur la figure suivante :

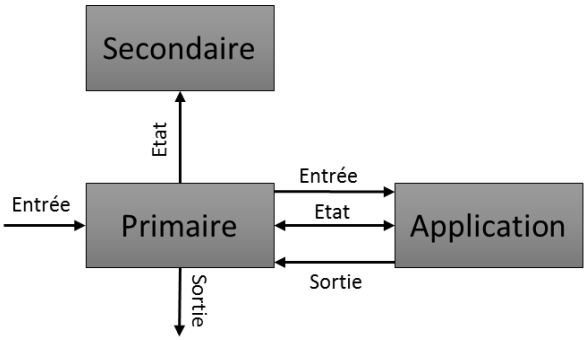


Figure 9 - Modèle de redondance passive (Primary Backup Replication)

Le mécanisme TR fonctionne par comparaison de résultats. Or, pour que les deux résultats successifs du traitement d'une même requête soient égaux l'application se doit d'être déterministe. De plus, lorsque l'application exécute une requête, le résultat peut dépendre de l'état de l'application. Dans ce cas, il nous faut être capable de réinitialiser l'état de l'application afin que les deux exécutions se déroulent dans le même contexte (même état, même entrée). Dans le cas de l'utilisation de TR, il n'y a qu'une seule copie qui effectue deux fois les requêtes d'entrée :

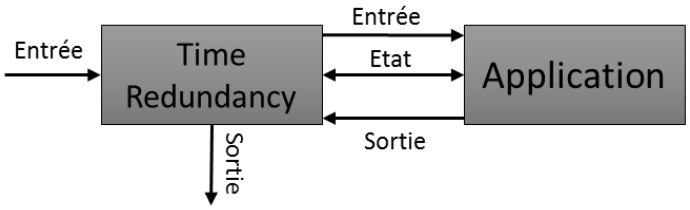


Figure 10 - Modèle de redondance temporelle (Time Redundancy)

Cette analyse permet de déterminer trois caractéristiques applicatives sur lesquelles reposent les hypothèses décrites précédemment pour l'ensemble des mécanismes considérés dans l'exemple :

-Le déterminisme.

-L'accès à l'état.

-La capacité de l'application à demeurer silencieuse sur défaillance.

En conclusion nous avons extrait six paramètres pour établir le modèle : trois caractéristiques applicatives et trois types de fautes. La méthode vue dans cette partie permet donc d'énoncer un ensemble de paramètres à partir de l'ensemble des FTMs disponible. Le modèle obtenu dépend donc de l'ensemble des mécanismes de tolérance aux fautes disponibles. Une extension de cet ensemble peut conduire à l'énoncé de nouveaux paramètres

La partie suivante sera dédiée à la représentation de ces paramètres et à l'instanciation des composants fonctionnels dans le modèle ainsi qu'à la définition des propriétés qui caractérisent les interactions entre fonctionnel et non fonctionnel.

2.4 Modéliser les composants

2.4.1 Le composant fonctionnel et son profil

Les six paramètres identifiés précédemment seront utilisés durant la suite de cette thèse. Pour plus de clarté nous utiliserons les notations suivantes pour les caractéristiques applicatives :

-DT=1 si l'application est déterministe, DT=0 sinon.

-SA=1 si l'état l'application est accessible, SA=0 sinon.

-FS=1 si l'application est silencieuse sur défaillance, FS=0 sinon.

Les modèles de faute correspondent à la notation suivante :

-C=1 si l'application subit des fautes par crash, C=0 sinon.

-O=1 si l'application subit des fautes par omission, O=0 sinon.

-V=1 si l'application subit des fautes en valeur, V=0 sinon.

Le choix des valeurs affectées à chaque paramètre est une étape critique pour la suite. Pour qualifier les interactions entre applications et mécanismes de tolérance aux fautes de la manière la plus générique possible il faut que l'hypothèse la plus forte soit affectée à la valeur la plus haute. On dit qu'une hypothèse h1 est plus forte qu'une hypothèse h2 si tous les mécanismes qui acceptent h2 acceptent aussi h1. Prenons l'exemple du déterminisme, si on suppose que l'application n'est pas déterministe alors cela est équivalent à ne pas faire d'hypothèse. En effet du point de vue des mécanismes de tolérance aux fautes, si un mécanisme est compatible avec une application non-déterministe il est en particulier compatible avec une application déterministe.

A noter que l'hypothèse la plus forte est souvent la plus contraignante du point de vue du développement de l'application. Par exemple, il est plus difficile de s'assurer que l'application est silencieuse sur défaillance (h1 : FS=1) que de ne pas avoir à le vérifier (h2 : FS=0). Elle impose en effet une programmation défensive plus robuste par l'introduction d'assertions exécutable, de traitement d'exceptions, de contrôles systématiques des codes de retour de fonctions, etc… La validation doit être plus conséquente pour évaluer la couverture de détection et donc mesurer l'auto testabilité de ladite application.

Afin d'assurer la cohérence du modèle on s'assurera que tout ajout de paramètres dans le modèle suivra la même logique et donc la même représentation.

L'ensemble de ces paramètres seront regroupés dans un couple de deux vecteurs. Le premier contiendra les caractéristiques applicatives et le second le modèle de fautes. Ainsi on utilise la notation suivante:

$$A_i = \left(\begin{pmatrix} a_{i,1} \\ a_{i,2} \\ \dots \\ a_{i,m} \end{pmatrix}, \begin{pmatrix} f_{i,1} \\ f_{i,2} \\ \dots \\ f_{i,p} \end{pmatrix} \right)$$

Avec A_i une application, $a_{i,m}$ les caractéristiques applicatives et $f_{i,p}$ les types de fautes subit ou non par l'application. Ce qui une fois instancié pour l'exemple avec les mécanismes {PBR, LFR, TR} nous donne :

$$\mathrm{Ai} = \left(\begin{pmatrix} DT \\ SA \\ FS \end{pmatrix}, \begin{pmatrix} C \\ O \\ V \end{pmatrix} \right)$$

Prenons l'exemple d'une application A déterministe (DT=1), dont nous n'avons pas accès à l'état (SA=0), mais qui est silencieuse sur défaillance (FS=1). On note que cette application est soumise à des fautes par crash (C=1) et rien d'autre (O=0 et V=0). On peut la représenter de la manière suivante :

$$A = \left(\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right)$$

Cet ensemble de paramètres auxquels on affecte des valeurs pour une application à un instant donné est appelé un *profil*. C'est ce profil qui va être impacté lors des changements qui surviennent au cours de la vie du système et c'est également ce profil qui va nous servir à déterminer quels sont les mécanismes que nous devons et pouvons utiliser.

Un profil est donc l'instanciation dans un modèle donné d'une. Ce profil est susceptible de changer au cours du temps.

La question que l'on se pose à présent est la suivante : Quel mécanisme de tolérance aux fautes peut être utilisé pour garantir la sûreté de fonctionnement de A ?

2.4.2 Cohérence : compatibilité et adéquation

Dans le cadre de l'association entre un mécanisme de sûreté de fonctionnement et une application il faut vérifier deux choses :

- Le mécanisme doit tolérer les fautes de l'application.
- L'association entre les deux doit être possible.

Si ces deux propriétés sont vérifiées alors l'association est cohérente. Soit une application A et un mécanisme de tolérance aux fautes FTM la formalisation de ces propriétés est la suivante :

Si FTM tolère l'ensemble des fautes subies par A alors les deux sont en **adéquation**.

FTM peut également couvrir d'autres fautes qui ne font pas partie du modèle de fautes de A, dans ce cas il y a toujours adéquation. En revanche si le mécanisme de tolérance aux fautes n'offre qu'une tolérance partielle aux fautes de l'application il n'y a plus adéquation. Cette vision manichéenne est nécessaire puisque l'objectif est de rendre le système sûr de fonctionnement.

Pour autant vérifier l'adéquation n'est pas suffisant. Nous avons vu dans les sections précédentes que nous avons besoin de faire des hypothèses sur les caractéristiques applicatives de l'application pour que l'association entre composant fonctionnel et mécanisme de sûreté de fonctionnement puisse être viable. Ainsi on définit la compatibilité comme suit :

Si FTM accepte les propriétés applicatives de A alors les deux sont **compatibles**.

Et enfin on peut définir la cohérence :

Si FTM et A sont compatibles et en adéquation alors ils sont **cohérents**.

En cas d'absence d'adéquation le système n'est plus sûr de fonctionnement comme nous l'avons vu, cela n'a donc pas un impact immédiat quant au fonctionnement du système tant qu'aucune faute ne survient. En revanche si la compatibilité est brisée les conséquences peuvent être immédiates. C'est le cas de l'exemple sur la figure 4 où le système est protégé jusqu'à l'instant t2 puis il devient fragile pendant une période dt2 nécessaire à la mise en place d'un nouveau mécanisme de tolérance aux fautes. L'enjeu est donc de mesurer et de réduire cette période de fragilité et c'est ce que nous aborderons dans les chapitres 4 et 5.

Prenons à présent l'exemple du mécanisme TR de réplication temporelle. Si l'application à laquelle il est associé devient non déterministe à la suite d'une mise à jour par exemple, alors la comparaison entre les deux répliques détectera potentiellement une erreur en comparant deux réponses pourtant valides. En conséquent l'application devient muette ce qui équivaut à un crash. Cette incompatibilité conduit donc à l'arrêt de l'application bien que celle-ci ne soit pas défaillante. Ceci illustre l'importance de la compatibilité pour ne pas déclencher systématiquement de fausses alarmes.

L'objectif est donc de vérifier la cohérence entre les mécanismes et les composants pour s'assurer non seulement de la sûreté de fonctionnement de l'application mais également de son bon fonctionnement tout court. Pour ce faire, nous avons besoin d'inclure les interactions entre application et mécanismes dans notre modèle afin de pouvoir mesurer l'impact des changements qui peuvent survenir au cours de la vie du système.

2.4.3 Sur la composition des mécanismes.

La solution pour implémenter la tolérance aux fautes adaptative retenue dans le contexte de cette thèse est une approche d'architecture logicielle basée sur des composants. Cela permet non seulement de garantir la séparation entre fonctionnel et non fonctionnel mais également d'agglomérer plusieurs mécanismes entre eux. On notera FTM1+FTM2 l'association de deux mécanismes FTM1 et FTM2.

Cette association entre mécanismes n'est pas évidente a priori, en effet considérons une application A associée à un FTM1 que l'on notera FTM1◊A. Si l'on souhaite ajouter un second mécanisme que nous noterons FTM2 pour obtenir l'association FTM2◊(FTM1◊A) notée (FTM1+FTM2)◊A alors les hypothèses que nous devons faire pour ajouter ce mécanisme de tolérance aux fautes doivent être vérifiées sur le composite (FTM1◊A) et non plus sur A uniquement.

Prenons le cas des mécanismes LFR et TR. On souhaite les composer pour assurer une tolérance à la fois aux fautes par crash et aux fautes en valeurs. Il y a deux solutions, la première est d'utiliser d'abord un mécanisme de type LFR puis d'utiliser sur chaque copie *leader* et *follower* un mécanisme de réplication temporelle TR. On effectue donc LFR puis TR et on note cette composition LFR+TR.

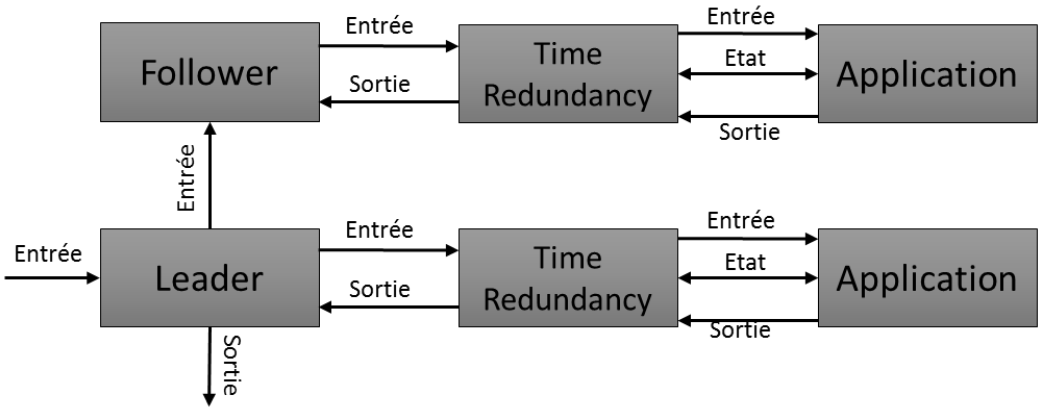


Figure 11 - Modèle de composition LFR+TR

Cette composition permet de nous assurer que chaque requête est effectuée sur les deux copies et qu'en plus elles y sont répliquées. De plus, en cas de crash du *leader,* le *follower* continue d'assurer la tolérance aux fautes en valeurs. Ce mécanisme nécessite que l'application soit déterministe (contrainte LFR et TR), qu'elle soit silencieuse sur défaillance (contrainte LFR) et que l'on puisse accéder à son état (contrainte TR).

Regardons à présent la composition inverse, c'est-à-dire TR+LFR. Dans ce cas, on effectue d'abord la réplication temporelle avant l'utilisation de la redondance physique. Cette composition implique également que le mécanisme LFR sur lequel on connecte le mécanisme TR soit capable de fournir l'état de l'ensemble LFR◊A. On peut le représenter de la manière suivante :

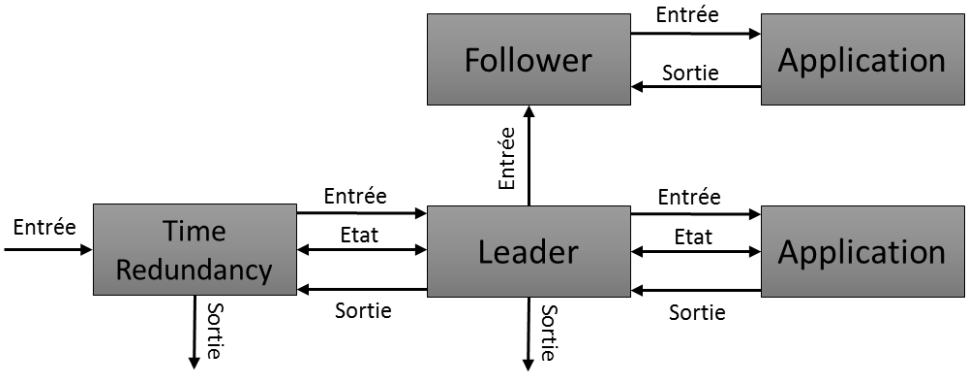


Figure 12 - Modèle de composition TR+LFR

On remarque plusieurs problèmes concernant cette association. Premièrement, en fonctionnement nominal la copie *follower* ne bénéficie d'aucun traitement en cas de faute en valeurs. On effectue deux fois une requête d'entrée sans pour autant revenir à un état précèdent. Il y a donc une désynchronisation des deux copies *leader* et *follower*, ce qui est contraire aux principes de fonctionnement du LFR. De plus, en cas de crash du *leader* la copie *follower* reprend la main mais sans pouvoir garantir la tolérance aux fautes en valeurs puisqu'il n'y a pas de mécanisme TR sur cette copie.

De ces observations on peut déduire que FTM1+FTM2 n'est pas équivalent à FTM2+FTM1. Par exemple effectuer d'abord une réplication avant de dupliquer l'exécution

sur une deuxième copie (TR+LFR) n'est pas équivalent à répliquer les deux exécutions sur les deux copies (LFR+TR)

En résumé, la composition entre une application et un mécanisme produit un composite dont les caractéristiques applicatives et le modèle de fautes sont différents de l'application initiale. Chaque mécanisme de tolérance aux fautes étant différent, les propriétés du composite changent en fonction du mécanisme utilisé. Enfin, la conformité de la composition FTM1+FTM2 doit être vérifiée et dépend de l'implémentation des mécanismes.

Dans la suite nous utiliserons par exemple une de ces compositions qui consiste en l'association de LFR et de TR. Cette association crée un nouveau mécanisme qui doit être étudié afin de vérifier qu'il n'ajoute pas d'hypothèses sur les caractéristiques applicatives.

L'étude de ces compositions ne fait pas partie du périmètre de cette thèse. Nous utiliserons d'autres associations entre mécanismes dans la suite de cette thèse. Toutes ces compositions ont été étudiées au préalable afin de vérifier qu'il est possible de créer ces nouveaux mécanismes.

2.5 Modéliser les interactions entre composants

2.5.1 Modélisation des FTMs

La modélisation des mécanismes de tolérance aux fautes répond à la même notation que celle utilisée pour les composants fonctionnels. Cependant nous ne parlons plus ici de caractéristiques propres aux mécanismes mais d'exigences faîtes au regard de l'application à protéger :

-DT=1 si l'application doit être déterministe, DT=0 sinon.

-SA=1 si l'état de l'application doit être accessible, SA=0 sinon.

-FS=1 si l'application doit être silencieuse sur défaillance, FS=0 sinon.

De façon similaire on utilise pour les modèles de faute la notation suivante :

-C=1 si le mécanisme tolère les fautes par crash, C=0 sinon.

-O=1 si le mécanisme tolère les fautes par omission, O=0 sinon.

-V=1 si le mécanisme tolère les fautes en valeur, V=0 sinon.

L'ensemble de trois mécanismes (PBR, LFR et TR) qui ont été étudié dans la partie précédente peuvent être résumés dans le tableau suivant :

Hypothèses		PBR	LFR	TR	LFR+TR
	Déterminisme		✓	✓	✓

Caractéristiques Applicatives (AC)	Accès à l'état	✓		✓	✓
	Silence sur défaillance	✓	✓		✓
Modèle de fautes (FM)	Crash	✓	✓		✓
	Omission			✓	✓
	Valeur			✓	✓

Tableau 1 - Hypothèses nécessaires à la mise en place de mécanismes de tolérance aux fautes

Un mécanisme de tolérance aux fautes peut donc être représenté par deux vecteurs, le premier est le vecteur des paramètres applicatif et le second le vecteur du modèle de fautes. La généralisation de cette représentation est donc la suivante :

Soit FT_j un mécanisme de tolérance aux fautes, soit $b_{j,m}$ un ensemble de m caractéristiques applicatives, soit $ft_{j,p}$ un ensemble de p types de fautes. On peut représenter FT_j sous la forme vectorielle suivante :

$$FT_j = \left(\begin{pmatrix} b_{j,1} \\ b_{j,2} \\ \dots \\ b_{j,m} \end{pmatrix}, \begin{pmatrix} ft_{j,1} \\ ft_{j,2} \\ \dots \\ ft_{j,p} \end{pmatrix} \right)$$

En utilisant ce type de notation on peut définir un modèle générique de FTM qui serait le suivant :

$$FT_j = \left(\begin{pmatrix} DT \\ SA \\ FS \end{pmatrix}, \begin{pmatrix} C \\ O \\ V \end{pmatrix} \right)$$

Une fois les valeurs associées aux paramètres les trois mécanismes de tolérance aux fautes présentés précédemment peuvent être instanciés de la manière suivante :

$$\text{PBR} = \left(\begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right)$$

$$\text{LFR} = \left(\begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \right)$$

$$\text{TR} = \left(\begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right)$$

Les mécanismes ne sont donc représentés que par les hypothèses nécessaires à leur association avec un composant fonctionnel et par les types de fautes qu'ils tolèrent. Les autres caractéristiques de ces mécanismes telles que la couverture du mécanisme de tolérance aux fautes seront abordés dans la partie mesures et analyse.

2.5.2 Adéquation et Compatibilité : formalisation avec relations d'ordre

La formalisation sous forme de vecteur des applications et des mécanismes de tolérance aux fautes permet de mettre sous la forme d'équations les propriétés d'adéquation et de compatibilité. En effet, si l'on reprend la modélisation générique avec :

$$A_i = \left(\begin{pmatrix} a_{i,1} \\ a_{i,2} \\ \dots \\ a_{i,m} \end{pmatrix}, \begin{pmatrix} f_{i,1} \\ f_{i,2} \\ \dots \\ f_{i,p} \end{pmatrix} \right) \text{ et } FT_j = \left(\begin{pmatrix} b_{j,1} \\ b_{j,2} \\ \dots \\ b_{j,m} \end{pmatrix}, \begin{pmatrix} ft_{j,1} \\ ft_{j,2} \\ \dots \\ ft_{j,p} \end{pmatrix} \right)$$

Avec pour rappel : *p* le nombre de type de fautes et *m* le nombre de caractéristiques applicatives.

On peut définir l'adéquation et la compatibilité comme suit:

Adéquation : Soit un composant A_i et un mécanisme de tolérance aux fautes FT_j , A_i et FT_j sont en **adéquation** si et seulement si pour tout k dans $[\![1 ; p]\!]$ $f_{i,k} \geq ft_{j,k}$.

Compatibilité : Soit un composant A_i et un mécanisme de tolérance aux fautes FT_j , A_i et FT_j sont **compatibles** si et seulement si pour tout k dans $[\![1 ; m]\!]$ $a_{i,k} \geq b_{j,k}$.

Le principal inconvénient de ces définitions est qu'elle ne permet pas d'utiliser des hypothèses qui seraient la résultante de combinaison de caractéristiques applicatives. Supposons que l'on souhaite distinguer l'existence d'un état interne à l'application de la capacité d'y accéder. En effet, rien ne sert d'avoir un accès à l'état du composant si celui-ci n'en a pas.

On peut donc introduire une nouvelle propriété appelé ST qui serait définit comme suit :

-ST=1 si le composant ne possède pas d'état, ST=0 sinon.

On s'assure ici que l'hypothèse forte est bien affectée à la valeur 1. En effet, une application avec un état est plus difficile à associée avec un mécanisme. Par exemple un mécanisme de redondance temporelle nécessite qu'en cas de présence d'un état il faut être capable de le récupérer et de remettre l'application dans l'état initial lorsque l'on effectue la réexécution de la requête initiale.

Dès lors les hypothèses des mécanismes PBR et TR sur l'accès à l'état d'un composant fonctionnel deviennent:

-TR et PBR ne sont compatibles avec l'application si celui-ci a un état et qu'il est accessible.

Cette formulation introduit une dépendance entre deux paramètres (SA et ST) du vecteur des caractéristiques applicatives qui est à présent de taille 4 (DT, ST, SA et FS). Or l'utilisation d'une relation d'ordre paramètre par paramètre ne nous permet pas de modéliser cette dépendance.

En conclusion cette vision vectorielle permet une définition généraliste des interactions entre mécanismes et composants fonctionnels et d'utiliser une modélisation similaire pour les deux. Le problème de la dépendance entre paramètres peut-être évité en s'assurant que le vecteur des caractéristiques applicative est minimal c'est-à-dire que toutes ces composantes sont indépendantes. Ainsi on ne regarde plus ST et SA séparément mais la propriété *not ST and not SA* (il y a un état et pas d'accès). Ce faisant nous revenons à un vecteur de caractéristiques applicatives de dimension trois dont toutes les composantes sont indépendantes au regard de la compatibilité.

Cette manipulation permet d'introduire une représentation sous forme d'expressions booléennes des propriétés de compatibilité et d'adéquation comme nous allons le voir dans la sous-section suivante.

2.5.3 Adéquation et Compatibilité : formalisation par expression booléennes

Il s'agit ici de formuler les expressions booléennes associées aux propriétés de compatibilité et d'adéquation pour chaque mécanisme. Cette formulation est plus naturelle que la précédente dans la mesure où elle est directement tirée de l'étude des mécanismes et peut se voir comme une transcription littérale de cette étude.

Prenons le cas du mécanisme PBR. Nous avons vu que ce mécanisme requiert l'accès à l'état s'il y en a un et qu'il nécessite également que l'application soit silencieuse sur défaillance. En ce qui concerne les fautes, PBR ne tolère ni les fautes par omission ni les fautes en valeur.

Nous pouvons donc écrire que PBR et une application sont cohérents si les expressions suivantes sont vraies :

- *not (not ST and not AC) and FS* est vraie pour le composant alors il y a compatibilité.
- *not (O or V)* est vraie pour le composant alors il y a adéquation.

La propriété de cohérence est donc la conjonction de ces deux expressions et ce pour tous les mécanismes de tolérance aux fautes:

-L'application et PBR sont cohérents ⇔ (not (not ST and not AC) and FS) and not (O or V)

-L'application et LFR sont cohérents ⇔ *(DT and FS) and not (O or V)*

-L'application et TR sont cohérents ⇔ (not (not ST and not AC) and DT) and (not C)

-L'application et LFR+TR sont cohérents ⇔ *(not (not ST and not AC) and FS and DT)*

Dans le cadre de l'utilisation d'expression booléenne on substitut la modélisation vectorielle des mécanismes de tolérance aux fautes par ces expressions. Un mécanisme est donc vu comme une assertion qui, une fois évaluée avec les valeurs obtenues lors de l'instanciation d'une application dans le modèle, indique s'il y a compatibilité et/ou adéquation.

Cette méthode présente l'avantage de pouvoir modéliser des dépendances entre paramètres. Toutefois lors de l'ajout d'un paramètre (modèle de fautes ou caractéristiques applicatives) il faut redéfinir ces expressions en incluant ces nouveaux paramètres alors que l'utilisation de la comparaison sur des vecteurs est plus souple quant à la modification du modèle comme nous le verrons quand nous parlerons des outils proposés dans cette thèse.

2.6 Conclusion

Nous avons donc présenté dans ce chapitre la méthode pour créer un modèle à partir du modèle de fautes et d'un ensemble de mécanismes de tolérance aux fautes. Les interactions entre mécanismes et application ont fait l'objet de définitions précises qui ont par la suite était décrite en fonction des paramètres du modèle soit en utilisant des expressions booléennes soit en utilisant des relations d'ordre.

Les deux modélisations des interactions ont donc leurs avantages et leurs inconvénients. Elles répondent à des besoins différents et seront toutes deux utiles pour la création d'un outil permettant d'analyser la résilience d'un système.

La modélisation vectorielle est plus rapide à mettre en place et à étendre puisque la définition d'un mécanisme dans un modèle M1 et facilement transposable à un model M2 pour peu que le second soit une extension du premier, c'est-à-dire que tous les paramètres de M1 soient inclus dans le modèle de M2. Ceci permet de facilement étendre les modèles sans nécessiter de refaire toute la modélisation des mécanismes.

La modélisation booléenne permet une représentation des mécanismes dans laquelle les propriétés de compatibilité et/ou d'adéquation s'expriment via des combinaisons et des dépendances entre les caractéristiques applicatives. Cette modélisation permet de supporter les dépendances entre caractéristiques applicatives.

Chapitre 3 –Analyse des mécanismes de tolérance aux fautes et de leur impact sur la résilience

“If you can’t measure it, you can’t improve it.”
Peter Drucker (1909-2005)

3.1 Introduction

Le chapitre précédent pose les bases d'une modélisation, qui une fois instanciée, nous permet de représenter l'ensemble des propriétés nécessaires à la mise en œuvre de mécanismes de sûreté de fonctionnement. Il est à présent temps d'utiliser ce modèle afin d'obtenir une première mesure de la résilience.

Dans ce chapitre nous verrons comment utiliser ce modèle afin d'identifier l'ensemble des profils d'applications qui peuvent apparaitre lors de la vie du système.

On rappelle qu'un profil d'application combine des caractéristiques structurelles (existence et accessibilité d'un état par exemple) et comportementales (déterminisme par exemple,) de l'application avec des spécifications non fonctionnelles, à savoir le modèle de fautes (par exemple, fautes matérielles générant des arrêts, des erreurs en valeur, etc.) que l'implémentation de cette application doit traiter.

Puis nous pourrons comparer ces profils avec l'ensemble des mécanismes de tolérance aux fautes envisagés. Toujours grâce aux propriétés de compatibilité et d'adéquation, il s'agira ensuite de qualifier la résilience globale du système en fonction de l'ensemble des mécanismes de tolérance aux fautes.

Un des aspects qui sera mis en avant dans cette étude est l'influence des hypothèses nécessaires à la mise en place des mécanismes. Nous verrons que considérer des variantes des mécanismes permet d'optimiser un premier critère de quantification de la résilience.

Nous verrons donc une première mesure de la résilience que nous appelons le Ratio de Cohérence (RC). L'objectif ici est de proposer une première approche quantitative de la résilience. Le RC sera défini de manière intuitive, puis formellement, afin de permettre une automatisation du calcul en utilisant le modèle précédemment décrit.

3.2 Première approche

Cette section est une présentation étape par étape de l'outil que nous présenterons par la suite. C'est une première approche détaillée qui décrit le procédé et le raisonnement qui nous conduira à quantifier la résilience d'un système.

3.2.1 Première analyse des profils

Supposons que l'on dispose d'un système qui n'est composé que d'une seule application. Conformément au chapitre 3 nous pouvons générer un modèle s'il n'en existe pas encore ou bien en utiliser un déjà disponible. Dans le cas où on utilise un modèle préexistant il faut s'assurer que tous les paramètres sur lesquels reposent les hypothèses nécessaires à la

mise en place de mécanisme de tolérance aux fautes sont présents ainsi que les différents types de fautes subies par l'application.

Si l'application et son mécanisme sont toujours compatibles et en adéquation et si un changement (des caractéristiques applicatives) survient durant la vie opérationnelle du système alors deux cas sont possibles : soit ce changement fait varier un des paramètres présents dans le modèle, soit il impacte d'autres paramètres. Dans le premier cas, il y a un risque que la sûreté de fonctionnement de l'application soit compromise. Le second cas est trivial, si aucun paramètre du modèle ne change alors cet événement n'a pas d'impact sur la sûreté de fonctionnement par définition du modèle.

Cette méthode permet de définir un espace fini (définit par l'ensemble des paramètres) dans lequel nous pourrons calculer la résilience du système. Cela correspond à faire l'hypothèse que la probabilité d'un événement (comme l'apparition d'un type de fautes qui n'est pas dans le modèle) est négligeable. Dans le cas où cette hypothèse viendrait tout de même à être invalidée (vieillissement matériel prématuré par exemple) le modèle devra être mis à jour comme nous le verrons dans le chapitre 5.

Nous pouvons donc représenter l'ensemble des profils possibles pour une application compte tenu du modèle dont nous disposons. Dans notre cas, le modèle est composé de 7 paramètres binaires. Parmi eux, 4 caractéristiques applicatives :

-DT : déterminisme
-ST : présence d'un état
-SA : accès à l'état
-FS : silence sur défaillance

Et 3 types de fautes :

-C : crash
-O : omission
-V : fautes transitoires

Une fois combinées nous obtenons 2^7 combinaisons possibles soit 128 possibilités de profil. Nous faisons le choix de ne pas traiter les cas triviaux où l'application n'est affectée par aucune faute (C=0, O=0, V=0) dans un but d'optimisation notamment lorsque nous automatiserons le processus. De plus, ces cas n'apportent pas d'information quant à la résilience du système. L'application a donc 112 profils possibles que nous allons étudier.

Afin de simplifier les notations nous utiliserons la notation « !C » comme un équivalent de « C=0 » et ce pour tous les paramètres du modèle dans la suite de cette thèse.

On peut représenter ces profils dans un tableau où chaque cellule correspond à un profil. Chaque colonne est une combinaison de types de fautes et chaque ligne une combinaison de caractéristiques applicatives.

AC \ FM	C !O !V	!C O !V	!C !O V	C O !V	C !O V	!C O V	C O V
!DT,!ST,!SA,!FS							
!DT,!ST,!SA,**FS**							
!DT,!ST,**SA**,!FS							
!DT,!ST,**SA,FS**							
!DT,**ST**,!SA,!FS							
!DT,**ST**,!SA,**FS**							
!DT,**ST,SA**,!FS							
!DT,**ST,SA,FS**							
DT,!ST,!SA,!FS							
DT,!ST,!SA,**FS**							
DT,!ST,**SA**,!FS							
DT,!ST,**SA,FS**							
DT,ST,!SA,!FS							
DT,ST,!SA,**FS**							
DT,ST,SA,!FS							
DT,ST,SA,FS							

Tableau 2 - Tableau de profils non rempli

Ce tableau ce lit comme suit : la cellule bleue correspond à un profil tel que l'application n'est pas déterministe (!DT), ne possède pas un état (ST), ne permet pas l'accès à cet état (!SA), et est silencieuse sur défaillance (FS). De plus cette application n'est pas sujette aux fautes par crash (!C) et aux fautes par omission (!O) par contre elle ne prend subi des fautes en valeurs (V).

On peut la représenter de la manière suivante en utilisant le formalisme introduit dans le chapitre 2 :

$$A = \left(\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right)$$

Il s'agit à présent de reprendre la définition des mécanismes de sûreté de fonctionnement donnés précédemment, Primary Back-Up Replication, Leader Follower Replication, Time redundancy et la composition LFR+TR. Pour chaque profil de vérifier on vérifier alors s'il existe un mécanisme *cohérent* c'est-à-dire *compatible* et *adéquat*. Si un tel mécanisme existe alors on l'inscrit dans la cellule correspondante au profil, sinon on la laisse vide.

Dans un premier temps nous utiliserons les définitions strictes des mécanismes de tolérance aux fautes présentées précédemment. Ainsi les mécanismes de types duplex PBR et LFR, qui tolèrent les fautes par crash, nécessitent que l'application soit silencieuse sur défaillance. De plus, PBR requiert un accès à l'état si celui-ci existe et LFR n'est compatible qu'avec des applications déterministes. Ces deux mécanismes permettent de tolérer les fautes par crash.

La redondance temporelle du mécanisme TR est quant à elle définie comme un mécanisme de répétition puis de comparaison et nécessite un comportement déterministe ainsi qu'un accès à l'état si celui-ci existe pour effectuer la seconde itération dans le même contexte que la première. Ce mécanisme permet de tolérer les fautes en valeur et les fautes par omission.

On obtient alors le tableau 3 de la page suivante :

AC \ FM	C !O !V	!C O !V	!C !O V	C O !V	C !O V	!C O V	C O V
!DT,!ST,!AC,!FS							
!DT,!ST,!AC,**FS**							
!DT,!ST,**AC**,!FS							
!DT,!ST,**AC,FS**	PBR						
!DT,**ST**,!AC,!FS							
!DT,**ST**,!AC,**FS**	PBR						
!DT,**ST,AC**,!FS							
!DT,**ST,AC,FS**	PBR						
DT,!ST,!AC,!FS							
DT,!ST,!AC,**FS**	LFR						
DT,!ST,**AC**,!FS		TR	TR			TR	
DT,!ST,**AC,FS**	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,ST,!AC,!FS		TR	TR			TR	
DT,ST,!AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,ST,AC,!FS		TR	TR			TR	
DT,ST,AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR

1*
2*
3*

Tableau 3 - Tableau de profils rempli avec {LFR, TR, PBR, LFR+TR}

1* : Cette case est vide, il n'y a donc aucun mécanisme cohérent avec le profil ((!DT, !ST, !AC, FS),(C, !O, !V)). En effet pour tolérer les fautes par crash uniquement (C, !O, !V) d'une application il y a deux solutions : le mécanisme LFR mais il nécessite une application déterministe ce qui n'est pas le cas (!DT) ou bien le mécanisme PBR qui nécessite d'avoir accès à l'état si celui-ci existe ce qui n'est pas le cas (!ST et !AC).

2* : Ce profil nécessite de tolérer les fautes par crash. N'étant pas déterministe (!DT), et donc incompatible avec LFR, on souhaite utiliser le mécanisme PBR. Dans ce cas l'application possède un état et il est accessible (!ST et AC), elle est donc cohérente avec le mécanisme PBR.

3* : Contrairement au cas précèdent nous n'avons pas accès à l'état de l'application (!AC) mais comme l'application ne dispose pas d'un état (ST) alors elle est compatible avec le mécanisme PBR. Comme pour les deux autres profils l'application exige de tolérer les fautes par crash et n'est pas compatible avec LFR puisque non déterministe.

On note également que plusieurs mécanismes peuvent être cohérents avec un même profil. Par exemple une application qui n'est sujette qu'au fautes par crash tout en étant déterministe, silencieuse sur défaillance, et sans état est cohérente à la fois avec PBR et LFR. Pour plus de clarté nous choisissons dans ce cas-là de ne noter que LFR comme solution. Comme nous le verrons par la suite l'objectif de cette représentation est de savoir s'il existe au moins une solution pour chaque profil.

La première observation que l'on peut faire est qu'il y a plus de profils (donc de cellules) sans mécanismes cohérents que de profils pour lesquels il existe une solution de tolérance aux fautes, avec l'ensemble des mécanismes que nous considérons dans cette première étape. Si l'on suppose qu'une application correspondante à ce modèle est tolérante aux fautes, puis qu'après un événement quelconque elle change de profil alors il a

statistiquement plus de cas dans lequel ce nouveau profil rend l'application vulnérable aux fautes.

Une première analyse chiffrée permet de mettre en avant qu'il n'existe un FTM cohérent que pour 30% des profils. Cela signifie que dans le cas d'une application dont les caractéristiques applicatives et le modèle de fautes sont aléatoires (en supposant que chaque profil a une probabilité identique d'apparition) alors nous avons seulement 30% de chances d'être dans un cas où notre ensemble de mécanismes de tolérance aux fautes permet de fournir une solution.

Une observation plus fine permet de se rendre compte que la moitié supérieure de ce tableau est moins remplie (5%) que la partie inférieure (55%). Dans la première moitié sont regroupés tous les profils non déterministes. Or dans notre ensemble de quatre mécanismes de tolérance aux fautes trois ne sont compatibles qu'avec des profils déterministes (LFR, TR, LFR+TR).

Nous pouvons tirer plusieurs conclusions de ces premières observations. Premièrement, avec 70% de profil sans solutions l'ensemble des mécanismes de tolérance aux fautes dont nous disposons est insuffisant pour garantir la sûreté de fonctionnement. Deuxièmement, avec cet ensemble, passer d'un profil déterministe cohérent à un autre profil déterministe plutôt qu'à un profil non-déterministe augmente les chances de trouver un mécanisme cohérent et donc d'être résilient. Enfin, si l'on veut augmenter la résilience de l'application à l'étude il sera nécessaire de trouver des mécanismes compatibles avec des applications non-déterministes.

Cette mesure du pourcentage de profils pour lesquels il existe un mécanisme de tolérance aux fautes cohérent est appelée *Ratio de Cohérence*. La sous-section suivante aura pour objectif de définir avec plus de précision comment on la définit et ce qu'elle nous permettra de mesurer

3.2.2 Introduction au Ratio de Cohérence

Le Ratio de Cohérence est donc un premier estimateur de la résilience. Il répond à la question suivante : *Etant donnée une application A et un ensemble de mécanismes de tolérance aux fautes, quelle est la probabilité que A soit tolérante aux fautes grâce à un mécanisme dont nous disposons ?*

Comme nous l'avons vu précédemment, nous considérons ici des applications qui évoluent dans le temps, ces évolutions se traduisent par un changement de profil et peuvent être représentées dans le tableau par le passage d'une cellule du tableau à une autre.

On note pour tout profil *j*, α_j tel que α_j=1 si et seulement si il existe au moins un mécanisme de tolérance aux fautes cohérent avec le profil j, 0 sinon. On peut alors définir le Ratio de Cohérence comme suit :

$$RC=\frac{\sum_j \alpha j}{N}$$

Avec *N* le nombre de profils considérés (112 dans notre exemple).

Comme nous l'avons dit en introduction de ce travail, la résilience est notre capacité à garantir les propriétés de sûreté de fonctionnement lors des changements. C'est pourquoi le Ratio de Cohérence fournit un premier indicateur qui nous permettra par exemple d'orienter les efforts d'un éventuel développeur pour augmenter la résilience d'un système. C'est d'ailleurs ce processus que nous chercherons à illustrer dans la section suivante.

3.3 Analyse des profils avec extension des mécanismes

L'objectif à présent est donc de maximiser la valeur du ratio de cohérence pour le premier ensemble de mécanismes (PBR, LFR et TR). En effet, plus le nombre de profil pour lesquels on a une solution est important, plus la probabilité de garantir les propriétés de sûreté de fonctionnement suite à un changement est élevée.

Nous verrons ici deux approches d'extension des mécanismes de tolérance aux fautes: la première consiste à ajouter un nouveau mécanisme de tolérance aux fautes ; la seconde consiste à redéfinir les mécanismes pour créer des versions plus adaptées à l'application.

3.3.1 Ajout d'une stratégie TR0

Une première idée est donc d'ajouter de nouveaux mécanismes. Bien entendu, développer une nouvelle solution est couteux. Il faut donc trouver une solution pour optimiser la recherche de nouveaux mécanismes de tolérance aux fautes.

Comme nous l'avons vu dans la section 4.1. L'ensemble des mécanismes dont nous disposons est particulièrement inefficace lorsque l'application s'avère non-déterministe. En effet dans le cas d'une application non-déterministe il n'existe aucune solution permettant de tolérer les fautes par omissions et les fautes transitoires en s'appuyant sur la comparaison des sorties.

Dans le cas déterministe ces deux types de fautes sont tolérés par le même mécanisme de redondance temporelle TR. Ce mécanisme fonctionne en deux parties. Premièrement on exécute deux fois l'application, puis on compare les résultats pour vérifier qu'ils sont identiques. Les deux exécutions doivent s'effectuer dans le même contexte afin de produire les mêmes résultats. C'est cette comparaison qui impose le déterminisme de l'application.

Cependant, on peut envisager une version réduite de ce mécanisme. En effet, si l'on supprime la partie comparaison nous obtenons un mécanisme qui n'est plus capable de détecter des fautes en valeurs transitoires mais qui est capable de compenser une omission en traitant deux fois une même requête. On appelle TR0 ce mécanisme.

Dans la mesure où TR0 n'effectue pas de comparaison entre les résultats, il n'est pas nécessaire que l'application soit déterministe comme vu précédemment. L'objectif est ici de produire seulement un résultat par répétition. Nous avons donc deux nouveaux mécanismes :

-TR0 tolère uniquement les fautes par omission (O et !C et !V) et requiert l'accès à un état si il existe (!(ST et !AC)).

Par ailleurs ce nouveau mécanisme peut être couplé au mécanisme duplex PBR. En effet le principe de composition est identique à celui du mécanisme LFR+TR. Dans le cas de PBR+TR0 les deux mécanismes ont pour avantage d'être compatibles avec des applications non déterministes. Or on observe un manque de mécanismes pour les applications dont le profil est non-déterministe, c'est pourquoi il est intéressant de faire l'effort de développer PBR+TR0. On obtient donc le mécanisme suivant :

-PBR+TR0 tolère les fautes par omission et par crash ((O ou C) et !V) et requiert l'accès à un état existant ainsi que le silence sur défaillance (!(ST et !AC) et FS).

Il est important de noter ici que les fautes par omission et par crash seront tolérées uniquement pour des applications à silence sur défaillance, ce qui signifie que toute valeur en sortie est correcte. Dans le cas d'une application qui ne serait pas silencieuse sur défaillance, alors des fautes en valeur pourraient survenir, puisque PBR+TR0 ne les tolère pas.

Comme précédemment, dans certains cas, plus d'un mécanisme peut être solution. Par exemple chaque fois que l'on utilise TR pour tolérer les fautes par omission uniquement il est possible de le remplacer par TR0. Cependant, pour une meilleure lisibilité, nous conserverons dans le prochain tableau les solutions trouvées aux étapes précédentes.

On peut donc reprendre notre tableau de l'ensemble des profils afin de le compléter de la manière suivante :

AC \ FM	C !O !V	!C O !V	!C !O V	C O !V	C !O V	!C O V	C O V
!DT,!ST,!AC,!FS							
!DT,!ST,!AC,**FS**							
!DT,!ST,**AC**,!FS		TR0					
!DT,!ST,**AC,FS**	PBR	TR0		PBR+ TR0			
!DT,**ST**,!AC,!FS		TR0					
!DT,**ST**,!AC,**FS**	PBR	TR0		PBR+ TR0			
!DT,**ST,AC**,!FS		TR0					
!DT,**ST,AC,FS**	PBR	TR0		PBR+ TR0			
DT,!ST,!AC,!FS							
DT,!ST,!AC,**FS**	LFR						
DT,!ST,**AC**,!FS		TR	TR			TR	
DT,!ST,**AC,FS**	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,ST,!AC,!FS		TR	TR			TR	
DT,ST,!AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,ST,AC,!FS		TR	TR			TR	
DT,ST,AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR

Tableau 4 - Tableau de profils rempli avec {LFR, TR, PBR, LFR+TR, TR0, PBR+TR0}

Le Ratio de Cohérence pour l'ensemble de mécanismes (LFR, PBR, TR, TR0, LFR+TR et PBR+TR0) est de 38%. L'amélioration est donc significative (+8%) d'un point de vue global.

Si l'on s'intéresse aux profils non déterministes nous passons de 5% à 21% avec TR0. Dans la première version de l'ensemble des mécanismes, nous avions vu que nous étions particulièrement peu résilient dans le cas d'une application non-déterministe. Cette observation a conduit à la mise en place d'un nouveau mécanisme de tolérance aux fautes qui accepte des profils non déterministes.

La mesure du RC permet donc d'identifier les paramètres pour lesquels l'ensemble de mécanismes de tolérance aux fautes n'est pas compatible. Une fois ces paramètres identifiés nous pouvons ajouter de nouveaux mécanismes qui sont compatibles avec ces caractéristiques.

En automatisant la mesure du RC nous pouvons donc envisager d'étudier la sensibilité de la mesure aux différents paramètres du modèle. Ce faisant, nous pourrons identifier les points sur lesquels travailler pour augmenter la résilience du système en ajoutant d'autres mécanismes ou bien en reconsidérant les hypothèses comme nous le verrons dans la sous-section suivante.

3.3.2 Redéfinition des FTMs

Il s'agit à présent de travailler sur l'application dont on souhaite assurer le fonctionnement nominal en présence de fautes ainsi que sur les mécanismes de tolérances

aux fautes. C'est en se concentrant sur les hypothèses que nous pourrons une fois de plus augmenter le ratio de cohérence et donc la résilience du système.

Jusqu'à présent nous avions fait l'hypothèse que les stratégies duplex ne peuvent pas être compatibles avec des applications qui ne sont pas silencieuse sur défaillance. C'est une définition très stricte des mécanismes duplex. Cependant, la notion d'application silencieuse sur défaillance est elle-même probabiliste et dépend de la couverture du mécanisme de détection d'erreur interne. Cette couverture bien que très élevée ne pourra jamais atteindre les 100%. Nous travaillons donc avec une estimation de cette couverture et avec un seuil à partir duquel le développeur déclare que l'application est bel et bien silencieuse sur défaillance, et ce basé sur les mesures qu'il a lui-même effectuées. Ces mesures peuvent être issues de campagnes intensives d'injection de fautes.

Notons que dans le cas où l'application ne serait pas silencieuse sur défaillance, les fautes par crash pourraient être tolérées par les mécanismes duplex tout en sachant que certaines erreurs en valeur pourraient être observées. C'est donc au développeur de choisir entre ne jamais se prémunir des fautes par crash pour les applications non silencieuses sur défaillance et en tolérer certaines en acceptant une probabilité faible d'erreur en valeur qui est intimement liée à la couverture des mécanismes de détection internes au programme de l'application.

Comme nous venons de le dire la propriété de silence sur défaillance est une mesure de la couverture d'un mécanisme de détection. Cependant lorsque l'on utilise un mécanisme de redondance temporelle tel que TR on augmente cette capacité à détecter les sorties erronées dues à des défaillances. On peut donc coupler les mécanismes TR ou TR0 avec une redondance froide de type PBR ou chaude de type LFR pour tolérer les fautes par crash même si l'application n'est pas définie comme silencieuse sur défaillance.

On peut donc de nouveau remplir le tableau avec de nouveaux mécanismes (ici en bleus).

AC \ FM	C !O !V	!C O !V	!C !O V	C O !V	C !O V	!C O V	C O V
!DT,!ST,!AC,!FS							
!DT,!ST,!AC,**FS**							
!DT,!ST,**AC**,!FS	PBR	TR0		PBR+ TR0			
!DT,!ST,**AC,FS**	PBR	TR0		PBR+ TR0			
!DT,**ST**,!AC,!FS	PBR	TR0		PBR+ TR0			
!DT,**ST**,!AC,**FS**	PBR	TR0		PBR+ TR0			
!DT,**ST,AC**,!FS	PBR	TR0		PBR+ TR0			
!DT,**ST,AC,FS**	PBR	TR0		PBR+ TR0			
DT,!ST,!AC,!FS	LFR						
DT,!ST,!AC,**FS**	LFR						
DT,!ST,**AC**,!FS	LFR	TR	TR	LFR+ TR0	LFR+ TR0	TR	LFR+ TR0
DT,!ST,**AC,FS**	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,**ST**,!AC,!FS	LFR	TR	TR	LFR+ TR0	LFR+ TR0	TR	LFR+ TR0
DT,**ST**,!AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,**ST,AC**,!FS	LFR	TR	TR	LFR+ TR0	LFR+ TR0	TR	LFR+ TR0
DT,**ST,AC**,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR

Tableau 5 - Tableau de profils rempli avec {LFR, TR, PBR, LFR+TR, TR0, PBR+TR0, LFR+TR0} et leurs variantes

Si l'on mesure à nouveau le ratio de cohérence nous sommes à présent à 55% de profils ayant au moins un mécanisme cohérent. Dans le cas d'application déterministe, ce pourcentage s'élève à 89% tandis qu'il est de 32% pour les profils non-déterministes.

En conclusion nous avons montré qu'avec de légères variations des mécanismes déjà disponibles ainsi qu'une révision partielle des hypothèses de compatibilité (en faisant un compromis du point de vue de la sûreté de fonctionnement) nous avons considérablement augmenté le RC. Toutefois, le RC n'est pas de 100% et ce pour deux raisons. Premièrement, les fautes en valeurs ne peuvent être tolérées dans le cas d'applications non déterministes avec nos mécanismes. Deuxièmement, les applications ayant un état auquel on ne peut pas accéder posent problème.

Dans le premier cas, il nous faudrait être capable de lever le non-déterminisme. C'est-à-dire identifier et capturer les points de non déterminisme lors de l'implémentation afin de synchroniser les deux copies. Ce qui pourrait entrainer le développement de nouvelles solutions comme des variantes du mécanisme LFR. C'est une approche possible qui a déjà été expérimentée dans le cadre du projet Européen DELTA-4 [28]. Le processus de développement doit être très rigoureux pour éviter par construction les sources de non-déterminisme (multithreading, valeurs aléatoires, etc.) et d'être en mesure d'identifier les points de non-déterminisme résiduels dans le composant final. Des campagnes de test adéquates sont aussi nécessaires pour valider cette propriété [18].

La problématique de l'accès à l'état quant à elle semble bien plus difficile à résoudre. Pour ce faire il est nécessaire d'avoir accès à un niveau très bas avec par exemple des solutions logicielles comme des hyperviseurs ou encore des plateformes matérielles spécifiques. Ces solutions permettraient de récupérer un journal d'exécution de l'application qui permettrait de reconstituer son état. La journalisation est en effet une technique bien connue. Elle peut être faite à différents niveaux d'abstraction dans le système y compris au niveau le plus haut, c'est-à-dire en capturant et en historisant toutes les requêtes qui sont transmises au composant d'application cible. Le rejeu des requêtes sur un composant à partir de son état initial permet de reconstruire l'état courant. Cette approche est cependant relativement lourde à mettre en œuvre sur le plan de la sauvegarde des requêtes et peut avoir un impact significatif sur le plan temporel.

3..4 Automatisation des mesures et analyse de sensibilité

3.4.1 L'outil

Comme nous l'avons vu dans la partie précédente le calcul du Ratio de Cohérence repose sur les paramètres du modèle pour définir l'ensemble des profils possibles et sur l'ensemble des mécanismes de tolérance aux fautes. Dans cette section nous présentons un outil nous permettant de calculer le ratio de cohérence.

3.4.2 Introduction à l'utilisation

Cet outil prend deux jeux de données en entrée :

-Les paramètres du modèle (caractéristiques applicatives et types de fautes)

-Un ensemble de mécanismes de tolérances aux fautes.

En sortie nous obtenons le ratio de cohérence.

Cet outil utilise la formulation booléenne des mécanismes de tolérance aux fautes telle que vue au chapitre précédent. Ainsi le mécanisme PBR se modélise via l'assertion suivante :

L'application et PBR sont cohérents ⇔ (not (not ST and not AC) and FS) and not (O or V).

On effectue un parcours exhaustif des profils possibles de l'application (AC et FM) et pour chaque profil on cherche les mécanismes cohérents. Chaque mécanisme cohérent est alors stocké dans la cellule correspondante au profil en cours d'analyse. Lorsque l'on a fini de parcourir tous les profils on calcule le RC comme le rapport entre le nombre de cellule pleine et le nombre de cellule total.

L'algorithme qui utilise ces assertions est donné en pseudo-code. Il a été vérifié en comparant les résultats obtenus avec les oracles calculés à partir des tables données dans les sections précédentes :

```
INPUT: Application model, FTMs set

FOR each application characteristics AC
      FOR each fault model FM
            FOR each FTM
                  IF FTM ◊ (AC,FM) is consistent THEN
                        Store FTM in the cell (AC,FM)
                  END IF
            END FOR
            IF the cell (AC,FM) is not empty
                  Increment NbrOfConsCells
            END IF
      END FOR
END FOR

RETURN: CR=NbrOfConsCells/TotNbrOfCells
```

Une présentation détaillée du code de l'outil et de l'interface est disponible en annexe. Cet outil permet d'explorer les propriétés de cohérence d'une application de manière systématique et exhaustive grâce au niveau d'abstraction du modèle simplifiant les calculs. On l'utilisera afin de conduire des analyses de sensibilité comme nous le verrons dans la section suivante.

3.4.3 Sensibilité aux caractéristiques applicatives

L'objectif de cette analyse est d'identifier les caractéristiques applicatives qui ont le plus d'impact sur le RC. Pour chaque caractéristique applicative on mesure donc le RC pour toutes les valeurs possibles de la caractéristique. Pour ce faire on utilise l'algorithme précèdent en le modifiant afin de limiter le parcours des caractéristiques applicatives et/ou du modèle de fautes en fonction de contraintes telle que : « *l'application est toujours déterministe* » ou « *l'application ne commet jamais de fautes en valeurs* ».

On obtient le diagramme de la figure 13, dans lequel sur l'axe des abscisses on retrouve la caractéristique qui a été fixée à une valeur pour effectuer la mesure. Pour simplifier la lecture on note DT=1 la colonne correspondante à l'affirmation "*l'application sera toujours déterministe*" et DT=0 celle où "*l'application ne sera jamais déterministe*". Cela signifie que dans le cas où l'on assure que l'application sera toujours déterministe le processus de développement doit garantir ce déterminisme même si cela nécessite un effort supplémentaire. En revanche, lorsque l'on garantit que l'application ne sera jamais déterministe cela signifie que du fait même de la nature de l'application (algorithme d'optimisation par exemple) il ne sera jamais possible d'obtenir une application déterministe.

Le RC de référence est la valeur obtenue lors d'un calcul sans contraintes sur les valeurs des paramètres.

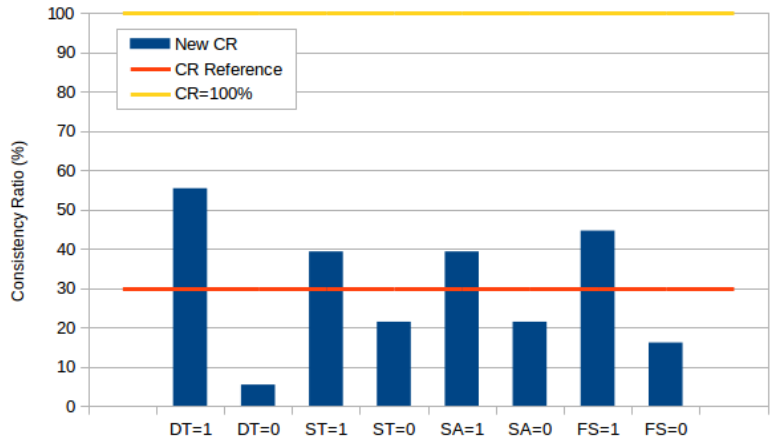


Figure 13- Analyse de sensibilité aux caractéristiques applicatives avec {PBR, LFR, TR, LFR+TR}

Ces mesures ont été effectuées avec l'ensemble de mécanismes de tolérance aux fautes composé de PBR, LFR, TR et LFR+TR. Cet ensemble est donc le même que celui présenté dans le tableau 1. Le CR de référence est identique à celui précédemment mesuré i.e 30%. L'analyse de sensibilité doit nous permettre de répondre à la question suivante : Quelle est l'impact sur la valeur du RC si nous décidons de fixer la valeur d'un des paramètres ?

On remarque que quelle que soit la caractéristique applicative, si l'on fixe sa valeur à 3 alors on obtient un RC supérieur à celui de référence. Par exemple, si l'application est toujours déterministe (DT=1) alors le RC passe de 30% à 55%. Réciproquement lorsque l'application est supposée jamais déterministe le RC est de 5%.

Cette augmentation systématique quand un paramètre vaut 1 s'explique par les choix de formulation des hypothèses faits pendant la modélisation comme nous l'avons expliqué au chapitre 2. Ainsi, une valeur de 0 correspond à une hypothèse plus faible en ce qui concerne l'application (ex : application non silencieuse sur défaillance). Or les mécanismes de tolérance aux fautes requièrent des hypothèses fortes afin d'être compatibles.

Un autre résultat de cette analyse est que le déterminisme est la caractéristique qui impacte le plus le ratio de cohérence. Cela signifie que si l'on veut améliorer la résilience du système il nous faudra trouver plus de moyens pour tolérer les fautes dans le cas de composants non-déterministes et/ou s'assurer que le composant ne le devienne jamais lors de ses évolutions successives. L'analyse de sensibilité permet donc d'exhiber les axes d'amélioration les plus significatifs.

3.4.4 Sensibilité au modèle de fautes

Dans cette section on considère l'ensemble des mécanismes de tolérance aux fautes le plus complet dont nous disposons, i.e. celui du tableau 4, afin de pouvoir illustrer les deux

conséquences possibles à la fixation d'une valeur du modèle de fautes. Dans ce cas le ratio de cohérence est de 55% pour l'ensemble des profils. De manière analogue à l'analyse de la sous-section précédente nous pouvons fixer la valeur de chaque type de fautes et calculer la valeur du nouveau RC.

La figure 14 regroupe les résultats obtenus lors de cette analyse. Lorsqu'un type de faute est retiré (non considéré) on lui affecte la valeur 0, et on lui affecte la valeur 1 si ces fautes sont actives et doivent être prises en compte. Le but est de trouver les types de fautes qui ont le plus d'impact sur le CR.

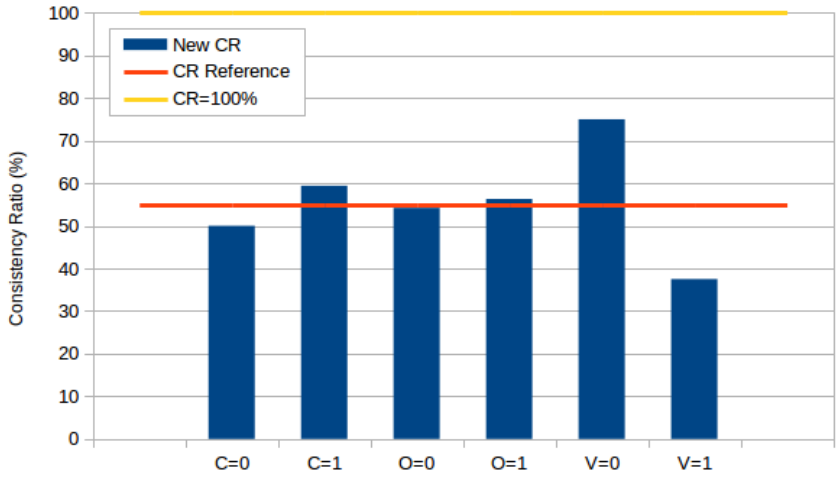


Figure 14 - Analyse de sensibilité aux types de fautes

Contrairement à l'analyse de sensibilité sur les caractéristiques applicatives, retirer un type de fautes n'implique pas nécessairement une augmentation du ratio. Comparons les fautes par crash et les fautes en valeurs :

-Retirer les fautes par crash (i.e. fixer C=0) diminue le RC. Ceci s'explique par la bonne capacité du système à tolérer les fautes par crash. Si le RC est affecté négativement lorsque l'on supprime ces fautes alors on peut conclure que la tolérance à ce type de fautes est plus élevée que la moyenne.

-Réciproquement, lorsque l'on retire uniquement les fautes en valeurs (i.e. fixer V=0) le ratio de cohérence augmente significativement. On peut donc en conclure que compte tenu des mécanismes à disposition le système sera moins résilient si des fautes en valeurs sont actives puisque nous ne sommes pas capables de tolérer les fautes en valeurs pour une grande partie de combinaison de caractéristiques applicatives.

En conclusion, cette analyse nous a permis d'identifier les points forts et les points faibles de l'ensemble des mécanismes de tolérance aux fautes {PBR, LFR, TR, LFR+TR, TR0, PBR+TR0, LFR+TR0} au regard des types de fautes à tolérer. Pour améliorer le RC nous avons deux possibilités analogues à l'analyse de sensibilité sur les caractéristiques applicatives : a) Effectuer un travail en amont pour éliminer ces types de fautes; b) définir et développer de nouveaux mécanismes.

3.4.4 Sensibilité à l'ensemble des mécanismes de tolérance aux fautes.

Comme nous l'avons précisé à chaque mesure, le ratio de cohérence est calculé à partir d'un ensemble de mécanismes de tolérance aux fautes. Or plus l'ensemble est conséquent, plus il y a de chance qu'un même profil soit cohérent avec plusieurs mécanismes. L'objectif de cette partie est donc d'optimiser le nombre de mécanismes à implémenter grâce à l'étude de leur modélisation théorique.

Prenons l'ensemble des mécanismes que nous avons considérés jusqu'à présent {PBR, LFR, TR, TR, LFR+TR, PBR+TR} pour lequel le RC est de 38%. Calculons maintenant la valeur du RC pour chaque sous-ensemble de mécanisme de tolérance aux fautes. Les résultats sont présentés sur la figure 15.

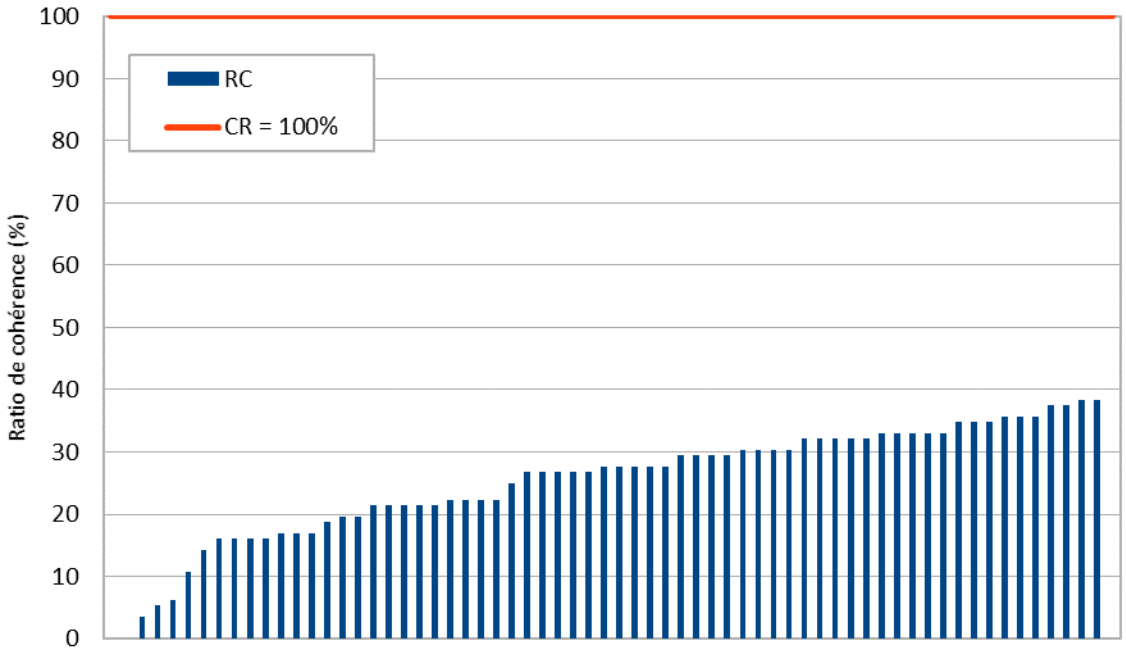


Figure 15- Analyse de sensibilité à l'ensemble des mécanismes

Nous avons arbitrairement choisi de classer les résultats par ordre croissant afin de faciliter la lecture. La valeur maximale de 38 % est atteinte pour l'ensemble complet de mécanismes. En rouge nous avons par exemple une valeur de CR=27% pour le sous-ensemble {LFR, LFR+TR, TR0}. Que peut-on conclure de ce type d'étude à ce stade ? Nous constatons que certains sous-ensembles, bien que incomplets par principe, permettent d'obtenir un ratio de cohérence qui peut être acceptable pour une application avec un coût de développement moindre des mécanismes de tolérance aux fautes. Dit autrement, le coût de développement d'un ensemble plus large de mécanismes qui peut être important n'améliore pas parfois de façon significative la valeur du RC.

Ce type d'analyse peut également être couplé à des analyses de sensibilité que nous avons précédemment citées. Par exemple on peut comparer les valeurs du RC lorsque

l'application est déterministe. Sur le diagramme figure 16 on fait figurer en orange l'impact sur le RC du choix d'une application déterministe. On s'aperçoit par conséquent que l'effort sur le développement des applications au niveau de leurs caractéristiques peut limiter le nombre de mécanismes de tolérance aux fautes pour atteindre un niveau de résilience satisfaisant.

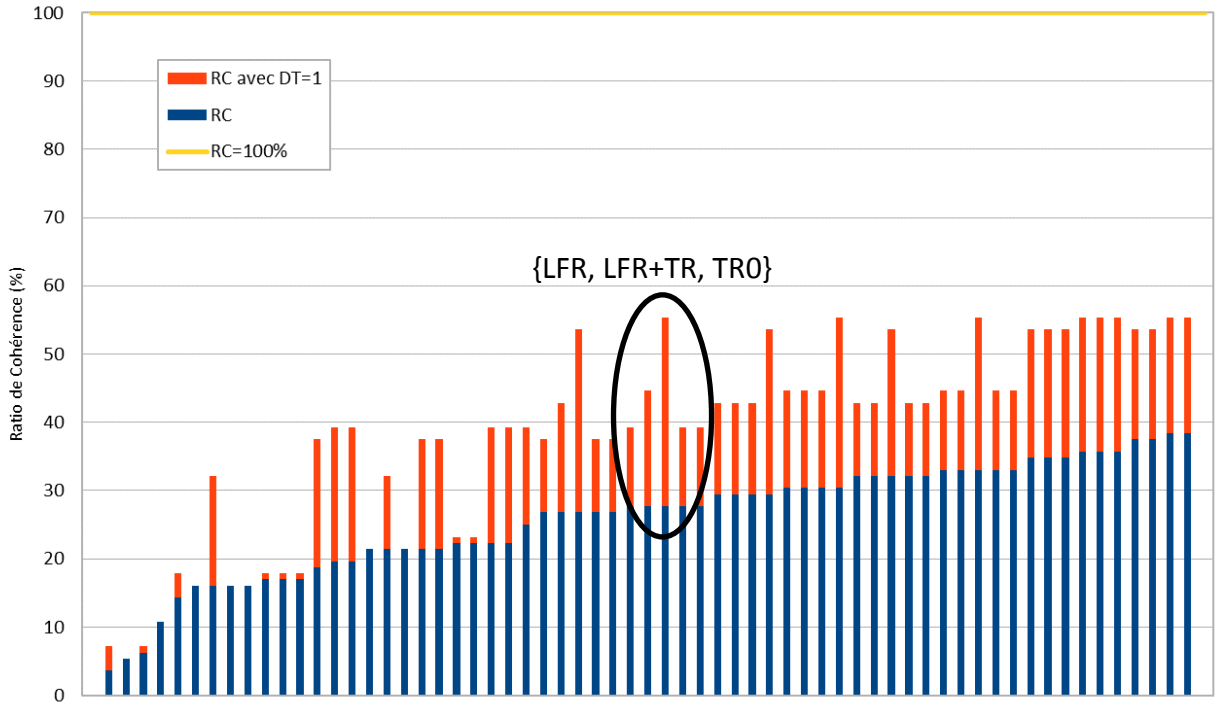


Figure 16 - Analyse de sensibilité à l'ensemble des mécanismes couplée à la sensibilité au déterminisme

On remarque la valeur maximale est obtenue avec un ensemble complet de mécanismes (CR=55%). Toutefois, l'ensemble de mécanisme {LFR, LFR+TR, TR0} est suffisant pour obtenir un ratio de cohérence de 53,5% dans le cas où l'application est toujours déterministe.

En conclusion, combiner une analyse des sous-ensembles de mécanismes avec une analyse de sensibilité permet d'optimiser les efforts de développement. Ainsi un développeur pourra choisir de forcer son application a être déterministe dans le but d'améliorer la résilience du système. Une conséquence de ce choix sera que trois mécanismes suffisent pour obtenir un ratio de cohérence qui se rapproche du maximum que l'on peut atteindre avec les 6 mécanismes envisagés.

Bien entendu, de pareil choix (notamment forcer le déterminisme de l'application) implique des contraintes fortes lors du développement. On notera également que le RC n'est qu'un estimateur de la résilience et qu'il doit être mis en parallèle avec d'autre critère. En effet, si d'un point de vue du coût il est intéressant de diminuer le nombre de mécanismes à implémenter, avoir une diversité de mécanismes doit nous permettre d'avoir des solutions plus souples du point de vue de la consommation de ressources notamment. Or la disponibilité de ressources telles que le CPU, la bande-passante, etc., sont des critères importants lorsque l'on mesure la résilience du système.

3.5 Définition du Ratio de Cohérence

Dans la sous-section précédente nous avons introduit de manière intuitive la notion de Ratio de Couverture RC. Dans cette partie il s'agira de présenter cette mesure de manière plus complète et plus mathématique.

Tout d'abord il nous faut énoncer explicitement certaines hypothèses. En effet, il était implicite dans la partie précédente que, sachant que l'application a un profil P_0 à l'instant t_0, la probabilité pour qu'elle ait un profil P_1 à l'instant t_1 est identique à celle d'avoir un profil P_1' quelconque à ce même instant. Cela signifie que tous les profils sont équiprobables dans la vie du système, ou tout au moins pour une série d'évolutions successives d'un composant du système.

Dans un le cas où tous les profils ont la même probabilité d'être à un instant quelconque la définition donnée précédemment est parfaitement valide :

Le **Ratio de Cohérence** est le pourcentage de profils tels qu'il existe au moins un mécanisme de tolérance aux fautes cohérent avec ce profil.

RC est donc valable pour un ensemble de mécanismes donnés et un ensemble de profils donnés (modèle) et selon une certaine distribution de probabilité. Il nous permet de se faire une première idée de notre capacité à garantir les propriétés de sûreté de fonctionnement lorsqu'une application est soumise à des changements qui peuvent affecter son profil.

Cette situation d'équiprobabilité n'est pas une hypothèse que l'on peut faire dans la plupart des cas réels. En fait la probabilité d'avoir un profil P_1 ou P_1' à l'instant t_1 depuis un profil P_0 est différente. Elle dépend d'ailleurs du profil à l'instant précédent le changement. En effet, si l'on considère une application pour laquelle les développeurs ont fait des efforts pour qu'elle soit déterministe alors il y a peu de chance qu'une mise à jour provenant de ces mêmes développeurs la rende subitement non-déterministe. Mais, qui sait ?

Dans le but de relâcher l'hypothèse d'équiprobabilité on peut ainsi définir la matrice des probabilités $MP=(p_{ij})_{1\leq i,j\leq 112}$ dans laquelle p_{ij} est la probabilité de passage du profil P_i au profil P_j. Nous considérons dans cette approche que ces probabilités n'évoluent pas avec le temps et que seul le profil en cours a eu une influence sur le profil suivant. Cela permet d'obtenir une valeur du RC qui ne dépend pas du temps auquel on effectue le calcul.

Nous devons à présent expliciter la matrice MP précédemment définie. Pour ce faire il nous faut revenir à la définition d'un profil. Supposons que notre modèle ne dispose que d'une seule caractéristique : une application est déterministe (DT) où elle ne l'est pas (!DT) et un seul type de fautes : les fautes par crash (C ou !C). Une application dans ce modèle ne peut avoir que quatre profils :

-DT et C

-DT et !C

-!DT et C

-!DT et !C

Pour l'exemple et contrairement à ce qui a était fait précédemment nous ne supprimerons pas les cas triviaux où aucune faute n'est présente. On peut pour chacun de ces paramètres définir une chaine de Markov qui représente la probabilité pour passer d'une valeur à une autre :

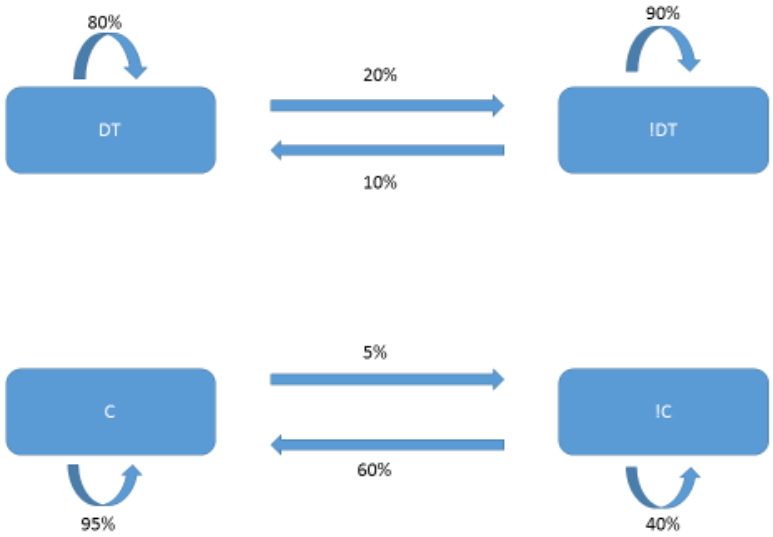


Figure 17 - Probabilité de changement des caractéristiques applicatives

La modélisation sous forme de chaîne de Markov permet de prendre en compte les spécificités des applications. En effet on s'attache ici à exprimer la probabilité d'être dans un profil après un changement sachant que l'application est dans un certain profil initialement. Les chaînes de Markov permettent donc de corréler les changements et les profils de l'application étudiée.

Les valeurs proposées dans cet exemple sont arbitraires afin d'illustrer la méthode utilisée. Pour obtenir des valeurs cohérentes avec la réalité il faut réaliser une étude préliminaire qui donnerait une estimation de ces probabilités. Ces estimations peuvent se baser sur des données d'application précédemment mises en production et dont l'étude du cycle de vie a été faite (analyse des journaux de mises à jour par exemple). Par ailleurs, on peut dans certains cas fixer la valeur de certains paramètres. Par exemple, si l'on sait qu'une application est à coup sûr non-déterministe (générateurs pseudo-aléatoires, etc…) on peut fixer sa probabilité d'être déterministe à 0%.

Une fois ce travail effectué pour chaque paramètre, on peut obtenir une chaine de Markov qui représente l'ensemble des profils et les probabilités pour passer de l'un à l'autre. Par exemple, sachant que l'on est dans le profil (DT, C) la probabilité de rester dans ce profil est égale à la probabilité de rester déterministe (80%) multipliée par la probabilité de garder les fautes par crash (90%) soit 76%. Lorsque l'on est dans le profil (DT, C) on a donc 76% de

chance de le rester. Pour passer de (DT, C) à (!DT, C) on multiplie la probabilité de devenir non déterministe (20%) par la probabilité de garder les fautes par crash (90%) soit 19%. Un raisonnement similaire est effectué pour tous les profils et toutes les transitions possibles.

Sur les arcs se trouvent les valeurs de la matrice MP que nous cherchions à déterminer. L'exemple précédent avec 4 profils nous permet d'obtenir la chaine suivante :

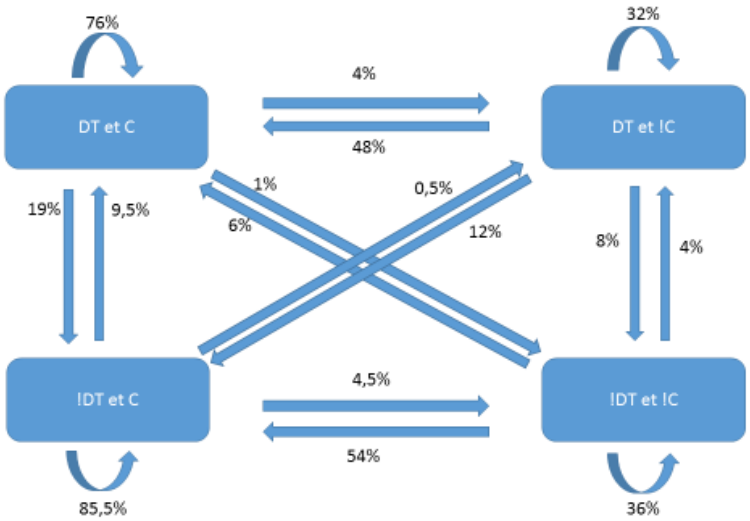


Figure 18 - Chaine de Markov d'évolution des profils

Comme les probabilités d'avoir un profil à un instant donné dépendent du profil de l'instant précédent, on ne peut plus calculer un ratio de cohérence global à l'application. Il nous faut définir un ratio de cohérence qui dépendra du profil dans lequel on se trouve.

Considérons que nous ne disposons que du mécanisme LFR pour protéger notre application dont les probabilités de changement sont représentées par la figure 18. Les profils (DT, C) et (!DT, C) sont les deux cas où l'application nécessite de tolérer des fautes. Or LFR requiert une application déterministe et donc seul (DT,C) peut être protéger. Les deux autre profils (DT, !C) et (!DT, C) sont cohérents puisque triviaux (pas de fautes possibles).

Si l'application a un profil (DT,C) alors elle a 19% de chance de devenir (!DT,C) au prochain changement comme indiqué sur la figure 18. Il y a donc 19% de chance de devenir incohérente et 81% de chances de rester cohérente. On a donc un Ratio de Cohérence de 81% pour le profil (DT,C).

Si on est dans le profil (!DT,C) alors l'application à 85.5% de chance rester incohérente au prochain changement et donc 14.5% de chance de devenir cohérente. Le Ratio de Cohérence pour ce profil est donc de 14.5%.

Ce raisonnement peut être effectué pour chaque profil. On généralise cette approche en la formalisant mathématiquement comme suit.

Soit E_{ftm} un ensemble de mécanismes de tolérance aux fautes. A chaque profil P_j on associe un booléen α_j qui vaut 1 si il existe un mécanisme de tolérance aux fautes qui soit cohérent avec ce profil et 0 sinon. Soit $MP=(P_{ij})_{1\leq i,j\leq k}$ la matrice des probabilités de passage d'un profil à un autre telle que définie précédemment.

On définit pour tout profil P_i :

$$RC_i=\sum_j Pij * \alpha j$$

On notera que cette définition est compatible avec celle donnée précédemment dans le cas équiprobable avec 7 paramètres et donc 112 profils possibles une fois retiré les cas triviaux. Dans ce cas-là tous les RC_i sont égaux à RC car tous les P_{ij} sont égaux à l'inverse du nombre de profils possibles, soit pour tout i et j P_{ij} = 1/112.

On peut donc écrire que pour tout i:

$$RC_i = \sum_j Pij * \alpha j$$

$$= \sum_j \frac{1}{112}\alpha_j$$

$$= \frac{1}{112}\sum_j \alpha j = RC$$

RC_i ne dépend donc plus de i dans le cas équiprobable. On retrouve bien la définition énoncée précédemment.

Par ailleurs la matrice des probabilités MP peut être utilisée pour calculer la probabilité en partant d'un profil P_i de se retrouver dans un profil P_j après n itérations. Pour se faire il suffit d'élever MP à la puissance *n*.

Et donc :

$$P_j = P_i * MP^n$$

Or les propriétés des chaînes de Markov nous assurent que la matrice associée converge vers une valeur limite lorsque *n* tend vers l'infini. En effet, la chaine de Markov étudiée est irréductible puisque pour tous les sommets sont reliés deux à deux (graphe fortement connexe), elle est également récurrente positive (chaque profils peut être atteint une infinité de fois) et apériodique (il n'y a pas de suite de profils qui se répète nécessairement). Le théorème de Perron Frobenius démontre qu'elle converge donc vers une distribution stationnaire qui nous assure que la probabilité d'avoir un profil P_j après un grand nombre d'itérations ne dépend pas du profil initial P_0.

L'utilisation du vecteur de probabilités stationnaire sera détaillée sur un exemple dans la section 4.7 du chapitre suivant.

3.6 Conclusion

Ce chapitre a permis d'introduire un premier estimateur de la résilience qu'est le Ratio de Cohérence. Cette mesure définie formellement nous permet non seulement d'évaluer un système mais également d'orienter nos efforts pour améliorer sa résilience. Nous avons donc proposé des outils d'analyse de sensibilité qui s'inscrive dans la démarche de proposer une méthode de développement qui a pour objectif de garantir au maximum les propriétés de sûreté de fonctionnement lors des évolutions d'un système.

Dans le chapitre suivant nous verrons de nouvelles mesures et de nouveaux outils de simulation qui proposeront une approche plus temporelle de la résilience en particulier au travers de l'étude de scénarios. L'objectif est notamment de nous permettre d'étudier plus précisément la résilience du système dans le cas où les changements seraient peu nombreux.

Chapitre 4 - Quantification de la résilience et mesures temporelles

« C'est quand la dernière fois qu'on s'est retrouvé tous d'accord sur un truc ? »
Arthur, Kaamelott, livre IV.

4.1 Introduction

Dans ce chapitre nous définissons de nouvelles mesures et de nouveaux outils de simulation qui proposeront une approche temporelle de la résilience en particulier au travers de l'étude de scénarios.

L'évaluation de la résilience d'un système doit se faire au cours du temps, c'est-à-dire pendant une durée de vie suffisamment longue pour représenter un scénario d'évolution composé d'évènements de changement. En conséquence il nous faut définir des métriques qui ne prennent pas seulement en compte l'aspect probabiliste des événements mais également leurs conséquences sur le système tout au long de sa vie opérationnelle. On définit ainsi un scénario comme une série finie d'événements datés qui définissent une séquence d'évolution pour une application donnée.

Chaque événement e_i arrive à un temps t_i et provoque une modification d'un ou plusieurs paramètres du modèle ce qui se traduit dans notre modèle par un changement de profil. L'objectif de ce chapitre est de mesurer l'impact d'une séquence, appelée scénario par la suite, d'évènements en termes de résilience.

Nous proposerons dans un premier temps une classification des évènements afin d'introduire la mesure **RE(s,t)** pour mesurer la résilience d'une application sur un scénario **s** d'une durée **t**. Par la suite nous introduirons les mesures du *Mean Time To Inconsistency*, du *Mean Time To Repair Inconsistency* et du *Mean Time Between Inconsistency* afin de mieux comprendre ce qu'est une application résiliente. Enfin nous conclurons ce chapitre par l'analyse détaillée d'un exemple.

4.2 Classification des événements

On peut d'ores et déjà noter que les événements peuvent avoir trois conséquences possibles. Soit $FTM_0 \Diamond A$ une application et un mécanisme de tolérance aux fautes associé. On suppose qu'à l'instant initial l'association des deux composant est cohérente. On suppose qu'à l'instant t un événement survient.

Un premier cas possible est que cet évènement n'impacte en rien la propriété de cohérence entre FTM_0 et la nouvelle version de l'application que nous appellerons A_1. En d'autres termes le système est résilient et les propriétés de sûreté de fonctionnement sont toujours garanties. Par exemple on a la configuration $PBR \Diamond A$, avec A déterministe. Le premier évènement transforme l'application en une nouvelle version non déterministe. PBR étant compatible avec des applications non-déterministes il n'y a pas d'impact sur la cohérence de la configuration.

Dans un second cas, cette nouvelle version de l'application n'est plus cohérente avec le mécanisme de tolérance aux fautes précédemment utilisé. Cependant il existe, parmi la

banque de mécanismes dont nous disposons, une solution de remplacement de FTM_0 par FTM_1. De plus ce changement de mécanisme peut être effectué dans un délai suffisamment court que nous noterons δt. Cet événement a donc entrainé une incohérence dite *transitoire*. On peut reprendre l'exemple d'une configuration PBR◊A, avec A déterministe. On suppose qu'un changement de version implique la perte de l'accès à l'état de l'application. PBR n'est plus cohérent avec A mais on dispose du mécanisme LFR qui peut le remplacer en laps temps réduit.

Enfin il reste le cas où ce changement impacte la cohérence de la configuration courante et il n'existe pas de solution ou, si elle existe, le temps de déploiement est bien trop important aux vues des fréquences d'apparition des évènements. L'incohérence est donc persistante et rend l'application non-sûre durant un intervalle de temps Δt non négligeable. C'est le cas d'une application A associée à un mécanisme de type LFR pour tolérer les fautes par crash qui deviendrait sujette à des fautes en valeurs. La seule solution serait de déployer un mécanisme composite LFR+TR mais celui-ci n'est pas disponible immédiatement. Dans ce cas Δt représente le temps nécessaire pour développer la solution LFR+TR et la déployer sur le système via une mise-à-jour par exemple.

La question que l'on se pose à présent est la suivante : comment peut-on quantifier l'impact d'un événement sur la résilience d'un système ? Pour répondre à cette question nous avons besoin d'introduire d'autres mesures de la résilience.

4.3 RE(s,t)

Le pire cas en tant que concepteur du système apparait lorsque l'application n'est plus cohérente avec un mécanisme de tolérance aux fautes alors que l'on sait que la probabilité d'occurrence d'une telle faute (celle à considérer dans le modèle de fautes de l'application) n'est pas négligeable. Une première approche consiste donc à quantifier le pourcentage d'événements qui introduisent des incohérences.

On définit ainsi RE(s,t) comme le rapport du nombre d'événements n_i n'ayant pas impacter la sûreté de fonctionnement d'une application et du nombre n d'évènements total survenus au cours du scénario s durant l'intervalle [0,t] .

Notons que si un événement survient alors que le système est déjà dans un état d'incohérence persistante alors il compte également comme un évènement qui rend l'application non sûre de fonctionnement.

Cette mesure se définit donc comme la disponibilité de la sûreté d'une application du point de vue des mécanismes de tolérance aux fautes. Plus la valeur de RE(s,t) est élevée pour une application, plus la capacité du système à garantir les propriétés de sûreté de fonctionnement de cette application est grande en dépit des changements. Dans notre cas cela se traduit donc par la capacité du système à garder FTM◊A cohérent et donc à assurer la résilience.

Notons par ailleurs que RE(s,t) est analogue à la mesure de la disponibilité telle que définit pour les systèmes sûrs de fonctionnement. Alors que la fiabilité quantifie la continuité de service de l'application fonctionnelle au sens de la norme ISO8402 [29], la mesure RE(s,t) quantifie la continuité de service de la partie non-fonctionnelle associée à cette application.

Prenons le cas simple d'une application A_0 déterministe, silencieuse sur défaillance dont on a accès à l'état et qui subit uniquement des fautes par crash. Nous disposons de deux solutions pour assurer la sûreté de fonctionnement de cette application : nous pouvons utiliser un mécanisme de type LFR ou un mécanisme de type PBR.

Trois évènements surviennent au cours du scénario sur lequel nous proposons de calculer RE(s,t) :

-e_1 : l'application devient non déterministe.

-e_2 : l'application redevient déterministe.

-e3 : l'application devient non-déterministe et subit des fautes en valeurs.

On propose de mesurer RE(s,t) dans le cas où le mécanisme initial choisi est PBR et dans le cas où on a choisi LFR. Le premier cas est illustré par la figure 19.

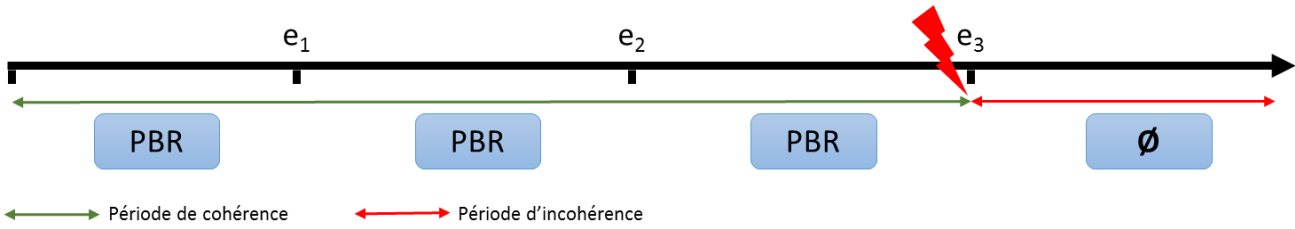


Figure 19 - Scénario avec PBR - Calcul de RE(t,s)

Si l'on choisit PBR comme mécanisme initial alors les deux évènements initiaux qui affectent le déterminisme de l'application n'ont aucun effet sur la cohérence entre l'application et PBR. En revanche le troisième évènement introduit des fautes en valeurs et nous ne disposons d'aucun mécanisme permettant de les tolérer puisque l'application est non-déterministe, il y a donc une incohérence persistante. On peut donc calculer que sur cet exemple :

RE(s,t)=2/3

En choisissant LFR comme mécanisme initial le cycle de vie de l'application n'est plus du tout le même comme on peut le constater sur la Figure 20.

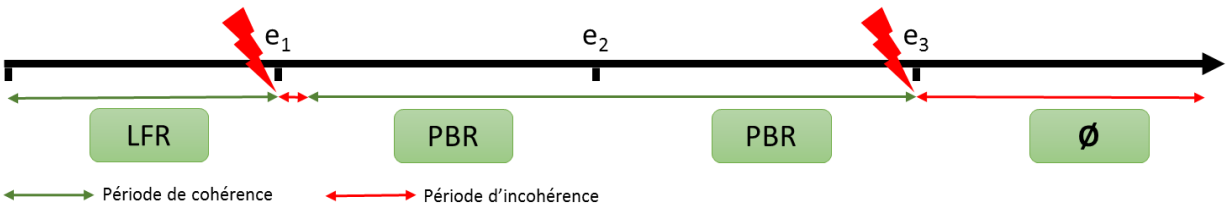


Figure 20 - Sénario avec LFR - Calcul de RE(s,t)

Le premier évènement e_1 rend l'application non-déterministe ce qui provoque l'incompatibilité et donc l'incohérence entre l'application et le mécanisme LFR. Pour retrouver les propriétés de sûreté de fonctionnement nous décidons d'installer un mécanisme PBR qui est cohérent et qui permet de tolérer les fautes par crash. L'évènement e_1 a donc généré une incohérence transitoire. L'évènement e_2 n'a alors aucun impact car le déterminisme de l'application n'est pas une hypothèse nécessaire à l'utilisation de PBR. Enfin le troisième évènement introduit une incohérence persistante comme on l'a déjà décrit pour le cas précédent. Puisque deux évènements sur trois ont introduit des incohérences (transitoires ou persistantes) on obtient la mesure suivante :

RE(s,t)=1/3

Cet exemple permet donc de conclure que pour ce scénario et compte tenu des mécanismes de tolérance aux fautes disponibles {PBR, LFR}, le choix de PBR comme mécanisme de tolérance aux fautes initial est meilleur pour la résilience du système puisque la valeur du RE(s,t) mesurée est plus grande que celle mesurée avec LFR comme mécanisme de départ.

La mesure de RE(s,t) permet donc de quantifier l'impact d'une stratégie de sélection de mécanismes de tolérance aux fautes. Cependant elle se limite à une utilisation sur des scénarios spécifiques. Pour généraliser la mesure de la résilience d'un système il nous faut introduire des indicateurs qui ne dépendent pas de scénario spécifiques et qui seront des indicateurs statistiques.

4.4 Mean Time To Inconsistency: MTTI

Lorsque l'on souhaite mesurer la résilience d'un système on analyse les périodes durant lesquelles l'application n'est pas protégée, c'est-à-dire non sûre de fonctionnement. De manière analogue à la mesure du *Mean Time To Failure* (MTTF) on cherche à mesurer la durée moyenne jusqu'à la prochaine incohérence.

Tout comme le MTTF représente le temps moyen de fonctionnement sans fautes, nous définissons le MTTI (*Mean Time To Inconsistency*) comme le temps moyen durant lequel l'application et son mécanisme de tolérance aux fautes sont cohérents. Toutefois il convient de rappeler qu'une application qui n'est pas associée avec un mécanisme de tolérance aux fautes cohérent n'est pas défaillante. En effet, lors d'une incohérence le système rompt la continuité des garanties des propriétés de sûreté sans toutefois que cela entraine une rupture de la continuité de service.

Le *Mean Time To Inconsistency* est une mesure temporelle. En reprenant l'analogie avec les mesures standards de la sûreté de fonctionnement on sait qu'un système dont le taux de défaillance λ est constant a un MTTF tel que :

$MTTF=1/\lambda$.

On définit ainsi :

$$MTTI = 1/\beta$$

Avec β le taux d'événement entrainant une incohérence. Nous disposons déjà depuis la section 3.5 d'un outil visant à calculer les probabilités de passage d'un profil à l'autre. Sachant que l'on est dans un profil p_0 à l'instant t_0, on note $FTM_0 \Diamond A_0$ une composition cohérente de l'application qui correspond à ce profil et d'un mécanisme de tolérance aux fautes. On peut alors calculer la probabilité à l'instant t_1 que l'application A_0 devenue A_1 suite à un changement devienne incohérente avec FTM_0. On note cette probabilité P_{inc}.

Pour passer de cette probabilité à un taux nous utiliserons la fréquence moyenne d'apparition des changements f_{ev}. On obtient donc :

$$\beta = P_{inc} \cdot f_{ev}$$

On définit également α comme le taux d'événement tels qu'ils n'entraînent pas d'incohérence. Ainsi on note :

$$\alpha = (1 - P_{inc}).f_{ev}$$

Cette fréquence peut par exemple correspondre à la fréquence de mise à jour du système. A titre d'exemple, entre septembre 2012 et février 2014, une Tesla Model S était mise à jour tous les 34 jours environ. Dans notre cas et pour la suite des calculs nous utiliserons une fréquence de changement de 1 tous les 31 jours, soit une période entre évènements de 744h.

En reprenant l'exemple d'une application mise à jour une fois par mois dont les changements sont complétement aléatoires et pour laquelle on dispose des mécanismes LFR, PBR, TR et LFR+TR. Pour illustrer cet exemple nous remettons ici le tableau 3 du chapitre 3 sur lequel est encerclée la cellule que nous étudierons dans l'exemple.

FM / AC	**C** !O !V	!C **O** !V	!C !O **V**	**C** **O** !V	**C** !O **V**	!C **O** **V**	**C** **O** **V**
!DT,!ST,!AC,!FS							
!DT,!ST,!AC,**FS**							
!DT,!ST,**AC**,!FS							
!DT,!ST,**AC,FS**	PBR						
!DT,**ST**,!AC,!FS							
!DT,**ST**,!AC,**FS**	PBR						
!DT,**ST,AC**,!FS							
!DT,**ST,AC,FS**	PBR						
DT,!ST,!AC,!FS							
DT,!ST,!AC,**FS**	LFR						
DT,!ST,**AC**,!FS		TR	TR			TR	
DT,!ST,**AC,FS**	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,ST,!AC,!FS		TR	TR			TR	
DT,ST,!AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR
DT,ST,AC,!FS		TR	TR			TR	
DT,ST,AC,FS	LFR	TR	TR	LFR +TR	LFR +TR	TR	LFR +TR

Tableau 6 - Tableau de profils rempli avec {LFR, TR, PBR, LFR+TR}

On suppose que l'application est associée au mécanisme TR. Elle est déterministe, possède un état qui est accessible et n'est pas silencieuse sur défaillance. On souhaite tolérer les fautes en valeur. Dans ce cas il n'y a que 18 profils sur 112 qui soient cohérents avec le mécanisme TR. Cela signifie que si un changement survient alors dans 94 cas sur 112 on passera par une phase d'incohérence (transitoire ou persistante). Ainsi on peut calculer :

$$P_{inc} = 94/112 = 0.84$$

Avec l'hypothèse d'un changement par mois, soit un changement toutes les 744h on peut calculer que :

$$\beta = P_{inc} \cdot f_{ev} = 0{,}0011\ h^{-1}$$

$$MTTI = 1/\beta = 886\ h$$

Pour cet exemple on peut donc calculer que le temps moyen avant incohérence est donc d'environ 886h soit 36 jours.

Notons que nous ne faisons pas de différence entre un évènement qui introduit une incohérence transitoire et un évènement qui introduit une incohérence persistante. Nous avons donc besoin de définir une mesure qui met en avant cette différence.

4.5 Mean Time to Repair Inconsistency: MTRI

On appelle « fragile » une application qui n'est attachée à aucun mécanisme permettant de tolérer les fautes définies dans son modèle de fautes. La mesure du temps moyen pour réparer une incohérence (MTRI) évalue la durée durant laquelle l'application est fragile.

De nouveau nous pouvons faire l'analogie avec la mesure standard du Mean Time To Repair (MTTR) en sûreté de fonctionnement. Le MTTR qui peut également se définir comme l'inverse du taux de réparation qui évalue le temps moyen durant lequel l'application récupère d'une défaillance n'ayant pas interrompue la continuité de service. Par exemple, dans le cas d'une application protégé par un mécanisme de type duplex, c'est le temps moyen nécessaire à la remise en fonctionnement d'une des deux copies lorsque celle-ci a subi un crash.

Dans notre cas le MTRI mesure le temps moyen qu'il faut pour passer d'un état incohérent (i.e. l'application est fragile) à un état où l'application est de nouveau protégée par un mécanisme de tolérance aux fautes.

Comme nous l'avons décrit précédemment il existe deux types d'incohérences : les incohérences transitoires et les incohérences persistantes. Dans le premier cas, les temps de transitions peuvent être de l'ordre de la milliseconde. C'est le cas sur des plateformes adaptées telles que ROS [30] ou FRASCATI [31] comme l'on démontré des travaux précédents [19] [20]. En revanche, lors d'incohérences persistantes le mécanisme n'est pas disponible immédiatement. Cela signifie que la plateforme ne supporte pas une reconfiguration réactive

rapide du mécanisme de tolérance aux fautes (Tolérance aux Fautes Adaptative) ou bien que le mécanisme correspondant au nouveau profil n'est pas encore développé. Il faudra donc attendre une mise à jour du système pour corriger cette incohérence qui peut durer plusieurs jours voire semaines.

Plus l'ensemble des mécanismes de tolérance est complet au regard de l'ensemble des profils de l'application (c'est-à-dire que la valeur du Ratio de Cohérence est élevée) plus les temps de transitions d'un état incohérent à un état cohérent sont réduits. Le MTRI mesure donc la réactivité du système en termes de résilience.

Tout comme on définit en notant μ le taux de réparation :

$$MTTR=1/\mu$$

On peut définir :

$$MTRI=1/\gamma$$

Avec γ le taux de réparation des incohérences. L'objectif en tant que développeur d'un système est donc de diminuer autant que faire se peut le MTRI pour diminuer la fenêtre de fragilité de l'application.

4.6 Mean Time Between Inconsistencies : MTBI

Le Mean Time Between Inconsistencies (MTBI) est la mesure du temps moyen qui sépare deux événements qui entrainent une incohérence qu'elles soient transitoires ou persistante. De façon similaire aux définitions données précédemment on peut rapprocher cette mesure du Mean Time To Failure qui mesure en sûreté de fonctionnement le temps moyen qui sépare deux défaillances.

On rappelle que :

$$MTBF = MTTF+MTTR$$

Et on notera donc que :

$$MTBI= MTTI+MTRI$$

On notera que lorsque le MTTR est négligeable on peut noter que MTBF=MTTF. Par exemple cela est vrai pour les systèmes avioniques dont les délais de réparation sont cours (de l'ordre du jour) alors que les taux de défaillance sont plus petits qu'une défaillance par année.

Lorsque la fréquence d'évènements est très élevée, la résilience repose sur le fait que peu de ces événements introduisent une incohérence (MTTI élevé) et que si ils en introduisent une, les temps de récupération soient négligeables (MTRI faible). En conséquence, et à moins

d'une connaissance poussée du système étudié nous ne nous permettrons pas de simplifier l'expression en MTBI=MTTI.

4.7 Exemple

Le but de cette section est de montrer comment estimer la résilience d'un système à l'aide des mesures proposées. Pour ce faire nous choisirons des valeurs arbitraires qui toutefois nous semblent cohérente avec ce que l'on a pu obtenir au cours de travaux expérimentaux précédents [20].

Afin de concentrer l'analyse sur les mesures déjà présenté dans ce chapitre nous rappelons ici que pour obtenir un modèle d'une application il est nécessaire de suivre les étapes présentées dans les chapitres 2 et 3:

-Une analyse des modes de défaillance de l'application permet d'obtenir un ensemble de type de fautes **FM** et un ensemble de mécanismes de tolérance aux fautes **{FTM_1, FTM_2,…}**.

-Les mécanismes de tolérances aux fautes sont analysés pour isoler les caractéristiques applicatives **AC** qui ont un impact sur la compatibilité entre l'application et les mécanismes.

-En utilisant le modèle (AC, FM) et l'ensemble {FTM_1, FTM_2,…} on calcule le Ratio de Cohérence en effectuant des analyses de sensibilité afin d'ajouter des mécanismes de tolérance aux fautes pour augmenter ce ratio.

4.7.1 Le modèle

Afin de rendre l'exemple lisible nous nous contenterons d'un ensemble de trois paramètres. Il n'existe par ailleurs aucune restriction quant au nombre de paramètres lors de la modélisation puisque toutes les analyses faites dans cette thèse sont automatisables. Du côté des caractéristiques applicatives on ne considère que le déterminisme. On note DT les profils de l'application qui sont déterministes et !DT ceux qui ne le sont pas. On note ainsi l'ensemble des caractéristiques applicatives AC :

$$AC = \{DT\}$$

On considère également deux types de fautes physiques, c'est-à-dire matérielles (cf. chapitre 2): les fautes par crash C et les fautes en valeurs V. Par convention un profil dont le modèle de fautes n'inclut pas les fautes par crash sera noté !C et si il n'inclut pas les fautes en valeurs !V. On notera le modèle de fautes FM :

$$FM = \{C, V\}$$

L'ensemble des mécanismes de tolérance aux fautes que nous utilisons ici est composé de quatre éléments :

-PBR : Primary Back-up Replication ou mécanisme de redondance froide

-LFR : Leader Follower Replication ou mécanisme de redondance chaude

-TR : Time Redundancy ou mécanisme de redondance temporelle

-LFR+TR : Mécanisme composé de LFR et de TR

Rappelons ici que LFR, TR et LFR+TR requièrent que l'application soit déterministe pour pouvoir être compatible avec celle-ci. PBR quant à lui ne nécessite pas d'hypothèse concernant le déterminisme de l'application à protéger.

Du point de vue du modèle de fautes, les mécanismes PBR et LFR tolèrent les fautes par crash, TR tolère les fautes en valeurs et LFR+TR tolère les deux types de fautes (crash et valeur).

4.7.2 Probabilités de transition

Une fois les caractéristiques applicatives et le modèle de faute établis, on affecte des valeurs arbitraires aux probabilités de transition entre les différentes valeurs pour chaque paramètre. On peut modéliser ceci par les diagrammes suivants :

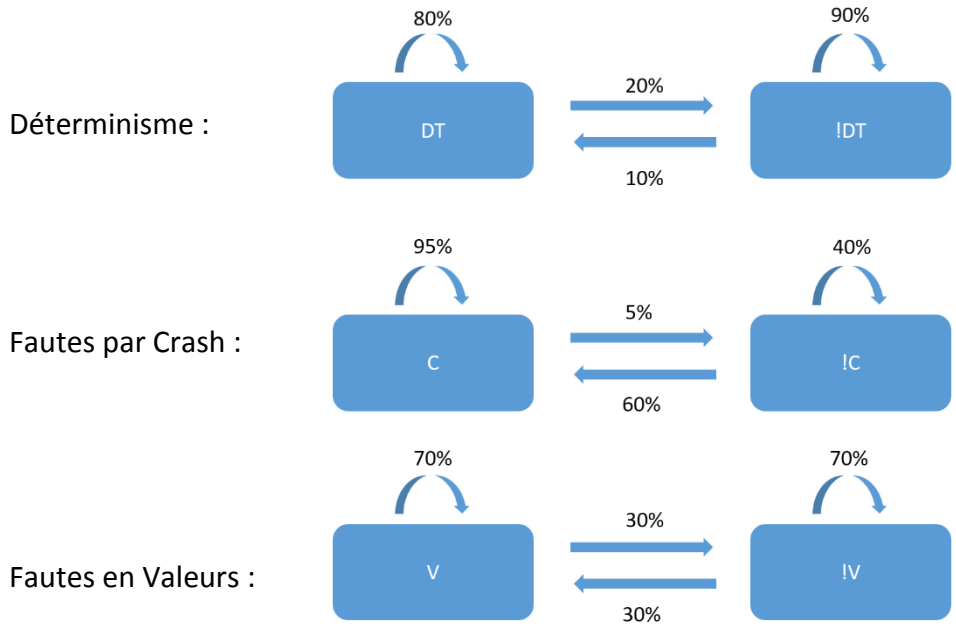


Figure 21 - Probabilité de changement des paramètres

Rappelons que la figure se lit de manière analogue à celle décrite dans le chapitre précèdent (figure 17). Ainsi, sur le premier diagramme, nous considérons qu'une application restera déterministe dans 80% des cas lors d'une mise à jour et deviendra non déterministe dans 20% des cas. Réciproquement elle restera à 90% non déterministe si elle l'est déjà et pourrait devenir déterministe dans 10% des cas restants. Une explication similaire peut être faite pour chacun des paramètres présentés sur la figure 21.

Ces valeurs sont ici arbitraires puisqu'elles dépendent en réalité de l'application, de la plateforme et du type de système étudié (grand système ou système embarqué de type

avionique, automobile, …). Elles peuvent être déduites d'études sur les versions successives des applications qu'elles modélisent, ou issues d'analyse de données recueillies sur un ensemble de système en fonctionnement.

Une fois les probabilités individuelles explicitées on peut modéliser sous la forme d'une matrice les probabilités de transition de l'ensemble des profils. Pour ce faire on regarde pour chaque profil la valeur des paramètres qui le composent par exemple, une application qui est déterministe et subit des fautes en valeur uniquement sera représenté par le profil : DT !C V.

On peut alors calculer la probabilité de transition de ce profil vers tous les autres profils de manière analogue à l'analyse conduite au chapitre 3 section 3.5. On obtient la matrice $M_1 = (a_{ij})$ où a_{ij} est la probabilité de passer du profil i au profil j lors du premier évènement. Dans notre cas on obtient la matrice M_1 suivante :

	DCV	$DC!V$	$D!CV$	$D!C!V$	$!DCV$	$!DC!V$	$!D!CV$	$!D!C!V$
DCV	0.532	0.228	0.028	0.012	0.133	0.057	0.007	0.003
$DC!V$	0.228	0.532	0.012	0.028	0.057	0.133	0.003	0.007
$D!CV$	0.336	0.144	0.224	0.096	0.084	0.036	0.056	0.024
$D!C!V$	0.144	0.336	0.096	0.224	0.036	0.084	0.024	0.056
$!DCV$	0.0665	0.0285	0.0035	0.0015	0.5985	0.2565	0.0315	0.0135
$!DC!V$	0.0285	0.0665	0.0015	0.0035	0.2565	0.5985	0.0135	0.0315
$!D!CV$	0.042	0.018	0.028	0.012	0.378	0.162	0.252	0.108
$!D!C!V$	0.018	0.042	0.012	0.028	0.162	0.378	0.108	0.252

Figure 22 – M_1 : Matrice de transition au premier évènement

De façon analogue au calcul effectué dans le chapitre précèdent, on peut à partir des probabilités de transition de chaque paramètre définir la chaine de Markov des transitions de profils. Cette chaine étant complexe nous ne la représenterons pas graphiquement ici. Cependant la figure 2 représente la matrice de transition associée à cette chaîne. On note cette matrice M_1.

On souhaite étudier l'évolution de ces probabilités après un certain nombre d'évènements pour pouvoir effectuer les mesures de MTTI, MTRI et MTBI qui sont avant tout des estimateurs statistiques qui n'ont de valeurs que pour une durée de vie (et donc un nombre d'évènements) suffisamment grand.

Par définition, élever M_1 à la puissance k revient à calculer la probabilité d'être dans un profil après k événements. Par convention on notera : $M_k = M_1^{(k)}$. Pour étudier la convergence des puissances successives de M_1 nous utiliserons une définition de la distance entre deux matrices égale au maximum de la valeur absolue de la différence coefficient à coefficient des deux matrices. Ainsi on calcule la distance entre deux matrices de $\mathbb{R}$x$\mathbb{R}$ de puissances successives k et k+1 comme étant :

$$dist(M_k, M_{k+1}) = \max(\left|a_{ij}^k - a_{ij}^{k+1}\right|)_{0 \le i,j \le n}$$

On peut ainsi mesurer la distance entre les différentes itérations ce qui nous donne dans notre cas le diagramme suivant avec en ordonnée la valeur de la distance mesurée avec la matrice précédente pour chaque itération :

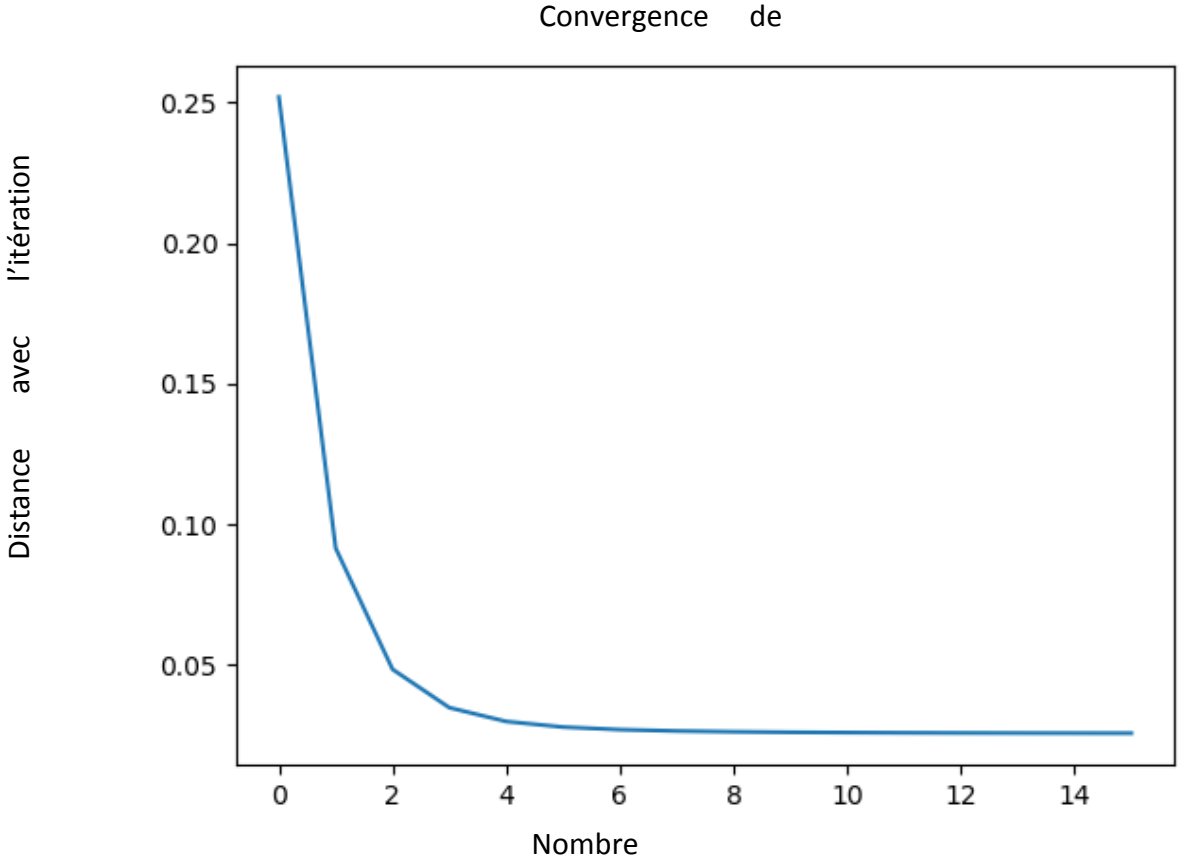


Figure 23 - Distance entre les matrice Mk+1 et Mk

On observe que la distance entre les puissances successives diminue très rapidement et tend vers une valeur proche de 0. Le 0 ne peut être atteint compte tenu des arrondis qu'opère l'algorithme de calcul. Cette observation est cohérente avec la nature de la matrice qui dérive d'une chaine de Markov. En effet, le théorème de Perron-Frobenius [22] implique que :

$$\lim_{k \to +\infty} M^k = M_{inf}$$

Avec M_{inf} une matrice unique telle que chacune de ses lignes soit identique et égale au vecteur des probabilités stationnaire que l'on note Π.

Dans notre cas la convergence est suffisamment rapide pour utiliser M_6 comme limite de cette nvergence pour la suite de nos calculs. On a donc la matrice $M_{inf}=M_6$ suivante :

$$\begin{pmatrix} 0.1546 & 0.1546 & 0.0129 & 0.0129 & 0.307 & 0.307 & 0.0256 & 0.0256 \\ 0.1546 & 0.1546 & 0.0129 & 0.0129 & 0.307 & 0.307 & 0.0256 & 0.0256 \\ 0.1546 & 0.1546 & 0.0129 & 0.0129 & 0.307 & 0.307 & 0.0256 & 0.0256 \\ 0.1546 & 0.1546 & 0.0129 & 0.0129 & 0.307 & 0.307 & 0.0256 & 0.0256 \\ 0.1535 & 0.1535 & 0.0128 & 0.0128 & 0.3081 & 0.3081 & 0.0257 & 0.0257 \\ 0.1535 & 0.1535 & 0.0128 & 0.0128 & 0.3081 & 0.3081 & 0.0257 & 0.0257 \\ 0.1535 & 0.1535 & 0.0128 & 0.0128 & 0.3081 & 0.3081 & 0.0257 & 0.0257 \\ 0.1535 & 0.1535 & 0.0128 & 0.0128 & 0.3081 & 0.3081 & 0.0257 & 0.0257 \end{pmatrix}$$

Figure 24 - Matrice de transition M_{inf}

Et on constate que toutes les lignes sont quasiment identiques. On peut donc lire dans la première colonne la probabilité d'être dans le profil (DT C V), dans la seconde la probabilité d'être dans le profil (DT C !V), … On constate que les lignes sont bien identiques il n'y a donc plus de dépendances entre le profil initial et la probabilité d'être dans un autre profil à partir de 6 évènements.

4.7.3 Regroupement par FTM

Ces probabilités ainsi exprimées sont une première étape vers la mesure de la résilience de notre système. Il nous faut à présent réfléchir en termes de cohérence du système. Pour chaque profil on cherche a associé un mécanisme de tolérance aux fautes qui soit compatible et adéquate au modèle de fautes de l'application comme défini dans le chapitre Modélisation.

Trois cas sont possibles. Soit il n'existe pas de solution, soit il en existe une seule, soit il en existe plus d'une et il est alors nécessaire de déterminer un critère de choix.

Les applications qui génèrent des erreurs en valeurs mais qui ne sont pas déterministes ne peuvent pas être associée à un mécanisme de tolérance aux fautes puisque le déterminisme est une hypothèse nécessaire à la tolérance de ce type de faute. Par conséquent pour les profils : (!DT C V) et (!DT !C V) il n'existe pas de mécanismes cohérents. Dans ce cas on utilisera le symbole Ø pour signifier l'absence de solution.

Pour certains profils il n'existe qu'une seule solution. Lorsque l'application n'a pas de faute à tolérer dans sa spécification fonctionnelle on dira que l'on est dans un cas trivial. Cela correspond donc à deux profils qui sont (DT !C !V) et (!DT !C !V). Dans ce cas il n'y a pas besoin de mettre des mécanismes de tolérance aux fautes. Pour le reste des profils ayant une solution unique :

- (DT !C V) n'est cohérent qu'avec le mécanisme TR.

- (DT C V) n'est cohérent qu'avec LFR+TR

- (!DT C !V) n'est cohérent qu'avec PBR

Enfin il reste le profil (DT C !V), pour tolérer les crash dans le cadre d'une application non déterministe deux solutions sont possibles : PBR ou LFR. Ce choix dépend des préoccupations du concepteur du système. En effet, si l'on souhaite économiser des ressources telles que l'utilisation du CPU, de la RAM et donc de l'énergie on s'orientera vers un mécanisme de redondance froide (PBR). Cependant on peut chercher à réduire le temps de reprise lors d'un crash, en termes de disponibilité les mécanismes de redondance chaude sont meilleurs. Dans notre cas on choisit d'associer le profil (DT C !V) au mécanisme LFR.

On résume l'ensemble de ces choix dans le tableau 7.

FM / AC	!C !V	C !V	!C V	C V
DT	Trivial	LFR	TR	LFR+TR
!DT	Trivial	PBR	Ø	Ø

Tableau 7 - Analyse des profils avec {LFR, PBR, TR, LFR+TR}

Le vecteur Π définit dans la sous-section précédente peut-être réduit pour représenter les probabilités stationnaires non plus de chaque profil mais de chaque configuration. Dans le cas où il y a plusieurs profils correspondant à une même configuration alors la probabilité d'être dans cette configuration est égale à la somme des probabilités des profils correspondant. Dans notre cas cela se traduit par :

$$P_{triv} = P_{DT\ !C\ !V} + P_{!DT\ !C\ !V}$$

$$P_{\emptyset} = P_{!DT\ !C\ V} + P_{!DT\ C\ V}$$

$$\begin{pmatrix} LFR + TR & 0.1546 \\ LFR & 0.1546 \\ TR & 0.0129 \\ Trivial & 0.0385 \\ \emptyset & 0.3326 \\ PBR & 0.307 \end{pmatrix}$$

Figure 25 – Vecteur des probabilités stationnaires des configurations

Notons que la probabilité d'être dans un profil pour lequel il n'existe pas de solution se rapproche de la probabilité complémentaire du ratio de cohérence soit environ 1-RC. Cette valeur n'est pas exactement identique puisque lorsque l'on calcul les probabilités stationnaires des configurations, on ne néglige pas les cas triviaux contrairement à ce qui est fait dans le calcul de RC puisque RC nous sert à qualifier un ensemble de mécanismes de tolérance aux fautes.

4.7.4 Modélisation de la sûreté de fonctionnement

Une des manières de modéliser la sûreté de fonctionnement d'un système est d'utiliser les chaînes de Markov. Nous proposons dans cette partie d'étendre la modélisation markovienne de la sûreté de fonctionnement afin d'y inclure les mesures décrites dans les sections précédentes de ce chapitre.

Prenons le cas d'un mécanisme de type duplex, un système protéger par un tel mécanisme peut se trouver dans trois états différents. Premièrement les deux copies sont opérationnelles, c'est-à-dire qu'aucune défaillance n'a encore affecté le système. Le deuxième état survient lorsqu'une des deux copies défaillent (crash). Si on note λ_c le taux de défaillance

d'une copie par crash alors le taux de défaillance de l'une ou l'autre est égale à $2\lambda_c$. Pour repasser dans l'état avec deux copies fonctionnelles il est nécessaire de réparer la copie défaillante via un taux de réparation μ. Enfin, il existe un troisième état dit « KO » qui correspond à la défaillance de la deuxième copie alors qu'une première est déjà défaillante. Cette transition est caractérisée par un taux défaillance λ_c. On considère également que la détection de la défaillance d'une réplique est parfaite.

C'est ce que l'on peut observer sur la représentation suivante :

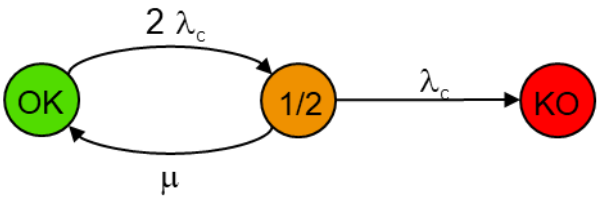


Figure 26 - Modèle Markovien d'une stratégie duplex

Pour ce système on peut calculer un taux de défaillance instantané du système avec mécanisme duplex noté λ. Ce taux de défaillance augmente en fonction du temps. A l'instant 0 le taux de défaillance est nul puisque le système est entièrement protégé. Toutefois lorsque l'on se rapproche de la valeur du MTTF=$1/\lambda_c$. On note λ_c le taux de défaillance du système dû aux fautes par crash. On s'aperçoit que le taux de défaillance augmente progressivement pour tendre vers la valeur λ_c à l'infini, le calcul du taux de défaillance instantanée est donnée en annexe 3. On peut représenter l'évolution du taux de défaillance par la courbe de tendance de la figure 27.

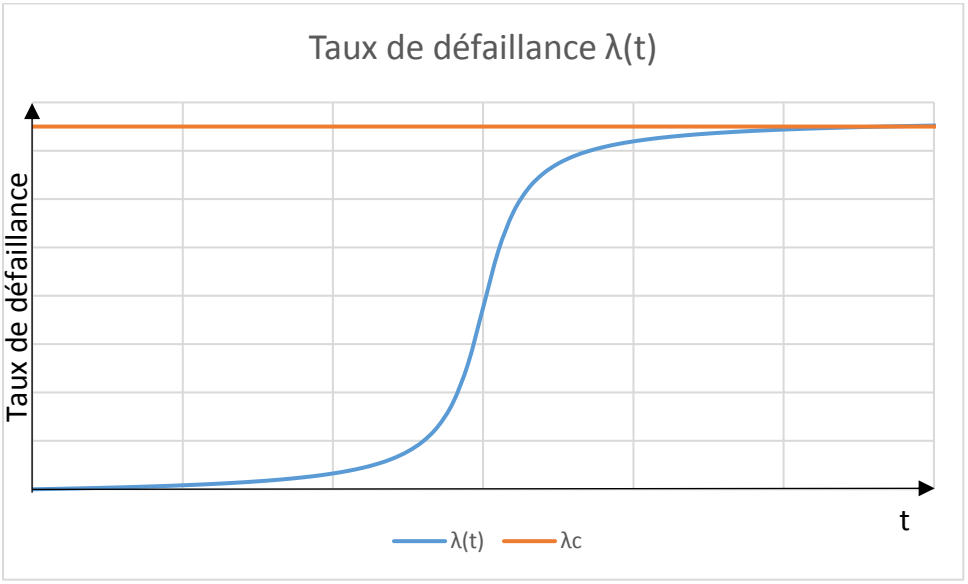


Figure 27 - Evolution du taux de défaillance en fonction du temps (stratégie duplex)

Concernant les mécanismes de type réplication temporelle TR la chaine de Markov ne fait apparaitre que deux états : OK et KO. La probabilité pour passer du premier au second est égale au taux de défaillance dû à une faute en valeur λ_v multiplié par un facteur que l'on note p. Ce facteur est égal au pourcentage de fautes que le mécanisme ne peut pas détecter, c'est-à-dire à la probabilité de défaillance du mécanisme de comparaison. Il est égal à 0 si le

mécanisme est supposé parfait et à 1 si le mécanisme est inutile. Cela se modélise par la chaine de Markov suivante :

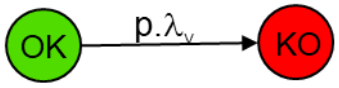


Figure 28 - Modèle markovien d'une stratégie de réplication temporelle

Enfin pour un mécanisme composite LFR+TR la chaine de Markov associée est une fusion des deux précédentes. Les taux de défaillance peuvent s'ajouter et on peut passer de l'état OK à l'état KO directement en cas de non détection d'une faute en valeur. On a donc la chaine suivante :

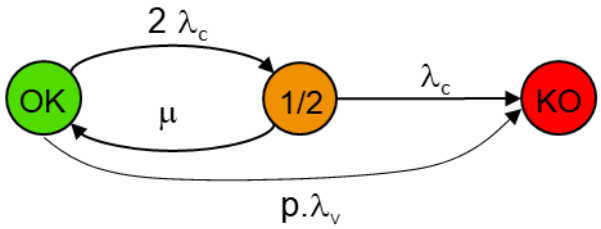


Figure 29 - Modèle markovien d'une stratégie composite duplex/redondance temporelle

Il est à présent temps d'inclure dans ce modèle les notions d'incohérence ainsi que les probabilités de transition entre profils afin de mesurer la résilience du système.

4.7.5 Mesure de la résilience

Afin de mieux comprendre les mesures qui sont faites, on effectue une représentation sous forme de chaine de Markov en utilisant les notations précédemment définies. Pour rappel :

-β_{FTM} est le taux d'événements qui conduisent à une incohérence de ce FTM.

-γ_{FTM} est le taux de récupération d'une incohérence de ce FTM.

Sur la figure suivante on représente un système pour lequel l'application est protégée soit par un mécanisme de type PBR, soit par un mécanisme de type LFR. On note I les états d'incohérence transitoires. Lorsque l'application est protégée par PBR il y a une probabilité qu'un évènement rende PBR et l'application incohérents. Ces évènements dont le taux est noté β_{PBR} peuvent survenir en fonctionnement normal (i.e. deux copies fonctionnelle, état OK) ou lorsqu'il y a déjà eu une faute par crash (i.e. une seule copie fonctionnelle, état ½). Le raisonnement est identique pour le mécanisme LFR. Enfin on note que lorsque l'application est dans un état d'incohérence transitoire, une faute par crash entraine une défaillance puisqu'il n'y aucun mécanisme en place.

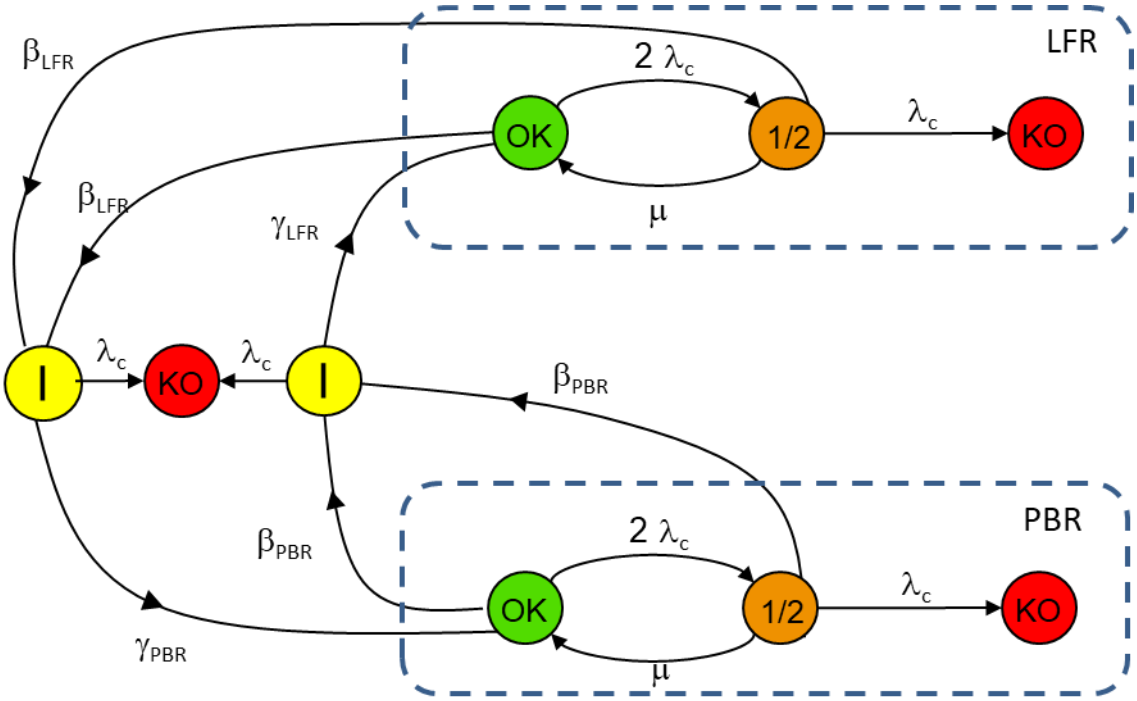


Figure 30 - Modèle markovien du système

Dans cette représentation les deux états I sont dissociés mais peuvent être agrégés de même que les états KO. Toutefois pour des raisons de lisibilité il a été choisi de les garder dissociés.

Ce type de modélisation peut permettre d'obtenir un modèle comportemental du système et donc de calculer le taux de défaillance global du système compte tenu de tous les paramètres liés à la résilience du système.

En reprenant les valeurs de la matrice figure 5 on peut à présent calculer les valeurs de α, β et γ. Concernant le calcul du taux γ de récupération après une incohérence, il est nécessaire d'introduire des valeurs chiffrées concernant les temps de transition entre les différents mécanismes de tolérance aux fautes. Ces valeurs, ici en millisecondes correspondent à des valeurs cohérentes avec ce qui peut être mesuré lorsque l'on fait de la tolérance aux fautes adaptative sur une plateforme telle que FRASCATI [21]. Dans le tableau suivant, la ligne correspondant à un FTM contient les temps de transition de ce FTM vers un autre mécanisme (en colonne).

$$\begin{pmatrix} & LFR+TR & LFR & TR & Trivial & \emptyset & PBR \\ LFR+TR & 0 & 120 & 110 & 5 & 5 & 130 \\ LFR & 50 & 0 & 130 & 5 & 5 & 125 \\ TR & 50 & 130 & 0 & 5 & 5 & 110 \\ Trivial & 130 & 120 & 120 & 0 & 0 & 130 \\ \emptyset & 130 & 120 & 120 & 0 & 0 & 130 \\ PBR & 140 & 135 & 120 & 5 & 5 & 0 \end{pmatrix}$$

Figure 31 - Temps de transition entre mécanismes (en ms)

Par propriété des chaines de Markov on sait que la matrice associée aux changements de profil converge vers une matrice M_{inf} dont toutes les lignes sont égales à une unique

distribution stationnaire que l'on notera Π. Chaque composante de ce vecteur correspond à la probabilité d'avoir un certain profil après un nombre de changement suffisamment important. Ce nombre de changement minimal est défini par la vitesse de convergence des puissances de la matrice initiale.

On note π=(PLFR+TR, PLFR, …) la distribution stationnaire de la chaine de Markov correspondante à la matrice de la figure 22, pour laquelle on a sommé les probabilités qui correspondent à un même mécanisme de tolérance aux fautes. Comme nous l'avons expliqué dans la section 5.7.3.

On cherche à présent à calculer les valeurs de α et β pour les différentes configurations. On rappelle que α est le taux d'événement qui ne nécessitent pas de changer de FTM. En conséquence, lorsque l'on a un profil associé à un mécanisme ce taux est égal à la probabilité de rester dans cette configuration que multiplie la fréquence d'apparition des événements f_{ev}.

Prenons par exemple le calcul de α_{LFR+TR}. La probabilité de rester dans cette configuration est la première composante du vecteur de probabilité stationnaire de la figure 25. Par définition de α comme vue dans la section 4.4 avec $(1-P_{inc})=P_{LFR+TR}$, c'est-à-dire que la probabilité de ne pas entrainer d'incohérence est égale à la probabilité de conserver un profil cohérent avec LFR+TR, on peut écrire que :

$$P_{LFR+TR}=\alpha_{LFR+TR} / f_{ev}$$

On cherche à présent à calculer β qui est le taux d'évènements qui produisent une incohérence (et nécessite donc de changer de mécanisme). Pour calculer cette valeur, on regarde l'ensemble des probabilités des événements qui introduisent une incohérence à partir d'une configuration. En effet, si l'on suppose que l'application soit attachée au mécanisme LFR+TR, alors la probabilité pour qu'un événement introduise une incohérence est égale à la somme des probabilités d'apparition de profils incohérents avec LFR+TR.

Enfin, de manière analogue au calcul d'α on passe d'une probabilité à un taux en multipliant cette somme par la fréquence d'évènement. On obtient donc la formule suivante pour LFR+TR par exemple :

$$\beta_{LFR+TR} = \sum_{ftm \neq LFR+TR} P_{ftm} \times f_{ev}$$

Ainsi on peut calculer directement à partir du vecteur de probabilités stationnaires π la valeur de β_{LFR+TR}.

Enfin il nous reste à calculer les valeurs de γ. Pour ce faire on va partir de l'expression MTRI=1/γ. Cela nous permet un calcul littéral plus clair.

Par définition, le $MTRI_{ftm}$ est le temps moyen que l'on va mettre pour passer de l'état d'incohérence transitoire à la configuration avec *ftm*. Pour Calculer cette grandeur nous devrons une fois de plus nous reposer sur le vecteur de probabilités stationnaires π.

Reprenons l'exemple du mécanisme LFR+TR. Le temps moyen pour réparer une incohérence et retourner dans une configuration cohérente avec LFR+TR pour assurer la protection de l'application, doit prendre en compte la configuration depuis laquelle on a effectué cette transition.

Supposons qu'à l'instant t+1 on est dans la configuration LFR+TR◊A. On suppose également qu'à l'instant précédent, c'est-à-dire l'instant t, nous ne sommes pas dans la même configuration. On est donc dans une configuration *ftm*◊A, avec *ftm*≠LFR+TR. On peut déjà calculer la probabilité d'être dans chacune des configurations possibles, c'est-à-dire la probabilité qu'on soit dans la configuration *ftm*◊A sachant que *ftm* ≠ LFR+TR. On cherche à calculer le MTRI pour se ramener à la configuration cohérente avec le mécanisme LFR+TR. Par exemple la probabilité d'être dans la configuration LFR◊A sachant que LFR+TR est exclu vaut :

$$P_{LFR\backslash LFR+TR} = {P_{LFR}}\Big/{\sum_{ftm\neq LFR+TR} P_{ftm}}$$

Bien entendu on calcule cette probabilité pour tous les mécanismes différents de LFR+TR. Ensuite on extrait de la figure 11 les temps de transition depuis n'importe quel mécanisme vers LFR+TR. C'est ce que l'on obtient sur la figure suivante.

	Temps de transition vers LFR+TR (en ms)
LFR+TR	0
LFR	50
TR	50
Trivial	130
∅	130
PBR	140

Tableau 8 - Temps de transition vers LFR+TR en ms

On note $T_{ftm\rightarrow LFR+TR}$ le temps de transition moyen d'un mécanisme *ftm* vers le mécanisme LFR+TR. Et on peut donc calculer le temps moyen pour réparer les incohérences de LFR+TR comme la somme des temps de transitions depuis les FTMs pondérés par les probabilités d'être dans chacune des configurations à l'instant précédent sachant que l'on n'était pas dans la configuration A◊LFR+TR. Soit :

$$MTRI_{LFR+TR} = \sum_{ftm\neq LFR+TR} P_{ftm/LFR+TR} \times T_{ftm\rightarrow LFR+TR}$$

Les valeurs que l'on obtient pour les calculs de ces trois grandeurs sont données dans le tableau 9. Sur ce tableau apparait également la mesure de MTTI=$1/\beta_{ftm}$ qui est le temps moyen avant l'apparition d'une incohérence alors que l'on est dans la configuration *ftm*◊A, afin de pouvoir interpréter la grandeur β et la mesure γ. On peut ainsi donner ainsi caractériser tous les arcs de la chaine de Markov complète vue en figure 30.

	MTTI	MTRI	α
LFR+TR	851,47	117,78	$2{,}147.10^{-4}$
LFR	851,47	125,6	$2{,}147.10^{-4}$
TR	729,26	120,0	$1.791.10^{-4}$
Trivial	748,67	3,27	$5.347.10^{-4}$
Ø	1078,49	4,71	$4.619.10^{-4}$
PBR	1038,66	128,51	$4.263.10^{-4}$

Tableau 9 - Mesures des taux de la chaine de Markov complète

Sur ce tableau le MTTI est donné en heures, le MTRI est donné en ms et α en h^{-1}.

On constate que dans notre cas, lorsque l'application est protégée par un mécanisme de type LFR+TR par exemple, le temps moyen avant incohérence est de 1038h environ. Cette valeur est à mettre en relation avec la période d'apparition des événements qui est de 744h (soit un évènement par mois). Cela signifie que dans notre cas, lorsque la configuration LFR+TR◊A est cohérente environ 1 évènement sur trois introduit une incohérence transitoire.

On notera également les très faibles valeurs du MTRI calculées ici. Cela s'explique premièrement par le fait que l'on considère des temps de transition entre mécanismes très court (de l'ordre de la milliseconde). Enfin, la définition du MTRI dans ce cas ne prend en compte que les transitions depuis l'état d'incohérence transitoire vers un état cohérent. C'est uniquement en s'intéressant à l'effet d'une incohérence persistante que l'on pourra calculer un MTRI global au système comme nous le verrons dans la section suivante.

4.7.6 Fragilité d'un système

Si on s'intéresse au cas d'une application protégée par un mécanisme duplex on peut observer les effets d'une incohérence sur le taux de défaillance du système. En reprenant la figure 7 on constate que la perte de cohérence lié à une modification a un impact sur le taux de défaillance du système. Prenons le cas d'une incohérence qui survient avant que le MTTF ne soit atteint. Son impact peut être représenté par la figure suivante :

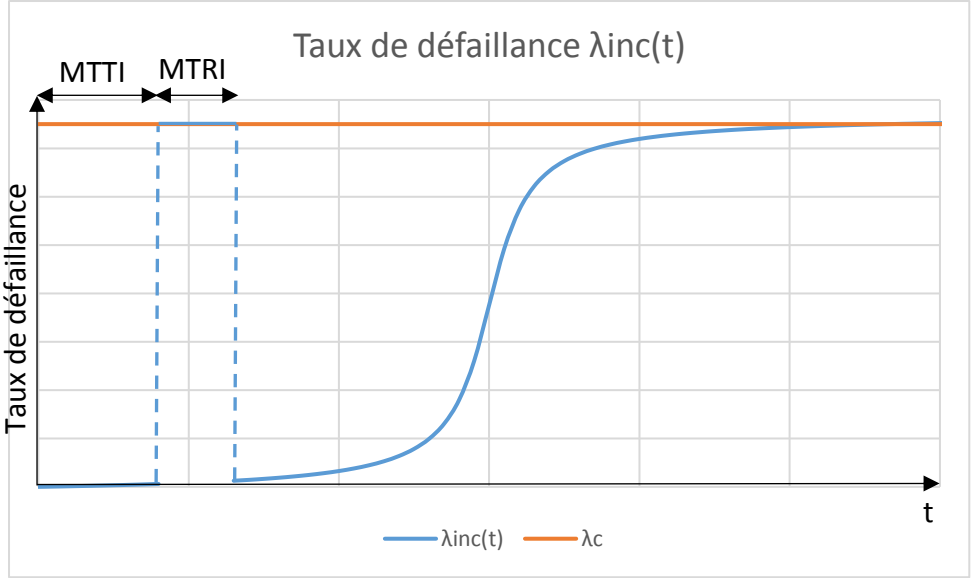


Figure 32 - Impact d'une incohérence sur le taux de défaillance du système

On remarque une discontinuité du taux de défaillance avec une incohérence $\lambda_{inc}(t)$. Lorsque l'événement survient après un temps moyen MTTI, l'application perd le bénéfice du mécanisme duplex et le taux de défaillance devient égal à celui d'une application sans protection, c'est-à-dire λ_c durant une période MTRI. Ce diagramme est donné à titre d'exemple puisqu'il n'illustre que le cas d'une application protégée par un mécanisme duplex qui devient incohérente avec ce premier mécanisme avant d'être associée à une autre solution duplex.

Cependant on peut déduire plusieurs choses, la première est que compte tenu de l'hypothèse que λ_c est constant, il n'y a pas de différence du point de vue de la probabilité de défaillance à faire un changement au tout début de la vie du système où vers un temps approchant du temps moyen jusqu'à défaillance. En effet, le taux de défaillance constant implique que le risque de faire une mise à jour est identique puisque la probabilité de défaillance alors que le système n'est pas protégé est la même à tout instant.

On pourra également considérer le cas d'un système dont le taux de défaillance suit une courbe dite « en baignoire ». Ce genre de courbe est caractéristique des systèmes électroniques et présente une phase de mortalité infantile et une phase d'usure durant lesquelles le taux de défaillance du système est largement supérieur au taux de défaillance lors du cycle de vie normale du système comme on peut le constater sur le diagramme suivant :

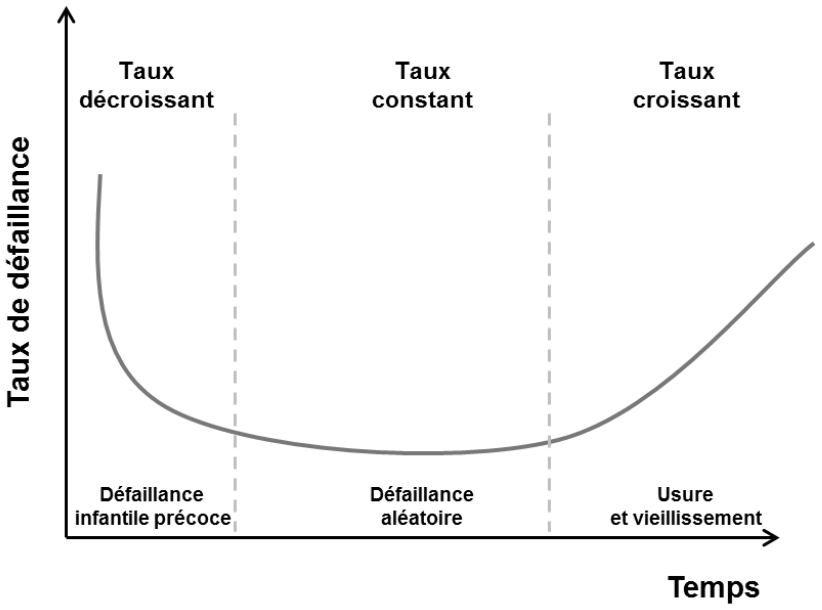


Figure 33 - Taux de défaillance en baignoire

On remarque que si l'on souhaite protéger au mieux le système il faut à tout prix éviter les périodes de fragilité lorsque le système risque des défaillances liées aux deux phases à risque (défaillances précoces et période de vieillissement). On préconise alors de limiter les sources d'événement dans ces laps de temps (ex : mises à jour autre que les mises-à-jour correctives nécessaires) quitte à sur dimensionner les mécanismes de tolérance aux fautes afin d'assurer robustesse et résilience au système.

Lors de la période de fonctionnement normale, avec un taux de défaillance minimal, on pourra considérer qu'une période de fragilité a peu d'impact. Puis lorsque les effets de l'usure et du vieillissement influent sur le taux de défaillance il faudra ajuster le modèle de fautes si nécessaire (i.e. apparition de nouvelles fautes) et s'assurer que le système ne reste jamais fragile trop longtemps, quitte une fois de plus à utiliser des mécanismes plus coûteux mais également plus résilient.

4.6.7 Incohérence et sûreté de fonctionnement globale

Changer de mécanismes de tolérance aux fautes durant la vie opérationnelle du système c'est également changer les hypothèses que l'on a faites sur les mesures standards de la sûreté de fonctionnement. En effet, pour chaque configuration on peut calculer un MTTF qui lui est propre. On cherche donc à calculer un MTTF global pour le système. Le calcul du MTTF que l'on propose est le suivant : sachant qu'après un certain nombre k de changements la matrice M^k converge vers une valeur limite par propriété des matrices stochastiques on peut calculer la probabilité d'être dans une configuration après k changement et ce sans connaitre l'état initial du système. Les valeurs de ces probabilités sont calculées dans le tableau suivant :

$$\begin{pmatrix} LFR+TR & 0.1546 \\ LFR & 0.1546 \\ TR & 0.0129 \\ Trivial & 0.0385 \\ pasdesol & 0.3326 \\ PBR & 0.307 \end{pmatrix}$$

Figure 34 - Probabilité d'être dans une configuration associée à un FTM

On peut alors calculer le MTTF global du système comme la somme pondérée par les probabilités des MTTF de chaque configuration. On note P_{FTM} la probabilité d'être dans une configuration FTM◊A. On note $MTTF_{FTM}$ le temps moyen jusqu'à défaillance associé à cette configuration. On peut donc calculer une première approximation du MTTF global du système comme :

$$MTTF_{sys} = \sum P_{FTM}.MTTF_{FTM}$$

Contrairement à la mesure du MTTF, le MTTR d'un système n'a de valeur que lorsque l'on s'intéresse à un mécanisme en particulier. En effet, dans notre cas il se peut que pour certains de nos mécanismes le taux de réparation soit nul (pas de réparation possible) ce qui impliquerait un temps de réparation moyen infini. Or, même si l'on pondère ce temps de réparation moyen par les probabilités d'apparition des configurations, un seul mécanisme quelconque dont le temps de réparation serait infini impliquerai un $MTTR_{sys}$ également infini.

En revanche on peut calculer un MTRI global au système. Pour ce faire on reprend l'ensemble des $MTRI_{FTM}$ pour chaque mécanisme de tolérance aux fautes et on somme les valeurs obtenues. On pondère cette somme avec les probabilités d'apparition des configurations comme pour le calcul du $MTTF_{sys}$. A cela s'ajoute le temps moyen passé dans les cas d'incohérence persistante. On Obtient ainsi (en gardant les notations précédentes) :

$$MTRI_{sys} = \sum P_{FTM}.MTRI_{FTM} + \frac{P_{\emptyset}}{\alpha_{\emptyset}}$$

Dans le cas de notre système on trouve une durée $MTRI_{sys} = \frac{P_{\emptyset}}{\alpha_{\emptyset}}$ =371h. En effet les temps de transitions lors des incohérences transitoires sont négligeables devant la durée passée en incohérence permanente. Cela illustre le fait que lorsque l'on parle de tolérance aux fautes adaptative le pire des cas est celui où nous ne disposons pas immédiatement d'une solution. Dans les cas d'une incohérence transitoire le système peut être considérer comme résilient.

Dans le cas d'un système dont l'adaptation des mécanismes de tolérance aux fautes n'est pas automatique, les valeurs de $MTRI_{FTM}$ prennent en compte le temps nécessaire de déploiement de la nouvelle solution qui rendra l'application à nouveau sûr. Ce temps de déploiement sera au minimum égal au temps séparant deux mises à jour. Il n'est alors plus

possible de négliger cette contribution au $MTRI_{sys}$ qui augmentera donc au détriment de la résilience de l'application.

La résilience dans le cas d'une incohérence persistante dépend de la capacité à développer et à installer un mécanisme adéquat. Ces temps de développement sont à rapprocher du processus d'occurrence de fautes et donc de la valeur du MTTF pour pouvoir quantifier la probabilité du risque encouru.

Toutes les mesures que nous avons proposées ici sont des indicateurs de la résilience d'un système. L'objectif est que ces indicateurs puissent être utilisé lors de la vie opérationnelle du système mais également lors du processus de développement comme nous le verrons lors du chapitre suivant.

Chapitre 5 – Perspectives d'intégration dans un processus de développement

"Heavier-than-air flying machines are impossible."
Lord Kelvin (1824-1907)

5.1 Introduction

Comme nous l'avons dit au début de cette thèse, la sûreté de fonctionnement est un concept dont découle un ensemble de mesures mais également d'outils qu'utilisent au quotidien les concepteurs de systèmes critiques. Nous avons présenté un ensemble de mesures associées à ce que l'on attend d'un système résilient. Or cela n'est pas suffisant pour permettre de concevoir de tels systèmes. C'est pourquoi nous proposons dans ce chapitre des outils semblables à ceux utilisés en sûreté de fonctionnement qui permettront d'aider les concepteurs de systèmes à s'orienter vers plus de résilience et ce dès les première étapes de la conception.

Analyser la résilience d'un système doit se faire dès la conception. Peut-on accepter que suite à des changements que telle ou telle application soit fragilisée, au sens ou l'avons définie dans cette thèse ? On peut dire rapidement que si cette application ne possède un niveau de criticité faible alors on peut considérer qu'il n'y a pas d'urgence à combler cette faiblesse. En revanche, si son niveau de criticité est élevé alors une attention toute particulière doit être portée sur cette application. Pour déterminer ce niveau d'urgence, nous ferons une analogie avec la notion d'AMDEC (Analyse des Modes de Défaillance, de leurs Effets et de leur Criticité) qui fait partie intégrante d'une analyse de sûreté dans un processus de développement en V.

Cependant, les processus de développement de systèmes sont en train de changer au bénéfice d'une approche « agile ». Cette approche fait l'hypothèse d'une non-connaissance complète des spécifications d'un système, finalement d'une perpétuelle évolution des besoins et donc des spécifications. Cela conduit à des développements partiels du système par incréments successifs de ses fonctions. Cela rend donc l'approche en V classique difficile à mener. Bien que certains parallèles puissent être faits entre cycle en V et étape incrémentale de production d'une version du système, la volonté de produire des versions aussi rapidement que possible écrase dans le temps le souci de validation. Ce type d'approche a démarré dans des domaines d'application très compétitifs commercialement, mais ils deviennent de plus en plus prisés dans des domaines plus critiques. On peut donc supposer que des incertitudes se feront jour sur le comportement des systèmes ainsi produits et donc sur leurs capacités de résilience.

Dans les deux approches de développement, les outils que nous avons proposés trouvent leur place. A ce titre, nous introduisons dans ce chapitre la notion de ACEC, *Analyse des Changements, de leurs Effets et de leur Criticité*.

Cette analyse met en lumière un autre aspect du développement d'un système résilient qui réside dans la nature des fautes que l'on considère et donc des mécanismes permettant de garantir la sûreté. Les AMDEC mettent en avant des propriétés très dépendantes de la sémantique des applications et dont la violation réside dans des fautes qui peuvent être matérielles ou logicielles. Ce dernier point sera illustré sur un exemple tiré d'une application industrielle.

5.2 De l'utilisation des AMDEC

5.2.1 Présentation Générale

L'analyse des modes de défaillance, de leurs effets et de leur criticité (AMDEC) [32] [33] est une démarche dont le principe fondamental est d'analyser pour chaque composant d'un système les conséquences d'une défaillance et sa probabilité d'occurrence. Cette démarche doit être systématique et collective afin d'examiner avec précision chaque mode de défaillance. Idéalement cette démarche doit être effectuée par un groupe de personnes dont les compétences englobent tous les aspects du système (de la couche matérielle jusqu'aux interactions avec l'environnement) et doit permettre à chaque participant de représenter un point de vue ou une expertise.

Ainsi pour chaque mode de défaillance il faut identifier et évaluer :

-Sa cause et son indice de fréquence.

-Ses effets et son indice de gravité.

-Les mesures mises en place pour détecter la défaillance et l'indice de détection.

On calcule à partir de ces indices la criticité du mode de défaillance comme le produit des trois indices précédents. En fonction de cette criticité les concepteurs du système peuvent faire des choix d'architecture, définir des mécanismes de protection et de tolérance aux fautes ou encore imaginer des modes de fonctionnement dégradé.

Les indices sont définis suivant des échelles arbitraires qui dépendent du type de système étudié et qui varient généralement de 1 à 10 voire de 1 à 5. Le risque 0 n'existant pas cette valeur est d'ordinaire exclue des notations. Voici un exemple de notation :

Fréquence		**Gravité**		**Détectabilité**	
Permanent	10	Pertes humaines	10	Non détectable	10
Fréquent	5	Pertes financières	5	Peut être détecté	5
Peu probable	1	Pas grave	1	Toujours détecté	1

Tableau 10 - Grilles d'évaluation pour AMDEC

A partir de ces données on peut calculer pour chaque mode de défaillance un indice de criticité. Par exemple un mode de défaillance fréquent, sans gravité mais non détectable possède un indice de priorité de risque de 5*1*10 = 50.

Une autre capacité des AMDEC est de classer les modes de défaillance en fonction de leur criticité. Il s'agit de fournir un barème selon lequel chaque mode de défaillance peut-être :

-Négligeable

-Acceptable

-Indésirable

-Inacceptable

Cette notation

On obtient alors une matrice semblable à la matrice suivante par exemple :

	Gravité			
Fréquence		Peu Grave	Pertes financières	Pertes humaines
	Permanent	Indésirable	Inacceptable	Inacceptable
	Fréquent	Acceptable	Indésirable	Inacceptable
	Peu fréquent	Négligeable	Acceptable	Indésirable
	Peu probable	Négligeable	Négligeable	Acceptable

Tableau 11 - Analyse de criticité

Chaque domaine d'application propose ses propres critères concernant la criticité mais il faut noter qu'aujourd'hui on utilise les AMDEC non seulement pour l'analyse fonctionnelle mais également pour vérifier la viabilité d'un produit, pour identifier les risques liés à un processus, pour identifier les risques liés au non-fonctionnement d'un moyen de production ou encore pour anticiper les risques liés à la rupture d'un flux (de matière ou d'informations).

Nous nous intéressons ici uniquement aux AMDEC fonctionnelles comme décrite précédemment et à la façon de les inclure dans un processus de développement.

5.2.2 Cycle en V

Nous nous focaliserons dans cette partie sur l'utilisation des AMDEC lors du développement de type cycle en V d'un système critique. Ce cycle de développement est un modèle conceptuel de gestion de projet qui permet de limiter au maximum le retour aux étapes précédentes. Il s'impose dès les années 80 comme un standard de l'industrie logicielle. On peut le représenter de la manière suivante :

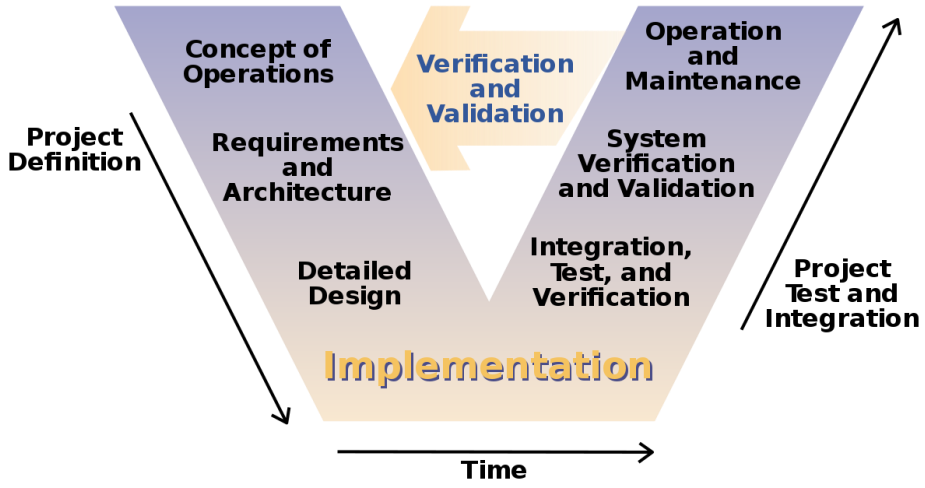


Figure 35 - Cycle de développement en V

La norme MIL-Std-1629A considère que l'utilisation des AMDEC doit se faire dès les premières phases de conception. Ainsi, chaque étape de la définition du projet doit permettre de préciser l'analyse des modes de défaillance puisque les fonctionnalités et les méthodes d'implémentation se précisent au fur et à mesure.

Notons également que lorsque le système est en phase d'opération et donc de maintenance il faut également mettre à jour les données de l'AMDEC afin d'intégrer les relevés opérationnels. En effet, un système peut rencontrer des défaillances non prévues ce qui peut conduire à un ensemble de procédure de maintenance afin de prendre en compte ces nouveaux modes de défaillance.

5.2.3 Méthodes Agiles

Les méthodes de développement agile sont un ensemble de pratique de gestion de projet qui se veulent plus pragmatiques que les méthodes dites traditionnelles (cycle en v, développement en cascade). L'émergence de ces méthodes date des années 1990, mais l'appellation « Agile » date de 2001 et tire son origine du manifeste Agile [34]. Parmi les méthodes agiles les plus utilisées on citera la méthode SCRUM [35] et la méthode Extrem Programming [36].

Ces méthodes reposent sur quatre valeurs fondamentales :
- Individus et interactions plutôt que processus et outils
- Fonctionnalités opérationnelles plutôt que documentation exhaustive
- Collaboration avec le client plutôt que contractualisation des relations
- Acceptation du changement plutôt que conformité aux plans

De ces valeurs découlent douze principes que nous n'énoncerons pas ici. Ce qui est intéressant de noter c'est que les méthodes agiles sont avant tout des méthodes itératives et

incrémentales. Du point de vue de la sûreté de fonctionnement cela signifie que chaque étape produit un système livrable dont il faut faire l'analyse des modes de défaillances.

Cependant, il n'existe pas à l'heure actuelle de formalisation générique d'étude de risques pour le développement agile d'application critique. C'est ce que montrent des études de l'université de Carnegie Mellon telles que [37]. On note que même si les méthodes agiles permettent une qualité de code meilleure que celle des méthodes traditionnelles, la question de la sûreté de fonctionnement n'est pas au centre des préoccupations des concepteurs puisqu'il faut avant tout satisfaire les besoins fonctionnels du client.

Certaines études montrent que l'inclusion d'une AMDEC devrait prendre place à la fin de chaque *sprint*, c'est-à-dire pour chaque itération dont le produit final est considéré comme livrable [38]. On note également l'émergence de méthodes agiles qui incluent la sûreté de fonctionnement dans le développement. C'est le cas de SafeScrum [39] dont on peut résumer l'utilisation au schéma suivant :

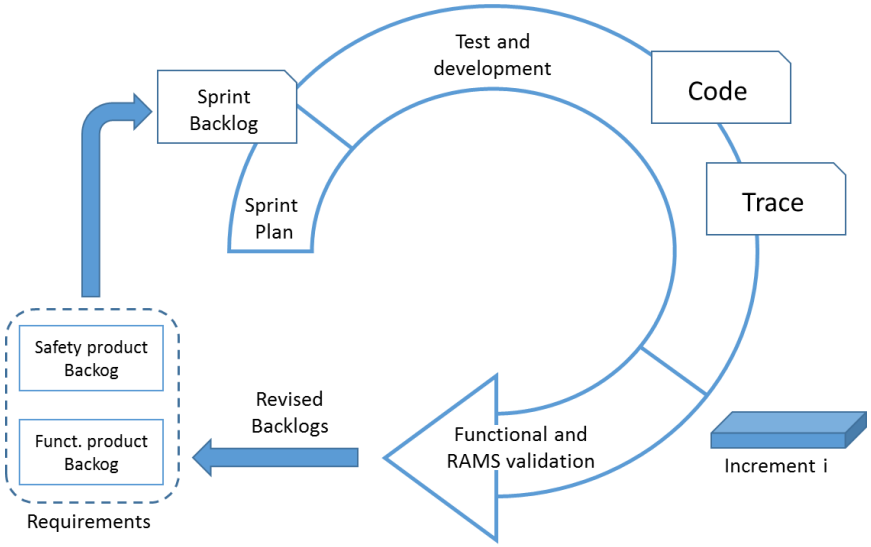


Figure 36 - Cycle de développement de la méthode agile Safescum

Dans cette méthode, chaque *sprint* produit un ensemble de documentation (*safety product backlog*) permettant d'avoir un suivi des propriétés non fonctionnelles du système. Ce que nous proposerons dans la suite de ce chapitre c'est d'inclure les outils que nous avons développés au cours des chapitres précédents dans des processus de développement.

5.3 Analyse des Changements, de leurs Effets et de leur Criticité (ACEC)

5.3.1 Présentation générale

Nous proposons ici une méthode d'analyse des changements qui, à l'instar des AMDEC, permet de classer les changements en fonction de leurs effets sur la cohérence et de leurs probabilités d'apparition afin de mieux mesurer leurs impacts en termes de résilience pour le système.

Pour chaque changement trois paramètres sont à étudier :

-Sa fréquence d'apparition

-Les effets sur la cohérence de l'application concernée par ce changement

-Sa détectabilité

La fréquence des changements dépend du type de système que l'on étudie. Dans le cadre d'un système de type satellite, les modifications du logiciel via des mises à jour sont quasiment inexistante. En revanche dans des véhicules de types Tesla on considère que la fréquence de mise à jour peut être supérieures à une par semaine. Comme pour les AMDEC on peut attribuer une note en fonction de la fréquence comme dans le tableau suivant :

Fréquence	
Plus de 1/semaine	10
1/semaine	8
1/mois	6
1/an	4
Moins de 1/an	2

Tableau 12 – ACEC : Fréquence d'apparition des changements

Les effets des changements peuvent se quantifier au regard de leur impact sur la cohérence de l'application qu'ils affectent. Ainsi, dans le meilleur des cas un changement n'aura pas d'impact sur la cohérence entre l'application et le mécanisme de tolérance aux fautes qui la protège, l'application est alors considérée comme résiliente. Dans notre exemple, nous considèrerons qu'un système capable d'adapter ses mécanismes de tolérance aux fautes dans un temps négligeable est résilient.

Dans le pire des cas, l'incohérence introduite par le changement provoque une défaillance du système. C'est le cas par exemple d'une application devenant non déterministe alors qu'elle est associée à un mécanisme de redondance temporelle qui détectera alors des fautes en valeurs puisque toutes les sorties de l'application seront différentes alors que le contexte est le même (i.e. les données d'entrée et l'état du système). Entre ces deux extrêmes il y a les cas où le mécanisme incohérent peut générer de temps en temps des fausses alarmes,

le cas où il ne protège plus l'application mais ne gêne pas l'application et enfin le cas où le mécanisme offre une couverture partielle des fautes.

Tout ceci peut être représenté par une notation analogue à celle présentée dans la table suivante :

Effets	
Défaillance de l'application	10
Faux positifs	8
Pas de protection	6
Protection partielle	4
Aucun	2

Tableau 13 - ACEC : Effets des changements sur l'application

Enfin, on considère la détectabilité de ces changements. Le pire des cas en tant que concepteur est de devoir faire face à un évènement imprévisible. Ce cas correspond par exemple à l'apparition soudaine de nouveaux de types de fautes. Cela peut être le cas avec des composants matériels dont le vieillissement est prématuré et qui entraînent des fautes de l'application. Ensuite on considère qu'il existe des changements prévisibles, comme dans le cas d'usure dont on prévoit d'assurer la maintenance à plus ou moins long terme. Enfin viennent les changements prévus comme les mises-a-jours de fonctionnalités.

Détectabilité	
Imprévisibles	10
Prévisibles	5
Prévus	1

Tableau 14 - ACEC : Détectabilité des changements

On peut donc, comme pour les AMDEC, proposé un indice de risque lié au changement qui se calcule comme le produit des coefficients exprimés dans les trois tables précédentes. Ainsi un changement prévu dont la fréquence est de 1 par mois entrainant une perte de protection aura un indice de fragilité de 36 (Fréquence : 6, Effets : 6, détectabilité : 1).

L'objectif des ACEC est de permettre de faire une analyse des changements redoutés afin de mieux les anticiper. Dans les sections suivantes nous proposerons d'inclure cette analyse dans les processus de développement pour permettre de développer des systèmes non seulement sûrs mais également résilient.

5.3.2 ACEC et cycle en V

Comme nous l'avons évoqué dans la section 5.2.2, dans un développement suivant un cycle en V il faut partir des spécifications haut niveau et raffiner ces spécifications jusqu'à pouvoir coder les différentes fonctionnalités. Les retours aux phases de conceptions doivent être les moins nombreux possibles afin d'optimiser le développement.

L'utilisation des Analyse de Changements, de leurs Effets et de leur Criticité (ACEC) peut se faire dans un premier temps juste après la phase d'implémentation. En effet, à ce stade on a déjà établi une AMDEC qui fournit le modèle de faute de l'application ainsi que les mécanismes de tolérance aux fautes en place. A partir de ces informations il est possible d'établir un modèle du système et de l'utiliser pour modéliser l'application en court de développement. Une fois l'application modélisée il s'agira de conduire une ACEC de manière à pouvoir mettre en avant les changements que l'on redoute de devoir effectuer durant les phases de validation.

Une autre utilisation des ACEC concerne la phase d'opération dans laquelle il nous faut assurer la maintenabilité de l'application. Or la maintenabilité d'un système est la capacité à réparer le plus rapidement et à moindre coût. Une analyse des changements permet de préparer au mieux les phases de maintenance. En effet, en connaissant les évènements redoutés qui ont un impact sur la cohérence de l'application étudié les concepteurs seront capables de développer préventivement des mécanismes de tolérance aux fautes.

Par exemple on peut considérer un changement comme le vieillissement prématuré d'un capteur qui se traduirait par l'apparition de fautes en valeurs. Alors on peut conduire une analyse en prenant en compte la probabilité de ce changement et son impact en termes de criticité pour le système. Dès lors il sera peut-être utile de développer une solution logicielle (par exemple un mécanisme de redondance temporelle) qui ne sera déployée que si l'on constate effectivement que l'évènement redouté est survenu.

Se pose alors la question de la détection du premier évènement indésirable. L'erreur provenant d'un capteur dans cet exemple on peut considérer que la détection de l'évènement est une probabilité dépendante de l'âge du capteur. On peut également imaginer plusieurs niveaux de défense et un diagnostic d'erreur à un plus haut niveau d'abstraction qui conduirait à la mise en doute de la fiabilité du capteur.

Ce développement préventif permet plusieurs avantages. Premièrement il permet une plus grande réactivité en cas de changement effectifs, il améliore donc la maintenabilité du système. On peut dans ce cas-là parler de maintenance préventive. De plus le processus de développement du mécanisme adapté n'est pas dissocié du processus de développement du système complet et permet donc de grouper les efforts dès la conception.

5.3.3 ACEC et méthodes agiles

Les méthodes agiles proposent une approche radicalement opposée à celle du cycle en v comme nous en avons discuté dans la section 5.1.3. Les changements sont partie intégrante de ces processus.

Lors du développement agile d'une application on ajoute des fonctionnalités progressivement. Supposons que nous sommes en cours de développement d'une application en suivant une méthode agile de type Safescrum comme mentionné précédemment. A la fin du premier *sprint* nous obtenons une version v_0 de cette application. Conformément à la

spécification de la méthode cette version est fonctionnelle mais elle ne remplit pas l'intégralité du cahier des charges tel qu'énoncé par le client lors de la commande de cette application. On suppose qu'à la fin de ce *sprint* l'AMDEC correspondante à l'application a était produite.

A ce stade du développement on sait deux choses : la première c'est que les besoins du client peuvent changer et qu'il faudra que l'équipe de développement soit capable de s'adapter à ces nouvelles exigences. La seconde c'est que l'on est en train de choisir quelles seront les fonctionnalités à implémenter dans le prochain sprint.

Concernant les besoins clients il est difficile de prévoir précisément quel sera leur évolution. Toutefois il est raisonnable de penser à anticiper ces besoins même si ceux-ci peuvent sembler peu probables. Prenons le cas d'une application ayant accès à des données sensibles dans un véhicule (la trace GPS par exemple). Un scandale relevant de la préservation de la vie privée pourrait forcer un client à prendre la décision de chiffrer les données de son application. Ce type de fonctionnalité peut par exemple utiliser des fonctions pseudo-aléatoires de génération de clés. Ce qui rend impossible la synchronisation de plusieurs copies sensées être identique et dont les exécutions s'effectuent parallèles dans un système distribué. Il y a donc possiblement un impact si l'application était protégée des fautes par crash par un mécanisme de redondance semi-active par exemple. Or l'ACEC permet d'anticiper ce type de changement en s'intéressant à l'avance à leurs effets ce qui permet de pouvoir préparer des solutions en cas de changements qui introduisent des incohérences.

De manière analogue, lorsque l'on termine un *sprint* il faut prévoir quelles seront les fonctionnalités à apporter sur l'application en cours de développement. Il est donc utile à ce moment-là de préparer une ACEC pour maitriser au mieux l'impact des nouvelles fonctionnalités sur la sûreté de fonctionnement de ce qui a déjà été développé lors des *sprints* précédents. Ainsi la conception des mécanismes de tolérance aux fautes (s'ils ne sont pas déjà utilisables) peut se faire avant même d'effectuer une AMDEC dans le but d'élargir la couverture de la tolérance aux fautes en prévision d'évolution ultérieures pouvant affecter la sûreté de fonctionnement.

5.4 Exemple

Les mécanismes dont nous avons fait usage pour définir et analyser notre méthode d'évaluation de la résilience d'un système sont peu nombreux et concerne une classe de faute particulières, les fautes matérielles. Ce sont des mécanismes génériques.

De façon similaire, des mécanismes génériques de tolérance aux fautes logicielles peuvent être considérés. Ces mécanismes se distinguent des précédents du point de vue de leur dépendance vis-à-vis des applications. On peut citer les assertions exécutables combinée à un mécanisme duplex (Assertion+Duplex), mais plus encore les *recovery blocks* [9] ou le *N-Version Programming* [10] qui font appel à de la diversification fonctionnelle. Le mécanisme *N-Self-Checking Programming* [11] fait appel à cette même notion. Si le comportement est

générique, le cœur du mécanisme est application-dépendant. Dans le cas d'une assertion simple, il s'agit d'une expression arithmétique et logique faisant appel à des données de l'application.

Nous pouvons illustrer sur un exemple différentes extensions des mécanismes, des hypothèses faites sur l'application, des hypothèses de fautes que l'on considère. Nous pourrons aussi voir l'impact d'une mise à jour sur l'application et sur le mécanisme de tolérance aux fautes.

Cet exemple est tiré des travaux de thèse de Ludovic Pintard [12]et s'intéresse à une fonction de verrouillage électronique de la colonne de direction dans une automobile (Electronic Steering Column Locking system (ESCL system). L'ESCL reçoit des commandes du tableau de bord pour verrouiller ou déverrouiller la colonne de direction. Il a la capacité d'agir sur l'alimentation et de connaître la vitesse du véhicule.

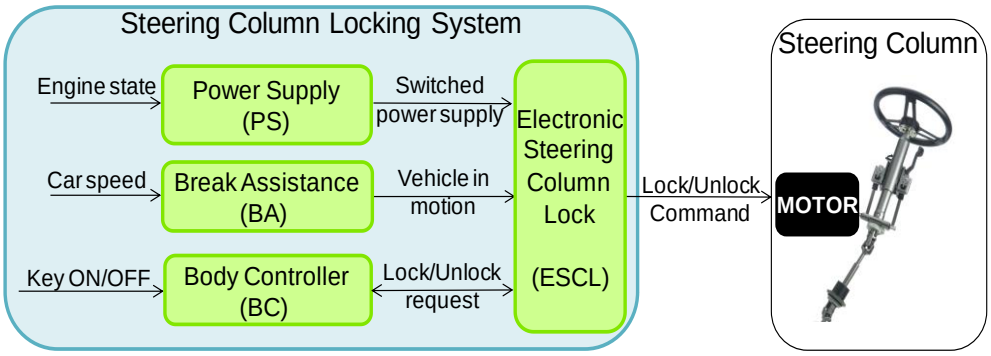


Figure 37 - Système de verrouillage d'une colonne de direction pour automobiles

L'application ESCL fournit deux services : *lock* et *unlock*. Nous nous intéressons ici qu'a la fonction lock et à ses propriétés de sûreté de fonctionnement.

Lors de l'analyse préliminaire de sûreté, on a défini un objectif de sûreté (*Safety Goal, SG*) qui porte sur le verrouillage de la colonne de direction et qui a le plus haut niveau de criticité dans le domaine automobile (ASIL D, *Automotive Safety Integrity Level*) :

SG – ASIL D – The system shall not lock the steering column when the vehicle speed is over a pre-defined threshold.

On suppose dans un premier temps que l'application ESCL est rendue sûre de fonctionnement par une assertion qui est vérifiée en interne lors de toute activation de l'ESCL. Dans le cas où cette assertion est fausse, alors une action de traitement d'erreur doit être engagée, par exemple, la coupure d'alimentation pour désactiver l'ESCL et débloquer le verrou. Elle est formulée de la manière suivante :

! [(lock=1) & car_speed > *threshold*]

La valeur *threshold* est un paramètre de configuration du système, par exemple 10 km/h.

Compte tenu du modèle de faute donné par l'analyse de sûreté et du composant étudié il n'y a qu'un seul évènement redouté : le composant ne garantit plus les propriétés de sûreté de fonctionnement. On peut imaginer que le fabricant du véhicule souhaite renforcer la sûreté de fonctionnement du système considérant que le fournisseur de l'ESCL n'est pas suffisamment fiable ou bien que pour des raisons budgétaire le constructeur automobile choisisse un ESCL n'implémentant pas de mécanisme de tolérance aux fautes interne.

Cet évènement est classé dans l'ACEC présenté en 5.2 comme :

-Peu probable avant moins d'un an (Fréquence = 2)

-Avec pour effet la perte de protection (Gravité = 6)

-Mais prévisible puisqu'annoncé par le fournisseur du composant (Détectabilité =5)

On évalue donc le risque à 2*6*5 = 60. A présent, soit on considère que ce risque est négligeable et on ne se préoccupe pas de faire en sorte d'y être résilient, soit on commence à concevoir un mécanisme pour pouvoir être résilient. Dans l'hypothèse où l'on choisirait d'être prévoyant on pourrait développer une solutions telle que celle présentée sur la figure suivante :

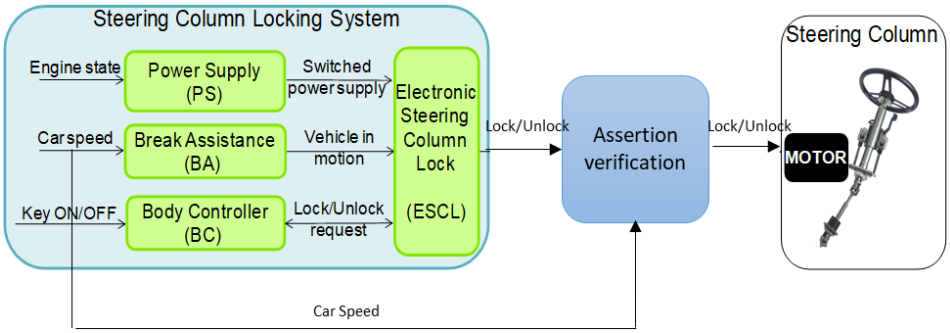


Figure 38 - Système de verrouillage d'une colonne de direction pour automobiles avec mécanisme de tolérance aux fautes

En externalisant la vérification de l'assertion *! [(lock=1) & car_speed > threshold]* dans un mécanisme de tolérance aux fautes qui viendrait intercepter les commandes envoyées par le ESCL, on parvient à assurer la tolérance aux fautes même dans le cas où on perdrait les garanties sur la sûreté de fonctionnement du ESCL.

Cet exemple conclue ce chapitre de perspectives quant à l'utilisation d'analyse de changement et de leur utilité dans des processus de développement déjà existant. La partie suivante conclura ces travaux de recherche.

5.5 Conclusion

Nous avons dans ce chapitre rapidement brossé comment la notion de résilience pouvait s'inscrire dans un processus de développement de système. La notion d'ACEC largement inspirée des AMDEC permet d'identifier les applications ou fonctions qui nécessiteront une attention particulière lors de leurs évolutions successives pendant la vie opérationnelle du système. C'est certainement cette analyse qui aidera le manager du système à monitorer le comportement de ces applications pour déterminer les mises-à-jour à faire au niveau des mécanismes de tolérance aux fautes, soit de manière réactive soit de manière préventive. Cette mise à jour peut être très large. Quand on parle de tolérance aux fautes, on parle de détection et de recouvrement d'erreur.

On peut donc imaginer de renforcer le mécanisme de détection d'erreur sans changer les actions de recouvrement. De façon similaire, on peut imaginer conserver le mécanisme de détection, mais imaginer un recouvrement plus adéquat, par exemple en imaginant un nouveau mode dégradé de fonctionnement. Nous avons plaidé à de multiples reprises pour une tolérance aux fautes adaptative, qui permet donc de faciliter la mise à jour des mécanismes de protection, de réduire le MTRI et donc d'améliorer la résilience.

On peut toujours se dire que munir les applications les plus critiques aux changements, de mécanismes ayant un spectre de fautes tolérées aussi large que possible, du genre TMR ou NSCP. En effet, on considèrerait alors a priori l'impact de toutes les fautes du matériel et du logiciel. C'est cette approche qui a été suivie pour le *Primary Fligh Computer* du Boeing 777 ou de l'Airbus 320 depuis de nombreuses années. Tout système est-il prêt à supporter le surcout de développement (diversification de fonctions) mais aussi le surcout en ressources, voire le surcout temporel, en opération ? Pas sûr que cette démarche soit acceptable pour tout système.

Nous avons donc privilégié une approche a grain fin de l'Adaptive Fault Tolerance [20] par rapport à une approche préprogrammée et paramétrée qui consisterait à munir un système de tous les mécanismes possibles à son démarrage pour en disposer en opération. Un aiguillage ou une modification de paramètre suffirait à mettre à jour les mécanismes. Cependant cette approche présente les mêmes inconvénients que ceux précédemment énoncés.

Les mécanismes spécifiques aux applications (assertions, test d'acceptation, versions) peuvent bien évidemment évoluer en fonction des mises à jour des applications ; ils sont difficiles à déterminer a priori ce qui milite encore une fois pour une approche d'adaptation des mécanismes qui soit flexible, à grain fin et dynamique

Enfin, l'intégration de cette approche dans une architecture système et sur un support d'exécution est en dehors du périmètre de cette thèse. Cependant des travaux récents et actuels ont pour objectif de fournir un support d'exécution permettant d'améliorer la résilience par une approche adaptative de la tolérance aux fautes, en particulier sur ROS auquel nous avons porté une attention particulière [19].

Conclusion

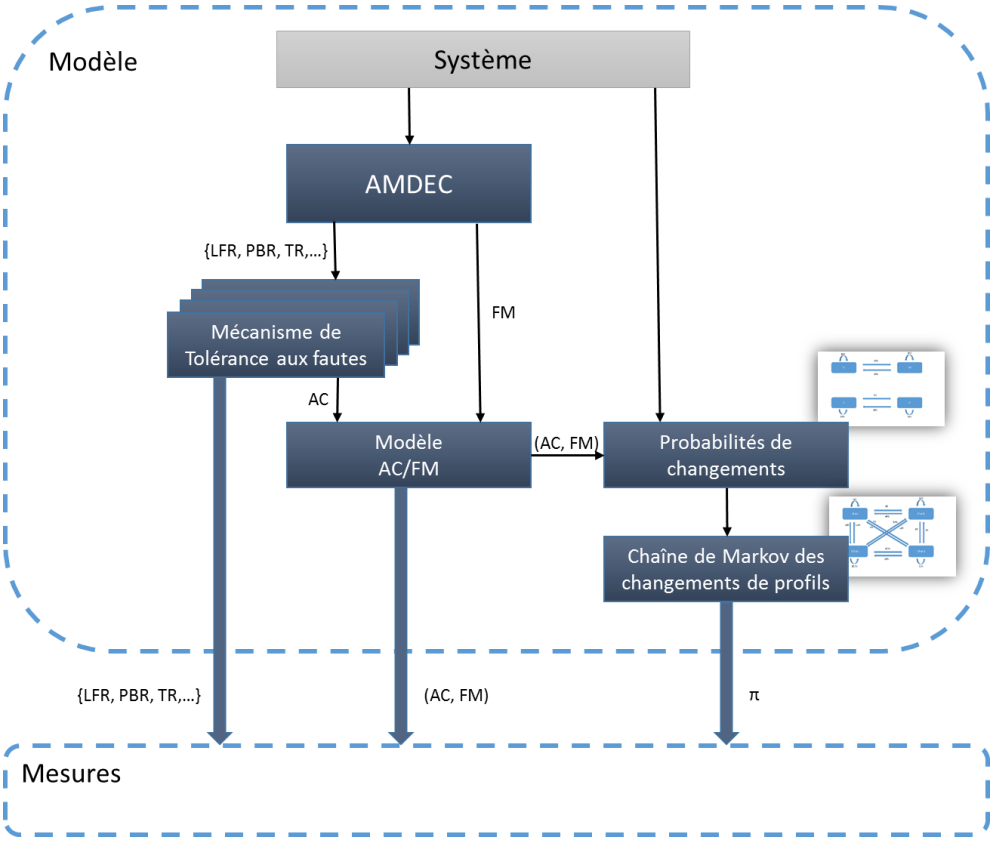
La résilience est un concept qui permet d'évaluer la capacité d'un système à demeurer sûr de fonctionnement et ce quels que soient les changements qu'il subit. Ces travaux nous ont permis de mieux comprendre ce qu'est la résilience d'un système en proposant d'une part une modélisation des paramètres qui l'affectent, et d'autre part un ensemble de mesures et d'indicateurs qui permettent de mieux appréhender ce que signifie être résilient.

Nous sommes partis du constat que la résilience est aujourd'hui une nécessité pour de plus en plus de système qui cohabitent avec nous. Elle est liée de façon claire à la fréquence des évolutions des systèmes et plus particulièrement des applications qu'ils hébergent. L'ouverture des systèmes est un autre aspect majeur du problème puisque les mises-à-jour sont faites à distance. Ce phénomène d'évolution qui est frénétiques pour des applications grand public (smartphone, pc, tablettes, …) commence à affecter des systèmes plus critiques (automobile, domotique…). Ce constat ouvre vers de nouveaux champs scientifiques dans le domaine de la sûreté de fonctionnement où tout doit être prévu a priori, y compris le modèle de fautes. Cette thèse apporte des éléments de réponse à ces problématiques.

L'analyse de ces systèmes et des Analyses des Modes de Défaillance, de leurs Effet et de leur Criticité nous ont fourni d'une part un modèle de faute (FM) et d'autre part un ensemble de mécanismes de tolérance aux fautes. Des mécanismes nous avons obtenu un ensemble de paramètres applicatifs (AC) nécessaires à leur mise en place. Ainsi nous avons pu bâtir un premier modèle (AC, FM) qui sera à la base des analyses proposées.

Les mécanismes et les paramètres retenus pour les applications nous on permis d'effectuer des analyses tout au long de cette thèse. Toutefois la méthode que nous avons proposé est extensible à d'autres paramètres applicatifs, d'autres modèles de faute et d'autre mécanismes de tolérance aux fautes comme nous avons pu l'apercevoir dans le chapitre 5 – Perspectives.

Afin de représenter le plus fidèlement la réalité nous avons proposé une modélisation des changements sous forme de chaine de Markov. En modélisant les probabilités de changements de chaque paramètre puis en les intégrant en une chaine de Markov représentant les probabilités de changements de profils nous avons pu déduire la probabilité à chaque instant d'être dans un profil spécifique (notée Π pour vecteur de probabilité stationnaire associée à la chaine de Markov). Enfin nous avons réunis les informations obtenues pour quantifier la résilience à l'aide de mesures.



Les mesures obtenues nous permettent d'analyser la résilience du système mais également de concevoir des outils pour augmenter cette résilience. Un résumé de ces mesures est disponible dans le tableau suivant.

Nom de la mesure	Informations nécessaires			Interprétation
	(AC, FM)	{FTM1,…}	Π	
RC	✓	✓	✗	Potentiel de cohérence d'un ensemble de mécanismes de tolérance aux fautes (peut-être ajusté avec des probabilités).
RE	✓	✓	✗	Pourcentage d'évènement auxquels le système est résilient lors d'un scénario donné.
MTTI	✓	✓	✓	Temps moyen avant l'apparition d'une incohérence.
MTRI	✓	✓	✓	Temps moyen pour retourner dans un état cohérent.
MTBI	✓	✓	✓	Temps moyen entre deux incohérences.

Cette thèse nous a également permis d'ouvrir des pistes de réflexion quant à l'utilisation de ces mesures dans les processus de développement actuels afin de permettre un jour de concevoir des systèmes critiques dont la résilience sera une des exigences.

Nous avons souligné dans nos travaux l'importance des hypothèses nécessaires à la mise en place des mécanismes de tolérance aux fautes. Le respect des hypothèses des mécanismes de sûreté et la confiance que nous avons en leur couverture sont essentiels comme cela a été démontré par D. Powell [44] en 1995. En ce qui concerne la résilience c'est fondamental. Lorsqu'un changement survient, il nécessaire de se poser la question de l'impact

de ce changement sur les hypothèses et sur la confiance que nous avons en ces hypothèses. Cela est déterminant pour améliorer la résilience des systèmes comme nous l'avons illustré dans cette thèse. Quelle que soit la nature du processus de développement, « classique » ou plus encore s'il est « agile », cette affirmation milite pour des méthodes de test qui examinent de façon approfondie la couverture des hypothèses et sur des techniques d'injection de fautes afin de s'assurer de la cohérence entre application et mécanismes de sûreté. La résilience d'un système dépend de cette phase de validation effectuée en amont, les méthodes agiles à la mode actuellement doivent y prêter une grande attention dès lors que l'on développe des systèmes critiques.

Comment améliorer la résilience des systèmes ? Tout repose sur notre capacité à anticiper. Anticiper les changements et donc les mécanismes de tolérance aux fautes développés hors-ligne afin de pouvoir les substituer en ligne grâce aux principes de la Tolérance aux Fautes Adaptative est sans doute une des clés du verrou scientifique et technique qu'est aujourd'hui la résilience. Cette approche permet de réduire le MTRI et donc de rendre les périodes d'incohérence négligeables au regard d'autres indicateurs de la sûreté de fonctionnement comme les taux de défaillance.

Anticiper c'est également être capable de mieux observer et comprendre les systèmes afin d'identifier les symptômes des changements. C'est donc un sujet complexe et ouvert qu'il nous faudra traiter pour pouvoir espérer un jour concevoir des systèmes résilients.

REFERENCES

[1] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. Dependable Secur. Comput. 1:11–33, IEEE Computer Society Press, Los Alami- tos, CA, USA, January 2004.

[2] Jean-Claude Laprie. From Dependability to Resilience. In International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, AK, USA, volume 8. 2008

[3] K. H. (Kane) Kim and Thomas F. Lawrence. Adaptive Fault Tolerance: Issues and Approaches. In Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems, pages 38–46. IEEE, 1990

[4] LYYRA, Antti K. et KOSKINEN, Kari M. With software updates, Tesla upends product life cycle in the car industry. *LSE Business Review*, 2017.

[5] UNIVAC conference, Charles Babbage Institute, University of Minnesota.

[6] Svoboda A.: From Mechanical Linkages to Electronic Computers: Recollections from Czechoslovakia, IEEE 1980.

[7] Peter A. Lee and Thomas Anderson. Fault Tolerance, volume 3 of Dependable Computing and Fault-Tolerant Systems. Springer-VerlagWien New York, Inc., 1990.

[8] Jean-Claude Laprie, A. Avižienis, and H. Kopetz, editors. Dependability: Basic Concepts and Terminology, volume 5 of Dependable Computing and Fault-Tolerant Systems. Springer-VerlagWien New York, Inc., 1992.

[9] Ashraf Armoush, Design Patterns for Safety-Critical Embedded Systems, Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

[10] Olivier Marin, Pierre Sens, Jean-Pierre Briot, and Zahia Guessoum. Towards Adaptive Fault-Tolerance for Distributed Multi-Agent Systems. In Proceedings of The Fourth European Research Seminar on Advances in Distributed Systems, ERSADS ’01, pages 195–201. 2001.

[11] Douglas McIlroy - Object-Oriented Programming - An Evolutionary Approach 1986. Isbn 0201103931

[12] Clemens Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[13] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. Computer, 36(1):41–50, IEEE, 2003.

[14] Roy Sterritt. Autonomic computing. Innovations in Systems and Software Engineering, 1(1):79–88, Springer, 2005.

[15] Abdeldjalil Boudjadar, Alexandre David, Jin Hyun Kim, Kim G. Larsen, Marius Mikučionis, Ulrik Nyman, Arne Skou, A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling, Science of Computer Programming, Volume 113, Part 3, 2015, Pages 236-260.

[16] Formal Methods for Real Time Systems: Automatic Verification & Validation, Kim G. Larsen. Presented at the ARTES summer school, August 1998.

[17] Huriaux, Christophe, Architecture FPGA améliorée et flot de conception pour une reconfiguration matérielle en ligne efficace, thèse de l'Université de Rennes 1

[18] Delta-4: A Generic Architecture for Dependable Distributed Computing

[19] Michaël Lauer, Matthieu Amy, Jean-Charles Fabre, Matthieu Roy, William Excoffon, et al.. Resilient Computing on ROS using Adaptive Fault Tolerance. *Journal of Software: Evolution and Process*, John Wiley & Sons, Ltd., 2017, 00, pp.1 - 18.

[20] Miruna Stoicescu, Jean-Charles Fabre, Matthieu Roy. Architecting resilient computing systems: A component-based approach for adaptive fault tolerance. *Journal of Systems Architecture*, Elsevier, 2017, 73, pp.6-16.

[21] Miruna Stoicescu. Architecting Resilient Computing Systems: a Component-Based Approach. Ubiquitous Computing. Institut National Polytechnique de Toulouse - INPT, 2013

[22] Laurent Saloff-Coste, *Lectures on finite Markov chains,* Lectures on Probability Theory and Statistics, 301-413. Lecture Notes in Math. n° **1665**. Springer (1997).

[23] Avizienis, A., Kopetz, H., & Laprie, J. C. (Eds.). (2012). *The Evolution of Fault-Tolerant Computing: In the Honor of William C. Carter* (Vol. 1). Springer Science & Business Media.

[24] Marin, O., Bertier, M., & Sens, P. (2003, November). DARX-a framework for the fault-tolerant support of agent software. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on* (pp. 406-416). IEEE.

[25] Fraga, J., & Siqueira, F. (2003, October). An adaptive fault-tolerant component model. In *null* (p. 179). IEEE.

[26] Lauer, M., Amy, M., Fabre, J. C., Roy, M., Excoffon, W., & Stoicescu, M. (2016, January). Engineering adaptive fault-tolerance mechanisms for resilient computing on ROS. In *High Assurance Systems Engineering (HASE), 2016 IEEE 17th International Symposium on* (pp. 94-101). IEEE.

[27] Edsger W. Dijkstra. On the role of scientific thought.In Selected Writings on Computing: A Personal Perspective, pages 60–66. Springer, 1982.

[28] Powell, D. (Ed.). (2012). *Delta-4: a generic architecture for dependable distributed computing* (Vol. 1). Springer Science & Business Media.

[29] International Organization for Standardization. (1994). *ISO 8402: 1994: Quality Management and Quality Assurance-Vocabulary*. International Organization for Standardization.

[30] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., ... & Ng, A. Y. (2009, May). ROS: an open-source Robot Operating System. In *ICRA workshop on open source software* (Vol. 3, No. 3.2, p. 5).

[31] Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., & Stefani, J. B. (2012). A component-based middleware platform for reconfigurable service-oriented architectures. *Software: Practice and Experience*, *42*(5), 559-583.

[32] United States Department of Defense (9 November 1949). MIL-P-1629 - Procedures for performing a failure mode effect and critical analysis. Department of Defense (US). MIL-P-1629.

[33] CEI 1985b Techniques d'analyse de la fiabilité des systèmes – Procédure d'Analyse des Modes de Défaillance et leurs Effets (AMDE), Rapport de normalisation np. 812 Commission Electronique International (CEI), 1985

[34] The Agile Manifesto by: K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mallor, Ken Shwaber, Jeff Sutherland (2001)

[35] Schwaber, K., & Beedle, M. (2002). *Agile software development with Scrum* (Vol. 1). Upper Saddle River: Prentice Hall.

[36] Beck, K. (1999). Embracing change with extreme programming. Computer, 32(10), 70-77.

[37] Levine, L., 2005, May, Carnegie Mellon Software Engineering Institute. Retrieved 2012, from Reflections on Software Agility and Agile Methods: Challenges, Dilemmas, and the Way Ahead

[38] ARVILLA, S. M. C. (2014). Risk Assessment in Project Planning Using Fmea and Critical Path Method. *Scientific Papers*, *14*(3), 39.

[39] Stålhane, T., Myklebust, T., & Hanssen, G. K. (2012). The application of Safe Scrum to IEC 61508 certifiable software. In 11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference (pp. 6052-6061).

[40] Kim, K. H., & Welch, H. O. (1989). Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE transactions on Computers*, *38*(5), 626-636.

[41] Chen, L. (1978). N-version programming: A fault-tolerant approach to reliability of software operation. In *Proc. International Symposium on Fault Tolerant Computing* (pp. 3-9).

[42] Laprie, J. C., Arlat, J., Beounes, C., & Kanoun, K. (1990). Hardware-and Software-Fault Tolerance. In *ESPRIT'90* (pp. 786-789). Springer, Dordrecht.

[43] Pintard, L. (2015). From safety analysis to experimental validation by fault injection-case of automotive embedded systems (Doctoral dissertation, INP Toulouse).

[44] Powell, D. (1995). Failure mode assumptions and assumption coverage. In *Predictably Dependable Computing Systems* (pp. 123-140). Springer, Berlin, Heidelberg.

ANNEXE 1 : Mes publications

I-Revue scientifique :

M.LAUER, M.AMY, J.C.FABRE, M.ROY, W.EXCOFFON, M.STOICESCU
Resilient Computing on ROS using Adaptive Fault Tolerance
Journal of Software: Evolution and Process, 18p., Mars 2018
TSF, EUMETSAT
Rapport LAAS N° : 17555
https://hal.laas.fr/hal-01703968

Soumission en cours:

W. EXCOFFON, J.C. FABRE, M. LAUER,
Resilient Computing by Adaptive Fault Tolerance: From Analysis to Simulation
2017/2018 Computing, Springer

II-Conférence internationale avec acte :

J.C.FABRE, M.LAUER, M.ROY, M.AMY, W.EXCOFFON, M.STOICESCU
Towards resilient computing on ROS for embedded applications
Embedded Real Time Software and Systems (ERTS²) 2016 du 27 janvier au 29 janvier 2016, Toulouse (France), Janvier 2016, 8p.
TSF, ESOC
Rapport LAAS N° : 16066
https://hal.archives-ouvertes.fr/hal-01288113

M.LAUER, M.AMY, J.C.FABRE, M.ROY, W.EXCOFFON, M.STOICESCU
Engineering adaptive fault-tolerance mechanisms for resilient computing on ROS
IEEE International Symposium on High Assurance Systems Engineering (HASE) 2016 du 07 janvier au 09 janvier 2016, Orlando (USA), Janvier 2016, pp.94-101
TSF, ESOC
Rapport LAAS N° : 16064
https://hal.archives-ouvertes.fr/hal-01288098

W.EXCOFFON, J.C.FABRE, M.LAUER
Towards modelling adaptive fault tolerance for resilient computing analysis
International Conference on Computer Safety, Reliability and Security (SafeComp) 2016 du 20 septembre au 23 septembre 2016, Trondheim (Norvège), Septembre 2016, 12p.
TSF
Rapport LAAS N° : 16564

W.EXCOFFON, J.C.FABRE, M.LAUER
How Resilient is your computer system?
Embedded Real Time Software and Systems (ERTS²) 2018 du 31 janvier au 02 février 2018, Toulouse (France), Février 2018, 8p.

TSF
Rapport LAAS N° : 18044
https://hal.archives-ouvertes.fr/hal-01708220

W.EXCOFFON, J.C.FABRE, M.LAUER
Analysis of Adaptive Fault Tolerance for Resilient Computing
European Dependable Computing Conference (EDCC) 2017 du 04 septembre au 08 septembre 2017, Genève (Suisse), Septembre 2017, 9p.
TSF
Rapport LAAS N° : 17569
https://hal.archives-ouvertes.fr/hal-01708205

III-Workshop international avec acte :

M.LAUER, M.AMY, W.EXCOFFON, M.ROY, M.STOICESCU
Towards adaptive fault tolerance: from a component-based approach to ROS
CARS Workshop. Critical Automotive applications: Robustness & Safety 2015 du 08 septembre au 08 septembre 2015, Paris (France), Septembre 2015, 4p.
TSF, ESOC
Rapport LAAS N° : 15665
https://hal.archives-ouvertes.fr/hal-01193039

W.EXCOFFON, J.C.FABRE, M.LAUER
An approach for resilient systems analysis
Safecomp FastAbstract 2016 du 20 septembre au 23 septembre 2016, Trondheim (Norvège), Septembre 2016, 2p.
TSF
Rapport LAAS N° : 16616
https://hal.laas.fr/hal-01370228

ANNEXE 2 : Guide d'utilisation du simulateur

Le logiciel est composé de deux fonctionnalités principales. La première est une analyse capable de donner le Ratio de Couverture d'un Composant, la seconde permet de simuler le cycle de vie d'un composant à travers l'exécution d'un scénario.

I-Généralités sur le logiciel

Les deux fonctionnalités présentent des similarités dans l'implémentation du modèle. Dans cette partie nous discuterons des détails d'implémentations.

A-Des composants fonctionnels (applications)

Dans la modélisation les composants sont représentables par un couple de n-uplets. Le premier représente les caractéristiques applicatives, le deuxième le modèle de faute. Dans ce logiciel les composants sont stockés dans des tableaux d'entier. Un ensemble de caractéristiques applicatives et un modèle de faute est appelé un profil de composant.

Dans la fonctionnalité simulateur il n'est pas nécessaire d'organiser de façon particulière les paramètres du tableau pour peu que l'ordre soit identique à celui utilisé pour l'implémentation des FTM comme décrit dans la section suivante.

Le logiciel permet deux solutions pour importer le modèle des composants. Dans les deux cas il faut indiquer le nombre de caractéristiques applicatives et le nombre de type de fautes. Puis on peut choisir d'importer depuis un fichier les valeurs max de ces paramètres et depuis un second fichier les valeurs min. Il est aussi possible de renseigner les valeurs min et max de chaque paramètre « à la main ».

B-Des FTMs

L'implémentation des FTMs est différente selon les fonctionnalités. Dans le cadre de la C-simulation à l'aide de scénario les FTMs sont implémenté de la même manière que les composants fonctionnels c'est-à-dire des tableaux d'entier. Chaque tableau contient les caractéristiques applicatives requises par le FTM puis le modèle de faute que tolère le FTM.

La fonctionnalité calcul du ratio de couverture utilise une implémentation différente dans La fonction générale d'initialisation de la configuration initconfig() permet de charger le modèle utilisé lors des phases de mesures. Cette fonction renvois les paramètres généraux suivant :

-nbac : le nombre de caractéristiques applicatives

-nbfm : le nombre de type de fautes

-Compstruct : Est un tableau contenant les valeurs maximales des caractéristiques applicatives et des types de fautes.

-CompStructMin : Est un tableau contenant les valeurs minimales des caractéristiques applicatives et des types de fautes.

C-Implémentation

La fonction générale d'initialisation de la configuration initconfig() permet de charger le modèle utilisé lors des phases de mesures. Le chargement du modèle se fait soir manuellement soit via des fichiers vmax.txt et vmin.txt. Cette fonction renvoi les paramètres généraux suivants :

-nbac : le nombre de caractéristiques applicatives

-nbfm : le nombre de type de fautes

-Compstruct : Est un tableau contenant les valeurs maximales des caractéristiques applicatives et des types de fautes.

-CompStructMin : Est un tableau contenant les valeurs minimales des caractéristiques applicatives et des types de fautes.

La fonction importComps() permet de charger un ensemble de profils de composants initiaux. Elle retourne une liste Comps dont chaque élément est un profil de composant initial.

La fonction importFTMs() charge le set de FTMs. La fonction propose de charger le set directement depuis un fichier ftms.txt ou de rentrer manuellement chacun des FTMs. Cette fonction retourne la liste FTMs dont chaque élément est un n-uplet correspondant au modèle d'un FTM.

D-Structures des fichiers

Les fichiers utilisés pour l'importation des valeurs min/max, des profils initiaux et de l'ensemble de FTMs ont tous la même structure. L'ordre des caractéristiques applicatives et des types de fautes doit être identique dans tous les fichiers.

Les fichiers vmin.txt et vmax.txt sont composés d'une seule ligne. Dans l'ordre sont notées les valeurs des caractéristiques applicatives séparées par un espace puis les valeurs des types de fautes. Par exemple pour un modèle à 4 caractéristiques applicatives et 3 types de fautes dont toutes les valeurs admissibles sont comprises entre 0 et 1 :

- vmin.txt contiendra l'unique ligne : 0 0 0 0 0 0 0
- vmax.txt contiendra l'unique ligne : 1 1 1 1 1 1 1

Le fichier comps.txt sera composé d'autant de ligne qu'il y a de composants à utiliser dans la fonctionnalité de simulation. La structure de ces lignes est strictement équivalente à celle décrite pour les fichiers précédents. Les valeurs renseignées sont celle du profil initial de chaque composant.

Le fichier ftms.txt contient autant de ligne que de FTMs, chaque ligne correspond à la modélisation du FTMs. La structure de ces lignes est donc identique à celles précédemment décrites.

II-Simulation et mesure de la résilience

A-Evènement et scénario

On définit ici un événement comme la modification d'une caractéristique applicative ou d'un type de fautes. Un événement peut donc être représenté par 3 données :

-Le composant affecté

-La caractéristique applicative ou le type de faute affecté

-La nouvelle valeur affectée.

On définit par ailleurs un scénario comme une suite d'événements. Le programme ne permet pas à l'heure actuelle de spécifier un scénario. Celui-ci est générer de manière aléatoire comme nous le constaterons dans la partie suivante.

B-Génération et application de scénario

Le programme prend en entrée un nombre d'évènements noté nbev. La création du scénario consiste à générer aléatoirement ces événements. Pour ce faire on tire aléatoirement (bibliothèque python random) un premier nombre compris entre 1 et nbcomp, cette dernière correspond au nombre de profils contenus dans le fichier comps.txt.

On tire ensuite un nombre compris entre 1 et nbac+nbfm, afin de déterminer la variable qui sera affectée. Si le nombre tombe dans [1 ;nbac] alors c'est une caractéristique applicative qui sera affectée, dans le cas où le nombre tiré est dans [nbac;nbfm] c'est un type de faute qui sera modifié.

Ensuite, on tire un troisième nombre qui sera la nouvelle valeur. Celle-ci sera comprise entre les valeurs minimale et maximale comprise dans les fichiers vmin.txt et vmax.txt. En cas d'impossibilité de trouver une valeur différente on relance un tirage d'événements complet. Cette fonctionnalité est implémenté dans la fonction scenarioGen().

Lorsque la liste d'évènement (i.e. le scénario) est complète, on génère l'a liste des profils successifs des composants. Supposons que chaque évènement ei correspond à un temps ti. On génère la liste dont l'élément i correspond à l'ensemble des profils des composants après avoir subi les i précédents évènements. Cette liste que l'on appelle syststates est renvoyé par la fonction statesGen().

C-Solutions et calcul de résilience.

Pour chaque élément de la liste syststates précédemment créée, on cherche pour chaque profil de composant s'il existe au moins un FTM cohérent. La fonction solGen() permet ainsi de créer la liste solution des FTMs solutions. Chaque élément i de cette liste correspond à la liste des FTMs solutions pour les profils de composants après i évènements.

Pour qu'un profil de composant soit cohérent avec un FTM il faut que chaque élément du n-uplet représentant le composant soir inférieur ou égal à la variable correspondante du n-uplet représentant le FTMs et contenue dans le fichier FTMs.txt.

On utilise à présent cette liste pour mesurer la résilience du système. On comptabilise le nombre d'évènements pour lequel il existe au moins un profil de composant tel que nous n'avons pas de FTMs solution. Ce nombre est ramené à un pourcentage sur le nombre d'évènement total. C'est cette mesure que nous appelons la résilience du système. La fonction en charge de ce calcul est la fonction resilience().

On notera qu'une analyse de sensibilité peut être conduite en forçant les valeurs des variables des n-uplets représentant les composants à partir des valeurs min et max.

III- Calcul du Ratio de Couverture

L'objectif de cette fonctionnalité est de mesurer le ratio de couvrabilité d'un composant. Cette partie inclus des outils d'évaluations d'un set de FTM et de tous les sous-sets associés. Cette fonctionnalité n'utilise pas de profil initial pour les composants. Le calcul s'effectue pour un seul composant.

A-Fonction Ratio

Afin de mesurer le ratio de couvrabilité d'un composant on parcourt l'ensemble des profils de composants possibles. Pour chaque profil on vérifie l'existence d'au moins un FTM cohérent. Si un tel FTM existe alors le profil est couvrable.

L'ensemble des profils possibles est défini par les valeurs max et min définies lors de l'importation du modèle via la fonction initConf(). On retire également les cas triviaux pour lesquels aucun type de faute n'est pris en compte.

La valeur du ratio se définit comme le pourcentage de profil couvrables divisés parmi le nombre de profils possibles du composant.

Il est à noter que dans ce mode de fonctionnement le programme utilise une autre implémentation des FTMs. En effet, chaque FTMs est ici une fonction qui prend en argument un profil de composant et qui renvoi TRUE si le composant et le FTM sont cohérents et FALSE sinon.

B-Analyse de sensibilité sur les ensembles de FTMs.

Puisqu'en python tout est objet on peut stocker la liste des fonctions correspondant au FTMs. On peut dès lors faire un parcours exhaustif de tous les sous-ensembles qui peuvent être généré à partir de l'ensemble initial de FTMs.

Pour chacun de ces sous-ensembles on calcul la couvrabilité du composant. Toutes les mesures ainsi obtenues sont exportées dans un fichier out.csv. Cette analyse de sensibilité est effectuée par la fonction sensFTM().

IV-Limites

Comme nous avons pu le voir les FTMs sont implémentés de deux manières différentes. Dans le cadre de l'implémentation sous forme de n-uplets on a :

-Une plus grande capacité d'adaptation aux variations du modèle puisque tout est stocké dans les fichiers textes facilement modifiables.

-Une incapacité à gérer le fait que la cohérence puisse dépendre d'hypothèses qui sont une combinaison de plusieurs caractéristiques applicative. Par exemple, l'hypothèse « *Si le composant à un état ST celui-ci doit être accessible* »

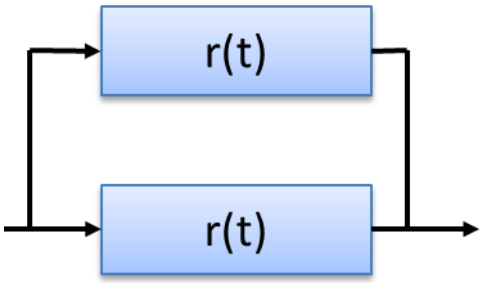
A l'inverse, lorsque l'on utilise des fonctions booléennes pour modéliser les FTMs on a :

-Une définition « en dur » des FTMs dans le code

-La possibilité d'exprimer des hypothèses de cohérence complexes.

ANNEXE 3 : Calcul du taux de défaillance instantané

Dans le cas d'une mise en parallèle de deux applications dont on note la fiabilité r(t) comme décrit sur la figure suivante :



On suppose que les lois de fiabilité sont exponentielles telles que :

$$r(t) = exp(-\lambda_c t)$$

La fiabilité globale du système R(t) peut se calculée comme suit :

$$\mathrm{R}(t) = 1 - (1 - exp(-\lambda_c t))^2$$

$$\mathrm{R}(t) = 2\,exp(-\lambda_c t) - exp(-2\lambda_c t)$$

D'où le calcul du taux de défaillance instantanée noté également □(t) :

$$\lambda(t) = \frac{-1}{R(t)} \frac{dR(t)}{dt}$$

$$\lambda(t) = 2\lambda_c \frac{\exp(-2\lambda_c t) - exp(-\lambda_c t)}{2\,exp(-\lambda_c t) - exp(-2\lambda_c t)}$$

$$\lambda(t) = 2\lambda_c \frac{1 - \exp(-\lambda_c t)}{2 - exp(-2\lambda_c t)}$$

Et ainsi on peut calculer la valeur limite de □(t) à l'infini :

$$\lim_{t \to +\infty} \lambda(\mathrm{t}) = \lambda_c$$