REPUBLIQUE ALGERIENNE DEMOCRATIOQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

Université des Sciences et de la Technologie Houari Boumedien

Faculté d'Electronique et Informatique

Laboratoire des systèmes informatiques

Option génie logiciel

Thèse pour l'obtention du diplôme
de magistère en informatique

Test des systèmes logiciels : proposition d'une architecture de test pour les systèmes réactifs temps réel

Présenté par : Ahmed BERBAR

Soutenu le : 13 juillet 2003

Devant le jury composé de :

Présidente du jury : M^me^ Z.ALIMAZIGHI maître de conférence.
Directeur de thèse : M^r^ M.AHMED NACER Maître de conférence.
Membre du jury : M^r^ N.BADACHE Maître de conférence.
Membre du jury : M^r^ A.BELKHIR Docteur d'état.

2003

RESUME

Ce travail porte sur les tests des systèmes logiciels réactifs temps réel. Dans ce document nous avons d'abord défini les systèmes réactifs temps réel tout en présentant les méthodes de test existantes, comme la méthode proposée est basée sur l'approche d'Algar et Zheng une partie de ce document est consacrée à la présentation de cette approche. Nous présentons ensuite notre approche de test, accompagnée par une architecture de test, complétée elle-même par un model de faute.

Un système réactif temps réel est un système qui interagit continuellement avec son environnement. Deux propriétés importantes caractérisent ces systèmes : la synchronisation des stimulants et la synchronisation des réponses. Comme exemples de ces systèmes le système de monitoring d'un patient ou un chien de garde d'une centrale nucléaire.

Plusieurs méthodes ont été développées pour tester ce type de système dont l'approche d'Algar & Zheng qui a suscité notre intérêt car elle se base sur le formalisme TROM qui est l'un des rares formalismes capables de décrire l'interaction entre les objets sans faire abstraction des contraintes temporelles.

Notre solution propose une extension du modele d'Algar et Zheng permettant de tester l'interaction entre les objets d'une même classe. Pour cela, on a redéfini le formalisme TROM. La deuxième phase de notre solution consistait à proposer une architecture de test adaptée à l'approche proposée. Cette architecture lève certaines contraintes temporelles imposées par d'autres architectures. Ensuite on a complété cette architecture par un modele de fautes qui permettra de diagnostiquer certaines erreurs décelées lors de l'étape de test.

Dédicaces

A ma très chère mère El Aaradj Raghda dont je ne pourrais exprimer le mérite et le rôle qu'elle a joué dans mon éducation, mon instruction et ma réussite, que dieu lui accorde sa sainte miséricorde.

A mon père BERBAR Seradj edine qui a su prendre le relais, et que j'ai trouvé à mes cotés à tout moment pour me prodiguer soutien, conseils et encouragements et dont le rôle est inestimable.

A mes très chers frères Ryad et Aimen.

A mon adorable petite sœur Inès.

A ma famille.

Je dédie ce travail.

Remerciements

Ce mémoire n'aurait pas vu le jour sans l'aide précieuse de mon promoteur
Monsieur Ahmed nacer, maître de conférence à l 'U.S.T.H.B, qui m'a fait bénéficier de ses conseils, ses commentaires et ses orientations efficaces pour arriver à ces résultats. Qu'il en soit remercié.

Je remercie madame Alimazighi pour l'honneur qu'elle m'accorde en présidant mon jury, je remercie également les membres du jury messieurs Ahmed Nacer, Badache et Belkhir

Je remercie également tous mes professeurs qui m'ont éclairés avec leur savoir.

Enfin mes remerciements s'adressent à tous ceux qui m'ont aidé de prés ou de loin, à la réalisation de cette thèse.

INTRODUCTION GENERALE

C'est en 1968 durant une conférence organisée par L'OTAN que s'est cristallisé un certain nombre de problèmes posés par la production de logiciels. C'est cette dernière conférence qui proposa le terme de "Génie Logiciel" pour regrouper l'ensemble de techniques et méthodes visant à améliorer cette production.

Les scientifiques réunis à cette occasion formulèrent un certain nombre de questions qui traduisaient les inquiétudes du moment :

- Pourquoi ne peut-on pas détecter toutes les erreurs d'un logiciel avant de remettre celui-ci au client ?
- Pourquoi y a-t-il un tel écart entre le coût estimé du logiciel et son coût réel ?
- Pourquoi a-t-on tant de difficultés à estimer la progression du développement d'un logiciel ?

Certaines constatations datant de cette conférence sont toujours d'actualité. Ainsi Djikstra remarqua que «la possibilité de convaincre les utilisateurs, où soi même, que le logiciel est bon est fortement dépendant du processus de conception utilisé » et proposa la création d'usine de composantes réutilisables. La pertinence actuelle des questions posées en 1968 montre s'il en était besoin qu'il reste encore un long chemin à parcourir pour industrialiser la production des logiciels.

Ainsi, le génie logiciel permet de traiter quatre propriétés suffisamment générales pour obtenir un degré de satisfaction conforme aux objectifs :

- L'**efficacité** qui est la capacité d'un logiciel à utiliser des ressources en temps et en espace de façon optimale.
- La **fiabilité** qui est la confiance que l'on peut accorder au bon fonctionnement d'un logiciel.
- La **modifiabilité** qui est la possibilité d'introduire des changements sans augmenter la complexité du système original.
- L'**intelligibilité** qui est la facilité d'utilisation d'un logiciel.

Dans cette thèse on ne s'intéressera qu'à un seul des aspects du génie logiciel qui est la fiabilité.

L'étude de fiabilité des logiciels a pour but de cerner les différentes caractéristiques et techniques permettant d'estimer la qualité des logiciels.

En effet, qualité et fiabilité sont intimement liées. Cependant, il est important de bien différencier ces deux notions que l'on a souvent tendance à confondre :

Un logiciel de qualité est un logiciel qui répond à ses spécifications : ce que l'on attend qu'il fasse.

Un logiciel tend à être fiable lorsqu'il reste de qualité malgré certaines erreurs ou pannes détectées lors de son exécution

.

La fiabilité d'un logiciel est une fonction du nombre des défaillances par les utilisateurs, il est clair que la fiabilité est déterminante pour l'utilisateur final ainsi que pour le client. Selon la définition ISO, un système est fiable s'il maintient ``son niveau de service dans des conditions précises et pendant une période déterminée'' (ISO9126). Cette notion est intéressante dans la mesure où un produit logiciel est un instrument déterministe, c'est-à-dire qu'il existe des conditions dans lesquelles il sera toujours fiable. L'élément critique de la définition est *les conditions* dans lesquelles le logiciel sera utilisé. Ces conditions ne seront pas toujours très claires, et même si elles le sont pour le programmeur, elles ne le seront peut-être pas pour l'utilisateur [**13**].

La fiabilité est une condition d'erreur lorsque le logiciel est en exécution, ce n'est pas le synonyme d'une faute logicielle. Une faute est une erreur de programmation ou de spécification, elle est de nature statique. Une faute peut causer une défaillance si le code fautif est exécuté avec un ensemble d'entrées qui expose la vulnérabilité du logiciel. Donc une faute ne mène pas directement vers une défaillance. Les fautes qui influent sur la fiabilité n'affectent pas tous les utilisateurs de la même manière.

On considère le problème de fiabilité sous deux angles :

1 – Comment développe t'ont des logiciels fiables ?

2 – Comment savoir si un logiciel déjà développé est fiable ?

La première question trouve sa réponse dans quelques techniques utilisées pour la programmation de logiciels fiables. Tandis que la deuxième question utilise généralement un certain nombre de métriques alliées à certains tests statistiques pour mesurer la fiabilité d'un logiciel.

Ce document est structuré en cinq chapitres :

- Le premier présente les concepts liés à la notion de fiabilité.
- Le second présente les différentes techniques de tests.
- Le troisième introduira la notion de sûreté des logiciels.
- Le quatrième chapitre présentera les différentes méthodes utilisées pour assurer la fiabilité d'après les types de logiciels
- Tandis que le cinquième chapitre présentera notre contribution qui consiste en une approche de test pour les systèmes réactifs temps réel accompagnée d'une architecture de test et d'un model de faute.

CHAPITRE 1

ETUDE DE FIABILITE DES LOGICIELS

CHAPITRE 1 : ETUDE DE FIABILITE DES LOGICIELS

La fiabilité d'un logiciel n'est pas un concept absolu, elle dépend du point de vue de l'utilisation de ce logiciel. Si chaque utilisateur se sert d'un même programme de façon différente, des fautes qui auront un grand impact sur la fiabilité du logiciel pour un utilisateur donné ne se manifesteront jamais avec un autre mode de travail.

1.1 spécifications de la fiabilité.

Les besoins en matière de fiabilité doivent être exprimés lors de l'étape de spécification des besoins.

- Exprimer le niveau de fiabilité requis.
- Le plan de test du logiciel doit être décrit de manière à prévoir des tests de fiabilité.

La spécification de la fiabilité doit identifier les différents types de pannes et décider de leurs traitements séparés ou non, pour cela on utilise généralement les arbres de défaillances.

Arbres de défaillances

Cette méthode a pour objet de déterminer les diverses combinaisons possibles d'événements qui entraînent la réalisation d'un événement indésirable unique. L'arbre de défaillance est une analyse déductive dont la représentation graphique des combinaisons est réalisée par une structure arborescente.

Ces arbres de défaillance sont ensuite quantifiés afin de déterminer la probabilité d'occurrence de l'événement supérieur ; cet événement correspond généralement à un événement indésirable pour le système **[11]**.

.

Pour la plupart des systèmes concrets le degré de fiabilité varie d'un sous systèmes à un autres car il est difficile de spécifier la fiabilité totale du système et de réaliser le degré de fiabilité voulu. Donc il est préférable de spécifier les degrés de fiabilité en fonction des sous systèmes du logiciel.

Les entraves de la fiabilité **[24]**.

Les principales entraves de la fiabilité sont : Les défaillances, les erreurs et les fautes. Une défaillance du système survient lorsque le service délivré dévie de l'accomplissement de la fonction du système. Une erreur est la partie de l'état du système qui est susceptible d'entraîner une défaillance : une erreur affectant le service est une indication qu'une défaillance survient ou est survenue. La cause adjugée ou supposée d'une erreur est une faute. Sachant que la plupart des défaillances sont reliées par des relations causales, cela signifie que ce qui est effectivement défini de façon non pas absolue mais par rapport à une référence est la défaillance, les erreurs et les fautes étant simplement les causes. L'erreur est la

cause concernant les données qui sont traités. La faute est ce que l'on déclare être la cause ultime, soit une hypothèse soit un jugement, bien entendu cette cause soit disons ultime varie selon les points de vue, l'évolution du système etc…. Les fautes mènent à des erreurs lorsqu'elles sont activées. Par propagation elles doivent mener à des défaillances, puis à leur tour les conséquences des défaillances peuvent être des fautes pour d'autres systèmes interagissant avec le système considéré dans l'optique de causalité.

Coût des erreurs **[19].**

Il existe deux types de tests : les tests de pre-implementation et les tests de post-implémentation, le premier inclut les tests qui se déroulent avant que le système ne soit dans un état opérationnel. L'objectif de ce test est de vérifier si le système correspond aux spécifications et que les défauts du système ont été supprimés avant la mise en fonctionnement de ce dernier. Le second type de test intervient après que le système ne soit opérationnel ; Il est considéré comme faisant partie de la maintenance du système.

Le prix de la suppression de défauts du système avant que le système n'entre en activité inclue :

- La construction des défaillances dans le système.
- Identifier l'existence de défaut dans le système.
- Corriger ces défauts.
- Effectuer des tests pour voir si ces défauts ont été réellement supprimés.

Les défaut découverts après la mise en marche du système génèrent les coûts suivants :

- Spécifier et coder les défaut dans le système
- Détecter les problèmes dans le système d'application
- Reporter le problème au service d'information et/ou au client.
- Corriger les problèmes générés par le défaut
- Arrêter le système jusqu'à la découverte de l'origine du défaut et le corriger.
- Corriger le défaut.
- Tester le système pour voir si le problème n'existe pas encore.
- Introduire la version corrigée dans la production.

Le test doit inclure le coût du test plus le coût des erreurs non encore détectées, généralement le coût des tests n'est jamais connu à l'avance un test est normalement considéré comme étant le processus qui détecte les erreurs et qui assure que le système fonctionne correctement cependant le coût de correction et de construction des erreurs dépasse de loin le coût de détection de ces dernières.

Le bureau national des standards et normalisations a estimé que le coût des tests incluant la correction des défaillances non découvertes avants la mise en œuvre d'un logiciel dépasse la moitié de prix de développement.

Le grand coût des systèmes défectueux pose les deux challenges suivants

- Comment quantifier le vrai prix de la suppression des erreurs ?
- Comment réduire le prix des tests ?

Etapes de spécification **[16].**

De manière générale pour établir la spécification de la fiabilité d'un logiciel on suit les étapes suivantes :

1 - Pour chaque sous système identifié les types de défaillances qui sont susceptibles de se manifester.
2 - Classer les pannes à partir de leur analyse en classe. (Tableau A).
3 - Pour chaque classe de pannes identifiée, déterminer les besoins en matière de fiabilité par l'application des métriques appropriées (tableau B).

Exemple : tableau A : les différentes classes de pannes.

Classes de pannes	Description
Transitoire	Ne se produit qu'avec certaines entrées
Permanente	Se produit avec toutes les entrées
Recouvrable	Correction sans intervention de l'opérateur
Non recouvrable	Correction exigeant l'intervention de l'opérateur
Non corruptrice	Ne modifie pas les données ou l'état du système
Corruptrice	La panne corrompt les données

Tableau B spécification de la fiabilité par classes de panne.

Classe de panne	Exemple	Fiabilité
Permanente	On ne peut pas lire les pistes magnétiques	1 sur 100.000 transactions
Transitoire et non corruptrice	On ne peut pas lire les pistes magnétiques de certaines cartes	1 sur 10.000 transactions
Transitoire et corruptrice	Les cartes des banques étrangères corrompent les données	1 sur 20.000.000 transactions

1.2 Les métriques de fiabilité.

Les métriques de la fiabilité du logiciel utilisent les même principes que les métriques de la fiabilité du matériel. Cependant les pannes des composants logiciel contrairement aux pannes des composants matériels sont transitoires.

On utilise généralement les métriques suivantes pour estimer la fiabilité d'un logiciel **[WON00]**.

a- POFOD (probabilité de défaillance sur demande).

Mesure la vraisemblance de la défaillance du système lorsqu'une requête est réalisée, cette requête est utilisée dans les systèmes critiques(de contrôle) en fonctionnement constant. Par exemple : 0.001 POFOD signifie qu'une requête sur 1000 provoquera une défaillance.

b- ROCOF (intensité de défaillance)

Mesure la fréquence à laquelle un comportement erratique est probable, cette métrique est utilisée dans les S.E et les systèmes transactionnels. Par exemple ROCOF = 2/100 signifie que deux défaillances sont probables pour chaque unité de temps opérationnels

c- MTTF (Le temps moyen entre deux pannes)

C'est la mesure du temps qui sépare les pannes. Cette métrique est très utile lorsqu'on désire estimer la fiabilité d'un logiciel stable(que l'on ne modifiera pas) et les systèmes embarqués. Par exemple : MTTF = 500 signifie qu'une défaillance est à prévoir à chaque 500 unités de temps.

d- AVAIL (la disponibilité).

C'est la probabilité pour laquelle le logiciel soit disponible pour utilisation. Cette métrique est très appropriée pour les systèmes de télécommunication où le temps de réparation et le temps de redémarrage sont très significatifs et la gestion des appels, le routage des connexions (ou transactions). Par exemple AVAIL = 0.98 signifie qu'à chaque 1000 unités de temps, le logiciel est opérationnel pendant 980 unités de temps.

Nous pouvons effectuer trois types de mesures pour établir ces métriques :

- Le nombre de défaillances système pour un ensemble de données d 'entrée : sert à établir la métrique POFOD
- Le temps ou nombre de transactions entre défaillances : sert à établir la métrique ROCOF et MTTF.
- Pour un système en fonctionnement continu le temps de réparation ou de redémarrage après une défaillance sert à établir la métrique AVAIL.

Remarque.

Il n'y a pas une seule métrique universelle. Le choix de la métrique dépend de la nature de l'application, donc on peut établir les métriques en apportant des mesures appropriées.

On augment la fiabilité d'un logiciel en optimisant l'ensemble des métriques applicables.

Généralement les métriques sont définies dans le cahier de charge, elles sont prisent en compte dans la conception pour mesurer lors de la vérification.

En gros on peut dire qu'une métrique sert à mesurer la fiabilité d'un logiciel. Mais cette mesure n'est pas suffisante pour dire qu'un logiciel est fiable pour cela d'autres méthodes existent pour valider la fiabilité d'un logiciel.

1.3 Comment valider la fiabilité du système ?

La validation de la fiabilité d'un système est une opération très difficile et très coûteuse, mais il existe plusieurs approches qui peuvent consister en **[24]** :

- Prévenir les fautes, à savoir empêcher l'occurrence ou l'introduction d'une faute. Ceci est concrétisé par des méthodes de programmation fiables dont on détaillera certaines dans les paragraphes suivants.
- Tolérer les fautes[1], à savoir assurer la délivrance d'un service en dépit des fautes.
- Les éliminer, à savoir en réduire la présence (nombre, sévérité) au minimum et ceci grâce à des méthodes de test
- Les prévoir, sous l'angle de leur présence, de leur création et de leurs conséquences puisque de toute façon il restera des fautes et que d'autres surviendront au cour de la vie opérationnelle

On distingue deux types de tests : les simulations. Et les tests statiques.

A) La simulation.

Nous pouvons vérifier le taux de fiabilité d'un système par simulation mais c'est une méthode très coûteuse et surtout extrêmement longue un aperçu de cette méthode est présenté dans le paragraphe 2.2.2 **[16]**.

B) Le test statique **[16]**.

L'objectif de ce test est de déterminer la fiabilité du logiciel plutôt que de mettre les fautes en évidence. Le test est composé des étapes suivantes :

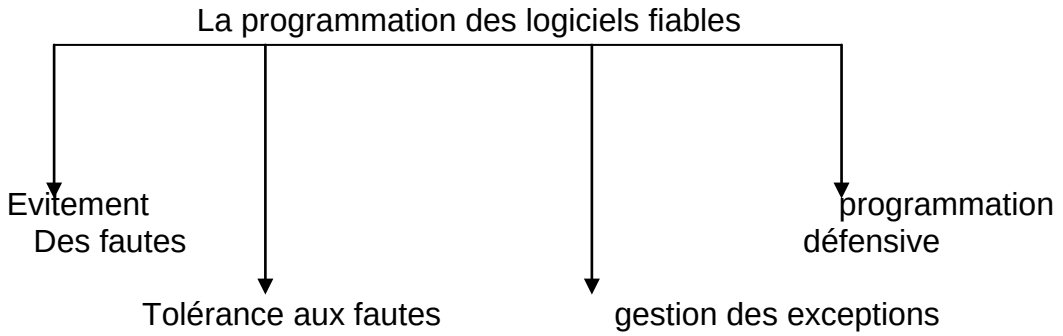
- Déterminer la configuration opérationnelle du logiciel : c'est à dire les différentes classes d'entrées du logiciel et leurs probabilités.
- Sélectionner un ensemble de données de test en fonction de cette configuration.
- Appliquer ces tests au programme en enregistrant le temps d'exécution entre chaque panne.
- Déterminer la fiabilité du logiciel après observation d'un nombre de pannes statistiquement significatif.

Ce type de test n'est pas toujours facile à réaliser. En effet, il faut d'abord déterminer la configuration opérationnelle la plus appropriée et générer un ensemble de données de test suffisamment grand pour pouvoir effectuer des statistiques (assez coûteuse).

[1] Voir chapitre page

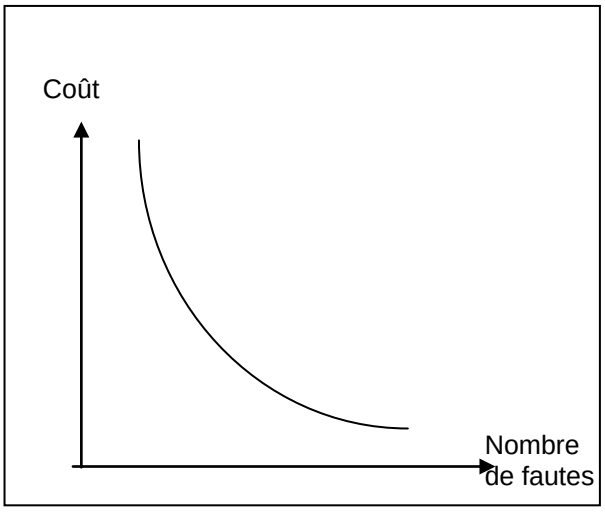
Les deux méthodes qu'on vient de voir sont utilisées pour valider des systèmes déjà confectionnés, cependant la fiabilité d'un logiciel passe nécessairement par l'élimination des fautes lors de la programmation pour cela il existe un certain nombre de méthodes à suivre durant la phase de développement du logiciel pour s'assurer de la fiabilité de ce dernier. Dans ce qui suit on essaiera de donner un aperçu de ces méthodes.

Méthodes de programmation de logiciels fiables.



a/ évitement des fautes [16].

Le développement d'un logiciel sans fautes coûte très cher le coût de détection d'une faute augmente exponentiellement.



Conséquence : Il est plus économique de payer les conséquences des pannes du système qui découlent de fautes résiduelles que de tenter de détecter ces fautes et de les corriger.

Le développement de logiciels sans fautes alliés au concept d'évitement de fautes se base sur les caractéristiques suivantes :

- Spécification précise.
- Approche de conception basée sur la dissimulation et l'encapsulation.

- Multiples revues.
- Planification de tests.

Spécification précise :

La précision des spécifications à travers une démarche rationnelle de programmation et une méthodologie prouvée de conception de logiciel permettra d'éviter des fautes de programmation qui risqueraient d'être coûteuse par la suite.

Par exemple : la programmation structurée est très utile parce qu'elle force les programmeurs à être prudent lors de l'écriture de leurs programmes
Elimination des GO TO

De plus certaines constructions ont plus de chances d'introduire des fautes graves dans des systèmes où la sécurité est critique :

- Les pointeurs (dangereux lorsqu'ils sont mal utilisés)
- La récursivité (des erreurs de récursivité : allocation de la totalité de la mémoire)
- Les interruptions (interruption d'une opération critique)

Dissimulation et encapsulation

Les concepts de dissimulation et d'encapsulation des données permettent de contrôler l'accès aux données du système.

Le concept d'objet implique l'encapsulation des données avec leurs méthodes (dans une même classe) assure que ces données ne peuvent être accéder qu'à travers ces méthodes.

Revue et planification de test.

L'utilisation intensive de revues lors d'un processus de développement de logiciel permet de corriger d'éventuelles erreurs et de procéder à une validation rapide du logiciel. Ceci permet également d'intégrer de nouveaux besoins au fur et à mesure du développement.

La planification des tests est une étape importante dans la mesure ou elle permet de détecter des fautes qui n'auraient pas étaient découvertes lors du processus de revue.

b/ Tolérance aux fautes [16].

La tolérance aux fautes nécessite de procéder aux phases de détection des pannes, d'estimation des dégâts, de recouvrement des fautes et de réparation

Les techniques utilisées pour la tolérance aux fautes des logiciels correspondent généralement à celles utilisées pour la tolérance aux fautes matérielles, notamment l'utilisation de la duplication.

Il existe deux approches possibles : le versionnement et le recouvrement.

- Le versionnement : différentes versions d'un même logiciel sont implémentées et exécutées en parallèle. La comparaison des résultats permet de choisir la supposée meilleure, cette approche est généralement la plus utilisée.
- Les blocs de recouvrement : c'est un test spécifique à chaque unité de programmation, il est intégré dans celle ci, l'exécution de ces unités étant séquentielle

c/ gestion des exceptions [16].

Une exception correspond à un événement inattendu qui survient en cour d'exécution, on distingue les exceptions matérielles et les exceptions logicielles. Généralement on remarque l'absence de mécanismes de gestion des exceptions dans la plupart des langages de programmation ce qui oblige le programmeur de les gérer par des structures de contrôle permises par le langage. Par contre certains SGBDOO actuels offrent cette possibilité par l'utilisation de mécanismes de triggers.

d/ La programmation défensive.

La programmation défensive est une approche ou le programmeur suppose qu'il peut y avoir dans le code des erreurs non détectées. On ajoute alors du code qui vérifie l'état du système après chaque modification et s'assure que le changement d'état est consistant. La programmation défensive regroupe la détection des pannes et le recouvrement des fautes **[16]**

- La détection des pannes.

Généralement une panne est provoquée par une corruption de l'état du système. Les approche suivantes sont généralement utilisées :

Traitement des exceptions.

Généralement cette approche détecte l'erreur mais ne la localise pas, de plus on ne peut pas effectuer toutes les vérifications.

Utilisation des assertions.

Dans cette approche on applique des prédicats sur des variables d'états du système. De manière pratique on vérifie l'état de chaque prédicat avant toute affectation.

Troisième approche.

Afin de limiter l'encombrement mémoire et ne pas ralentir l'exécution d'un système, on ne déclenche la vérification de l'état de celui-ci qu'avant certaines opérations critiques. Dans ce cas, si l'état n'est pas valide, on procède à l'estimation des dégâts et au recouvrement des fautes.

- Détection et recouvrement des fautes.

L'évaluation des dégâts est difficile il faut alors la minimiser par la détection des fautes avant la modification du système.

Principe.

Lorsque l'on a détecté une faute, on annule le changement d'état de manière à ne pas causer des dégâts. Cependant, il est fortement conseillé d'effectuer une estimation pour déterminer l'origine de la faute. La question qui se pose est la suivante :

Le viol d'assertion a t'il été provoqué par une instruction ou par un changement d'état antérieur valide mais logiquement incorrect ?

Quatrième approche.

C'est le cas d'utilisation de systèmes d'une limite temporelle (temps déterminé) ce qui exige l'utilisation d'une horloge ou contrôleur de délai (watchdog). Ceci est très utile en cas de problème d'exécution d'un processus. En effet, un processus est tenu de désactiver l'horloge à la fin de son exécution. Si, pour une raison quelconque, un processus ne se termine pas avant que l'horloge indique une certaine heure, alors celle-ci indique au système que le processus a dépassé le temps qui lui était imparti.

Toutes ces méthodes et approches tendent à créer des logiciels fiables, mais comme on l'a vu précédemment le concept de fiabilité n'est pas absolu pour cela les différents organismes internationaux ont élaborer certains standard et critères qu'il faudra respecter et suivre pour valider un logiciel.

1.4 - Norme et critères de fiabilité [8].

Afin d'assurer la qualité des produits logiciels, et particulièrement la qualité du développement, certains organismes internationaux cherchent à établir des critères concernant les démarches de production et la capacité d'une organisation à développer des produits fiables. L'existence de normes est un indicateur de la maturité d'une discipline.

La série des normes ISO9000 (1987) est une référence internationale en matière d'exigence pour l'assurance de la qualité. Les logiciels sont considérés comme une catégorie particulière de produits et relèvent d'un document particulier (ISO9000-3 1991) (lignes directrices pour l'application d'ISO9001 au développement à la mise à disposition et à la maintenance d'un logiciel).

Le capability maturity model CMM développé initialement à la demande du département américain de la défense et également une référence qui permet de

juger et d'auto-évaluer les processus de réalisation de logiciels. Le CMM propose cinq niveaux de maturité dépendant de la maîtrise du déroulement des projets et de la capacité de l'entreprise à s'améliorer.

Divers critères d'évaluation de la sécurité des systèmes informatiques ont été élaborés par plusieurs communauté (fédéral criteria aux Etats-Unis, CTCPEC au Canada (1993), ITSEC en Europe 1991) les common criteria (1994) constituent une tentative de regroupement de ces différentes propositions.

Certains donneurs d'ordres notamment dans les applications à caractère critique, imposent leurs propres démarches de production de logiciel. C'est par exemple le cas du ministère britannique de la défense. Qui impose ses propres standards à ces contractants, c'est t le premier standard ou l'utilisation d'une méthode formelle est exigées pour la spécification de la validation

1.5 - Conclusion

Durant la conférence de kitchenham en 1986 un nombre important de personne a défini la fiabilité comme étant l'absence d'erreurs. Malheureusement ce genre de définition n'est pas tous à fait juste pour cela les chercheurs en génie logiciels définissent la fiabilité comme étant une implémentation correcte des spécifications. Pareil définitions peuvent être utilisées durant la phase de développement mais elles ne sont pas adéquates pour la comparaison entre deux produits logiciels. Pour cela la norme ISO8492 1986 a défini la qualité comme étant la totalité des métriques ou caractéristiques d'un produit ou service qui supporte dans ses aptitudes les états et ou les besoins impliqués. Dans un environnement contractuel les besoins sont spécifies alors que dans d'autres environnements il sera nécessaire d'identifier et de définir les besoins impliqués **[7].**

Cependant la qualité du logiciel est généralement définie comme étant le fitnesse d'un produit pour sa construction, cependant plusieurs personnes peuvent avoir différentes constructions pour le même logiciel. Un nouvel utilisateur novice est probablement plus concerné par la facilité d'apprendre et par la robustesse que par l'efficacité. Les intégrateurs de système qui planifie d'incorporé les logiciels dans de plus grands systèmes doivent être plus concernés par la détection et le recouvrement d'erreurs que par la facilité de l'installation initiale. Tandis que les organisations de maintenance sont concernées par la documentation interne la suffisance des échafaudages (tests de surcharges, génération des tests), que par ce qui concerne l'utilisation directe par l'utilisateur. Ceci montre que la qualité d'un logiciel n'est pas une notion absolue mais qu'elle dépend du point de perception de celui qui veut évaluer la qualité en outre la qualité logiciel est multiface et l'importance des différentes faces change d'après le contexte même pour la même personne sur diffèrent points de l'axe du temps. C'est pour cela que différents types de tests ont été élaborés afin de permettre le test du logiciel sous différents angles et de façons diverses. Chapitre suivant définit la notion de test et présente les différentes approches de tests existantes.

CHAPITRE 2

MISE AU POINT ET TESTS.

CHAPITRE 2 : MISE AU POINT ET TESTS.

2.1 les différents concepts.

2.1.1 - Quelques définitions.

«La mise au point ne peut montrer l'absence d'erreurs ; elle ne peut montrer que l'existence d'erreurs » (Djikstra)

Tester : exécuter un programme dans le but de trouver des erreurs.

Le test de conformité : [6]

Le test est une activité permanente dans le développement matériel ou logiciel. Plusieurs aspects peuvent être étudiés : la performance, robustesse, la conformité (entre spécification et implantation). Le test a pour objectif de détecter les dysfonctionnements d'une implantation par rapport à la spécification. Le test de conformité se décompose en deux parties :

1- *La génération des cas de test :* elle consiste à extraire de la spécification des cas de test (des séquences d'actions) dont le but est de vérifier certaines propriétés sur l'implantation cette génération peut être exhaustive ou partielle. Pour tester une partie spécifique du système, le cas de test associé est composé de trois parties : le préambule (suite d'actions partant de l'état initial du système), le test de la partie et le postambule (suite d'action qui permet de revenir à l'état initial).
2- *L'exécution des cas de test :* elle consiste à expérimenter les cas de test générés sur l'implantation et observer les réactions de l'implantation. Le résultat d'un cas de test peut être :

Succès : le test a été réalisé avec succès.
Echec : le test a échoué.
Non Conclusif : aucun verdict ne peut être donné.

Debugger : une erreur ayant été constaté, trouver les causes et les corriger.
Bon cas de test : test qui a une forte probabilité de trouver une erreur non encore détectée.
Test couronné de succès : test qui découvre une erreur non encore détectée.
Un test qui ne découvre pas d'erreur n'est pas un bon test.

Pour un programme de taille réaliste, l'état de l'art ne permet pas de détecter toutes les erreurs avant la mise au point.

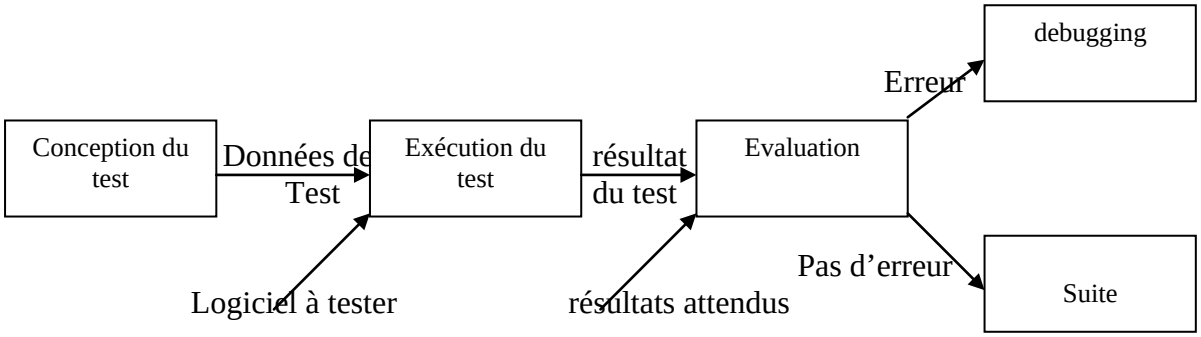
2.1.2 – planification de l'activité de mise au point.

C'est la définition d'un ensemble ordonné et systématique de tests destinés à satisfaire des critères (spécifications) fixés à l'avance.
Exemples de critères :

- Détecter et corriger un nombre d'erreurs définis à l'avance.
- Assurer un certain pourcentage de couverture de chemins.
- Etc.

L'ensemble est défini dans un plan de test. Pour chacun des tests prévus dans le plan :

- Conception du cas de test.
- Exécution du test.
- Enregistrement des résultats et évaluation.



2.1.3 – Conception de test.

Un test doit être conçu avec :

- Un objectif : genre d'erreurs à détecter.
- Une ou des techniques : approche systématique de construction du test en vue d'atteindre l'objectif.

2.1.4 – Le processus de test.

On peut déterminer cinq étapes dans un processus de test.

- Les tests unitaires : les composants sont testés individuellement comme des entités isolées.
- Les tests modulaires : un module encapsule des composants. On peut le tester indépendamment des autres modules.
- Les tests des sous systèmes : il s'agit dans ce cas de tester une série de modules qui ont été intégrés de manière à constituer un sous système. Ces sous systèmes sont conçus et implémentés indépendamment les uns des autres. Ce type de test se concentre sur la détection d'erreurs au niveau de l'interface.
- Les tests de système : tous les sous systèmes sont intégrés. Le processus de test consiste à ce niveau de mettre en évidence des erreurs d'interaction entre les sous systèmes. Les besoins fonctionnels et non fonctionnels sont également validés à ce niveau.
- Le test d'acceptation (test alpha) : il est considéré comme la dernière étape avant l'acceptation du système et sa mise en œuvre. C'est à ce niveau que les conditions fixées par le client sont vérifiées cela permet de révéler les défauts et les omissions lors de la définition des besoins.

2.1.5 – Stratégies de test.

Il existe plusieurs stratégies de tests qui sont complémentaires, chacune possède ses avantages et ses inconvenants que voici :

a) Les tests descendants.

Un test descendant suit l'approche de conception descendante. Ce test démarre du niveau «sous système », les modules étant remplacés par des «bouchons ». Un bouchon est un composant simple qui a la même interface que le composant qu'il remplace. Une fois le sous système testé, on passe au test de chaque module en utilisant le même principe.

Les avantages de ce test sont la détection (très tôt) des erreurs de conception, de plus il est moins coûteux. Par contre son inconvenant est la difficulté de simuler des bouchons pour des composants complexes, en plus de la difficulté d'interpréter les résultats des tests(pas de sortie aux niveaux supérieurs.

b) Les tests ascendants.

Cette approche de test est considérée comme l'approche opposée au test descendant. Le test démarre du plus bas niveau puis remonte dans la hiérarchie jusqu'au système final. L'approche étant opposée, les avantages et les inconvenants de cette approche correspondent respectivement aux inconvénients et avantages des tests descendants.

c) Les tests d'enchaînement

Ces tests sont généralement utilisés dans les systèmes à temps réel, ceux-ci étant souvent constitués de processus qui coopèrent. Les tests d'enchaînement prennent toute leur importance car les systèmes à temps réel sont difficiles à tester du fait des nombreuses interactions entre les différents processus.

d) Les tests de surcharge.

Ce type de test est très utile lorsque l'on a affaire à des systèmes conçus pour supporter une certaine charge. Un test de surcharge consiste alors à aller au-delà de la charge maximale d'un système jusqu'à ce que celui-ci tombe en panne. Les avantages de cette méthode sont :

- Permet de vérifier que le système surchargé n'occasionne pas de dégâts irréparables.
- Permet de faire apparaître des défauts qui ne se seraient pas manifestés autrement.
- Adapté aux systèmes distribués.

2.2 – Les techniques de test.

Il existe deux grandes méthodes de test :

Les méthodes de test statique : Ces méthodes consistent à tester le logiciel sans l'exécuter. L'idée est de ne traiter que le texte du logiciel. Par exemple des textes de spécifications.

Les méthodes de test dynamique (simulation) : Ces méthodes reposent sur l'exécution effective du logiciel sur un sous-ensemble bien choisi du domaine de ses entrées possibles.

2.2.1 – Les méthodes de test statique.

Parmi les principales méthodes de test statique on a :

Lecture croisée et inspection.

Il s'agit d'obtenir un point de vue sur un document afin de minimiser les erreurs d'interprétation.

- Lecture croisée : on associe un lecteur diffèrent du programmeur à chaque module en test.

- Inspection : c'est une relecture en groupe, après préparation individuelle. Généralement, le groupe est composé (IBM) :

- Un président garantissant l'objectivité et l'intégrité de l'inspection.
- Un chef de projet responsable de la conception du logiciel.
- L'auteur du document en test.
- Responsable des tests.

Partant d'un scénario et d'une liste de points précis de vérifications, l'inspection doit trouver des fautes éventuelles. Si des fautes sont détectées, l'inspection propose des tests dynamiques visant à vérifier ultérieurement la correction de ces erreurs.

Analyse d'anomalies.

Des anomalies peuvent être facilement détectées dans les logiciels lors de la compilation. Comme exemple d'anomalies, les violations de typages, utilisation impropre de pointeurs, etc.… généralement, l'analyse d'anomalies suit la démarche suivante :

- On établit un critère d'acceptabilité des logiciels testés, dont la vérification ne requière pas d'exécution de ces logiciels.
- On construit un graphe représentant un sous-ensemble des informations contenues dans les textes en question pour vérifier les critères choisis.
- On étudie le plus souvent des chemins du graphe, le long desquels on tient à jour un prédicat portant sur la satisfaction de chacun des critères.

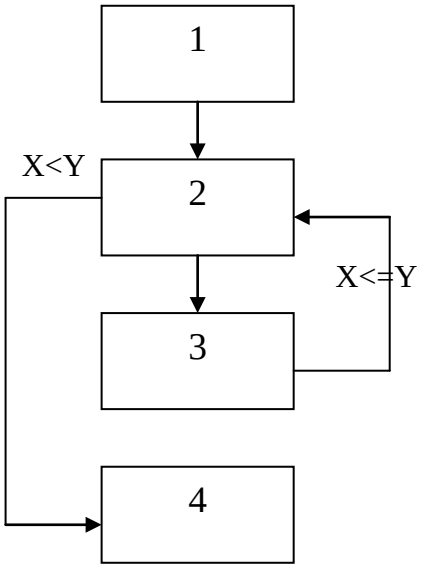
Exemple : démarche d'utilisation du graphe de contrôle.

1- Identification des blocs d'instructions indivisibles maximaux : ce sont des portions de codes qu'on exécute toujours du début à la fin c-a-d une suite d'instructions élémentaires telle que le contrôle passe consécutivement d'une suite à l'autre sans choix possible (pas de if, while …) chaque bloc n'a donc qu'un seul point d'entrée son début
2- Les sommets des graphes sont représentés par ces blocs ainsi que par les conditions associées aux instructions conditionnelles et aux boucles.
3- Les arcs reliant ses sommets correspondent aux transferts de contrôle entre les sommets.

Une instruction conditionnelle est représentée par trois sommets et deux arcs.
Une condition de boucle correspondra à un circuit du graphe.
Soit le texte suivant :

```
Open file1
Read ( x, file1)
Read ( y,file1)
Z :=0
While x>=y do  →   bloc 2
        Begin
                X :=x-y
                Z :=x+1        bloc 3
         End
Print (y,file2)
Close file1        bloc 4
Close file2
```

Le graphe de contrôle correspondant est :



Selon le type d'anomalies recherchées, on peut annoter ce graphe par toute information pertinente liée au contrôle : condition de branchement, manipulation de fichier,…. On considère alors l'ensemble des chemins allant du point d'entrée au point de sortie et on cherche les anomalies éventuellement détectables sur ces chemins.

Les différentes anomalies détectables à partir du graphe de contrôle peuvent être :

- La présence de code isolé.
- L'enchaînement logique d'instructions (exemple écrire sur un fichier non ouvert…)
- Trop de boucles imbriquées.
- Trop d'appels de fonctions enchaînées,…

Comme exemple d'annotation concernant les ouvertures et fermetures de fichiers, on constate que dans le bloc 4 le fichier 2 est utilisé sans avoir été ouvert.

Evaluation symbolique.

L'évaluation symbolique consiste à construire des expressions symboliques qui relient les données d'entrées aux données de sorties ainsi que les conditions permettant de choisir un chemin à suivre lors de l'exécution.

Pour tout chemin sélectionné du graphe de contrôle, les données d'entrée sont représentées par des symboles et l'évaluation symbolique produit, pour chaque variable de sortie du programme, une expression retraçant des calculs effectués pour l'obtenir (le long du chemin) et les conditions associées (sur le chemin). Il ne s'agit pas de test dynamique car l'évaluation symbolique ne requière pas d'exécution du logiciel sur des données réelles.

Graphes de MARKOV (Méthode des Espaces d'état)

Les graphes de MARKOV ne reposent pas sur une simulation du fonctionnement du système mais sur la mise en équations du fonctionnement en vue de leur traitement mathématique.

Processus semi-markoviens à espace d'états quelconque

Cette approche se base sur la modélisation des principaux indices de la fiabilité à l'aide de quatre fonctions il a été montré qu'elles vérifient des équations de renouvellement markovien. Par la suite, ont été établi des conditions d'existence et d'unicité des solutions de ces équations. Cette étude offre un cadre tout à fait général pour l'étude de la fiabilité de ces systèmes **[12]**.

Approximations des distributions de durées de vie par des distributions de type phase (ou Ph-distributions)

Il s'agit des distributions de temps d'entrée dans un sous-ensemble d'états d'un processus de Markov dont la distribution gamma est un cas particulier. Il a été développé des méthodes d'approximation de distributions quelconques par des Ph-distributions. Cela permet d'obtenir, d'une part, une approximation de systèmes semi-markoviens par des systèmes markoviens et, d'autre part, un traitement de données de la fiabilité par ces distributions. Il a été également établi des inégalités générales concernant la distance entre les fonctions de transitions de deux processus semi-markoviens ayant le même espace d'états. Ces inégalités permettent, entre autre, d'estimer l'erreur commise lors des approximations évoquées ci-dessus **[1]**.

De plus il existe un certain nombre de métriques spécifiques aux tests statiques tel que les métriques textuelles : HALSTEAD et les métriques structurelles : Mac Cabe et Henry Kafura. Dans ce qui suit, on présentera la métrique Mac Cabe

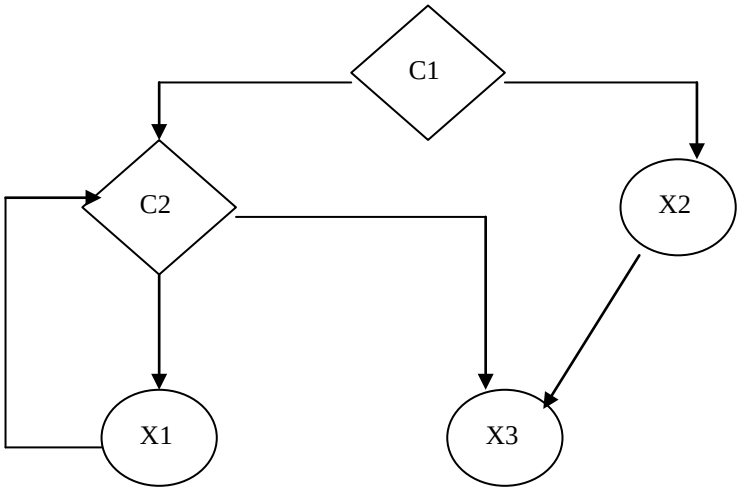
Mac Cabe
La Complexité cyclomatique.

Introduite par Mac Cabe, la complexité cyclomatique définit une mesure quantitative de la complexité logique d'un programme :
V(G) = E – N + 2 tel que :
V(G) : complexité cyclomatique
E : nombre d'arcs (Branches suivies par le programme)
N : nombre de nœuds (branches suivies par le programme)

Quant on dispose d'un point d'entrée et d'un point de sortie on applique la règle précédente tandis que quant on dispose de i points d'entrée et de s points de sorties on applique la règle suivante :

V(G) = E – N +i +s

traitons l'exemple suivant :
if C1 then while (C2) do X1 ; else X2;
X3;



on aura V= 3 (nombre de décisions = 2) n=5 et e=6

L'intérêt de la complexité cyclomatique est qu'elle définit une borne supérieure du nombre de chemins indépendants composant l'ensemble de base. C'est à dire la borne supérieure du nombre de test à concevoir et pour exécuter toutes les instructions du programme.

Application :

- Procédure de définition des cas de tests.

- ♦ Dessiner le graphe des flots du programme.
- ♦ Définir le nombre de complexité cyclomatique.
- ♦ Définir un ensemble de chemins linéairement indépendants.(2)

2.2.2 – Les méthodes de tests dynamiques.

Il existe deux grandes catégories de test, correspondant à deux objectifs :

- **Test de boite noire** : considère que la fonction du produit est bien spécifiée et vérifie que l'implantation (le logiciel développé) réalise correctement tous les aspects de la spécification.
- **Test de boite blanche** : considérer l'intérieur du produit comme bien spécifié et vérifier que la réalisation de l'intérieur est conforme à sa spécification.

a) Tests de boite noire [20].

Dans ce genre de test qui se déroule après l'installation du logiciel on ne s'intéresse pas au code source de ce dernier mais on étudie l'exécutable, l'objectif de ce type de tests est d'appliquer au logiciel une simulation pour savoir si ce dernier répond aux spécifications pour cela on l'exécute avec plusieurs ensembles de données d'entrées afin de pouvoir vérifier si chacune des spécifications est correctement réalisée. Contrairement aux tests à boite blanche il est possible d'exécuter ce type de tests sur n'importe quel logiciel ou produit acheté par l'utilisateur même si ce dernier ne possède pas le code source, ce qui n'est pas le cas pour les tests à boite blanche. Certaines approches existent tel que :

- La partition en classes d'équivalence
  - données valides et invalides
    - 1..99 donne 3 classes : [1, 99], > 99 , < 1
    - (émetteur, destinataire, réseau) donne 4 classes dont 3 valides
- Analyse aux bornes du domaine :Boundary domain analysis
  - Analyse précise aux bornes des classes d'équivalence des valeurs d'entrée.
- Graphe cause à effet :
  - sélection des combinaisons d'entrée de manière systématique.

b) Test de boite blanche (structurel) [16].

Objectif :

- ❑ Garantir que tous les chemins indépendants du module ont été exécutes au moins une fois.
- ❑ Exécuter les branches VRAIES et FAUX de toutes des décisions logiques.

[2] Un chemin qui parcours au moins un arc pas encore parcouru, c-a-d qui introduit au moins une nouvelle instruction.

- Exécuter toutes les boucles, dans leurs limites opérationnelles et à leurs limites.
- Tester les structures internes de données pour s'assurer de leur validité.

Justification des tests de la boite blanche (par opposition à la boite noire).

- Les erreurs ont tendance à ce concentrer dans des chemins, instructions et conditions qui sont hors de l'exécution normale.
- On a tendance à croire qu'un chemin particulier a peu de chances d'être exécuté alors qu'il l'est très souvent.
- Les erreurs typographiques sont reparties au hasard.

L'utilisation des tests à boite noires pour détecter ce genre d'erreurs serait beaucoup plus cher et plus long.

C) Test statistique [21]

Le test statistique est une méthode originale de vérification dynamique, qui utilise des données de test générées aléatoirement selon des lois probabilistes définies en fonction de l'objectif du test. Cette approche se démarque du test aléatoire classique qui consiste à générer des entrées selon une distribution uniforme sur le domaine d'entrée, et dont le caractère aveugle explique le manque d'efficacité. Le test statistique est basé sur la notion de qualité d'un test mesurée en termes de couverture vis-à-vis de critères structurels ou fonctionnels. Cette notion permet d'utiliser l'information procurée par un critère de test pour définir un jeu de test statistique, c'est-à-dire, pour choisir une distribution des probabilités sur le domaine d'entrée appelée profil de test, et un nombre d'exécutions nécessaire pour atteindre un objectif de qualité fixé.

Ces travaux ont été valorisés par une étude dont l'objectif était la conception et la génération de jeux de test statistique fonctionnel à partir d'une spécification SA/RT d'un logiciel extrait d'Ariane 5 (système de régulation de pression dans le réservoir d'hélium). Cette spécification avait été faite dans le cadre d'une action de recherche et de développement ciblant la méthode SA/RT. L'industrialisation de cette méthode a mis en évidence quelques dysfonctionnements sur le prototype de recherche et de développement développé à partir de la spécification SA/RT.

D) quelques outils utilisés pour la simulation.

<u>Réseaux de PETRI</u>

Les réseaux de Pétri sont une représentation du fonctionnement du système, qui, par un traitement logiciel permet de simuler le fonctionnement du système.

Le résultat de ce traitement permet de déterminer la probabilité d'occurrence de chaque état stable du système et donc par un ratio simple de calculer le rapport entre le temps de disponibilité et d'indisponibilité.

<u>APR (Analyse Préliminaire des Risques)</u>

Cette analyse permet d'identifier les situations potentiellement dangereuses vis-à-vis de la sécurité et d'évaluer la gravité des conséquences.

Tests des chemins de base.

Permet de définir un ensemble de base de chemins d'exécution. Les cas de tests qu'on en déduit garantissent l'exécution au moins une fois de chaque instruction.

2.2.3 - Les voies de recherches.

Il est admis qu'avec l'émergence des calculateurs tolérant les fautes physiques, l'influence des fautes imputables au logiciel sur le processus de défaillance d'un système informatique devient de plus en plus forte. Dans ce cadre, une voie de recherche actuelle considère des modèles du logiciel en phase de développement ou en opération, pour réaliser des prédictions du point de vue de la fiabilité. On constate que la plupart de ces modèles ne prennent pas en compte la structure du logiciel mais considèrent ce dernier comme une boîte noire. Il s'agit de modèles paramétriques relativement simples et qui se sont avérés précis dans certains cas.

Une approche markovienne, basée sur une représentation structurelle des systèmes a été développé dans le but d'explorer le gain possible en précision pour la prédiction et en utilisant les processus de Markov. Cette approche généralise des modèles précédemment publiés, en prenant en compte plusieurs caractéristiques supplémentaires des systèmes. Elle construit un modèle de défaillances distinguant celles ayant une influence sur l'évolution du processus d'exécution (par exemple, les défaillances nécessitant un redémarrage du système) de celles dont l'action sur le service requis peut être négligée (par exemple, certaines anomalies transitoires), mais que l'on souhaite tout de même comptabiliser. On peut également tenir compte des fins correctes d'exécution (services délivrés) pour permettre une évaluation de la disponibilité. Le modèle inclut des délais de reprise d'exécution.

Toujours dans le domaine des approches markoviennes James Ledoux a développé un model structurel markoviens dédié aux systèmes logiciels. Il permet le calcul (exact) de diverses métriques de sûreté de fonctionnement en tenant compte des éventuels temps de recouvrement des défaillances. Ces indices sont obtenus sous une forme analytique en utilisant les propriétés des processus markoviens, de plus il donne les moyens de mettre en œuvre leur 'évaluation numérique. Le modèle décrit est suffisamment général pour inclure la majorité des modèles structurels markoviens connus **[10]**

Apres avoir connu les différents types et techniques de tests ainsi que les démarches à suivre pour pouvoir les réalisés la question qui reste posée est la suivante :

A quel moment devrons nous arrêter un processus de test ? ?

Pour cela certaines conditions d'arrêt ont été définies :

- épuisement des ressources
- taux de couverture atteint
- durée
- nombre d'erreurs découvertes (avec catégorie)
- étude de courbe nombre d'erreur en fonction de la durée
- aucune erreur révélée dans ce cas qui est très rare on considère que le test n'a pas été concluant ou bien que c'est un mauvais test 'soit pas adapté à la méthode ou bien c'est le test qui n'est pas à la hauteur)

Cependant on est en droit de se poser la question suivante certains logiciels accomplissent des taches où l'erreur n'est pas admise dans ce cas bien précis est ce qu'il serait logique d'utiliser l'un des critères de tests cités précédemment pour certifier la validité de tels logiciels ? évidement la réponse est non car les méthodes et les critères d'arrêt de test qu'on a cité dans ce chapitre ne pourront jamais garantir l'absence totale d'erreurs, c'est pour cela qu'on utilise des approches bien spécifiques pour ce type de logiciel. Ces approches seront détaillées dans chapitre suivant.

CHAPITRE 3 :

LA SURETE DE FONCTIONNEMENT DES LOGICIELS

CHAPITRE 3 : LA SURETE DE FONCTIONNEMENT DES LOGICIELS

3.1 – problématique :

La progression constante de l'insertion de composants logiciels dans les biens de consommation les a tout naturellement amenés à assurer des fonctions critiques. La criticité est une notion assez difficile à définir. Intuitivement on pense à des fonctions qui si elles n'étaient pas correctement réalisées mettraient en jeu des vies humaines.

Il y a maintenant beaucoup de systèmes dont le moindre dysfonctionnement peut mettre en danger des vies humaines. Les systèmes de contrôle et de surveillance des avions sont des exemples de systèmes critiques.

On utilise l'expression logiciel critique pour designer des logiciels inclus dans des systèmes dont un dysfonctionnement peut s'avérer dangereux. Mais de toute évidence, le logiciel ne menace personne par lui-même. L'ultime conséquence d'un dysfonctionnement logiciel est une panne matérielle.

Les techniques d'assurance et d'analyse de la sûreté dérivées des systèmes matériel électromécaniques ne sont pas appropriées aux systèmes basés sur des logiciels. Cela est du à la complexité bien supérieure du logiciel, et au fait que l'ensemble du système est devenu plus complexe grâce à la flexibilité du logiciel.

Il existe deux catégories de systèmes logiciels critiques :

A) Le logiciel critique primaire.

Ce logiciel est intégré dans un système matériel utilisé pour contrôler et surveiller d'autres processus. Le dysfonctionnement d'un tel logiciel peut avoir un impact direct sur les vies humaines ou sur son environnement.

B) Le logiciel critique secondaire.

Ce logiciel peut indirectement causer des dégâts. Il s'agira par exemple, d'une de donnée médicale qui contient les détails sur les dosages des médicaments, si elle fonctionne mal, peut être à l'origine du mauvais dosage administré à un patient.

3.2 – Quelques exemples de dysfonctionnement d'un logiciel critique [17]

L'informatique médicale est à coup sûr un domaine d'applications critiques. On perçoit directement l'impact d'erreurs logicielles dans des appareils destinés à prendre en charge les fonctions vitales d'un patient. L'accident le plus grave

répertorié dans ce domaine concerne un appareil de traitement par rayons, le Therac 25(juin 1985 – janvier 1987). Partant du principe qu'un programme informatique ``ne s'use pas'', les concepteurs de cet appareil ont réalisé en logiciel de nombreuses fonctions de sécurité habituellement confiées à des dispositifs matériels indépendants. N. G. Leveson explique comment de mauvaises pratiques de Génie Logiciel, une procédure de validation insuffisante et un mauvais suivi des rapports d'erreurs de fonctionnement ont laissé subsister trop longtemps des portions de code incorrectes. En conséquence, des patients furent irradiés au delà des seuils admissibles et plusieurs en moururent.

Nom de l'erreur	Date de l'erreur
Fausse alerte au NORAD	Juin 1980
Refus d'autorisation des cartes de crédits en France	26 et 27 juin 1993
Embouteillage de portails web	Février 2000

Tableau : Quelque exemple de défaillances informatique

3.3 – Quelques définitions.

- **Un accident** *:* c'est un événement ou une séquence d'événements que l'on n'avait pas prévu et qui peut occasionner des pertes ou dommages humaines, économiques ou écologiques.
- **Un risque** *:* c'est une condition qui cause ou contribue à causer un accident.
- **Un dommage** *:* c'est la mesure des pertes subies suite à un accident.
- **La gravité d'un risque** *:* c'est l'estimation du pire que le risque peut occasionner.
- **La probabilité d'un risque** *:* c'est la probabilité d'occurrence des événements qui sont à l'origine des risques.
- **Un péril** *:* c'est un concept complexe qui a trait à la fois à la gravité d'un risque, à sa probabilité, et à la probabilité pour qu'il occasionne vraiment un accident. En gros, c'est une mesure de la probabilité pour que le système ne fonctionne pas comme prévu et mette en jeu des vies humaines. L'objectif de tous les systèmes sûrs étant de minimiser ce péril.
- **La sûreté de fonctionnement :** C'est empêcher les causes du risque de se produire (prévention), diminuer la graviter des conséquences jusqu'à un niveau acceptable (protection). Prévenir et se protéger des risques inacceptables. Il s'agit d'identifier les événements redoutés, de rechercher les causes éventuelles de ces événements et de mettre en place les parades pour prévenir et se protéger **[2]**. En d'autres termes la sûreté de fonctionnement d'un système informatique est la propriété qui permet aux utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre. Le service délivré par un système est son comportement tel que perçu par son ou ses utilisateur(s) ; Un utilisateur est un autre système (humain ou physique) qui interagit avec le système considéré. **[24]**

L'une des hypothèses de base lorsque l'on travaille sur la sûreté de fonctionnement d'un système, c'est que le nombre de fautes qui peuvent occasionner un risque est très inférieur au nombre total des fautes du système. Si on peut démontrer que ces fautes n'arrive jamais ou que lorsqu'elles arrivent le risque ne débouche pas sur un accident, alors on démontre que le système est sûr.

En dehors du fait que l'on ne peut pas être sûr qu'un système est sans fautes à 100%, il y a plusieurs raisons pour ne pas confondre fiabilité et sûreté de fonctionnement :

1- Un important pourcentage de dysfonctionnement est du à des erreurs de spécification et non de conception.
2- Les dysfonctionnements matériels peuvent constituer un environnement qui n'était pas prévu par le logiciel.
3- L'opérateur d'un système peut générer des entrées dont aucune n'est incorrecte mais dont la combinaison peut mener à un dysfonctionnement.

Toutes les possibilités peuvent arriver en même temps. Une analyse des accidents sérieuse laisse à penser que ces derniers sont quasiment tous dus à une combinaison de dysfonctionnement plutôt qu'a une simple panne. Des combinaisons auxquels on n'avait pas pensé peuvent entraîner des interactions qui occasionneront la panne.

Cependant il est impossible de prévoir toutes les combinaisons possibles de dysfonctionnement. Les accidents deviennent inévitables à partir du moment ou on utilise des systèmes complexes. Comme le logiciel a tendance à augmenter la complexité des systèmes, le fait d'effectuer un contrôle par logiciel est susceptible d'augmenter la probabilité pour que le système ait un accident.

Mais il y a des avantages à utiliser du logiciel dans les applications critiques car il permet de surveiller plus de conditions qu'un système électromécanique, et il peut s'adapter aisément. Cela nécessite d'utiliser des ordinateurs très fiables, très petits et très légers. On peut élaborer des systèmes très sophistiqués et utiliser des stratégies de contrôle qui réduisent le temps que l'homme passe dans un environnement dangereux.

Ainsi le logiciel de contrôle et de surveillance peut améliorer la sûreté de fonctionnement des systèmes. Même si on doute de la fidélité d'un logiciel, on peut augmenter la sûreté de fonctionnement du système final, maigres le résidu de pannes de logiciels risquées. Quant à déterminer si les bénéfices que l'on tire de l'utilisation, du logiciel contre balancent les risques que l'on introduit, c'est une décision politique et sociale.

3.4 - Quelques méthodes et outils pour assurer la sûreté de fonctionnement.

3.4.1 – Les méthodes formelles

Les méthodes formelles peuvent être vues comme les mathématiques appliquées de l'ingénieur logiciel. En pratique ce terme regroupe des langages dont la sémantique est définie mathématiquement et les techniques de transformation et de vérification basées sur la preuve mathématique

Les méthodes formelles se sont surtout attachées à la spécification des systèmes. Une première approche consiste en la modélisation d'un système par

l'énumération de ses propriétés sans aucune référence sur leur réalisation. C'est l'hypothèse axiomatique ou algébrique basée sur les types abstrait de données.

Une approche alternative paraît s'imposer dans la pratique industrielle. Il s 'agit de l'approche basée sur les models. Par model en entend des concepts mathématiques bien connus (ensembles, listes, fonctions….) qui regroupent implicitement une liste de propriétés. Une spécification est alors une combinaison de tels models complétée par l'expression de propriétés spécifiques à l'application. Z et VDM sont deux langages correspondant à cette approche ils font tous les deux l'objet d'un effort de standardisation. La méthode B est apparue plus récemment. Elle va au-delà de la phase de spécification et permet de construire par raffinement des programmes dont la correction est établie par rapport à cette spécification **[8]**.

Dans ce qui suit, on va essayer de détailler un certain nombre de ces méthodes formelles.

3.4.1.1 - La preuve par contradiction.

La preuve par contradiction constitue l'une des techniques que l'on peut utiliser pour effectuer des preuves de sûreté de fonctionnement. Pour cela, on suppose que le système peut se retrouver dans un état non sûr (que l'on identifie lors de l'analyse des risques), puis de prouver que c'est une contradiction.

3.4.1.2 - la méthode B [15]

La méthode B a été inventé par Jean Raymond ABRIAL, à partir des travaux de E.W Dijkstra et de C.A.R Hoare. Elle utilise une notation fondée sur les concepts mathématiques de la théorie des ensembles. L'existence d'une notation utilisable tout au long du développement offre un cadre formel uniformisé allant de la spécification et la conception jusqu'à la réalisation des composants logiciels exécutables

Généralement l'expression initiale des besoins est effectuée en langage naturel ou par juxtaposition de plusieurs descriptions : Graphes, automates, tables logiques, réseaux de pétrie, et les méthodes comme UML.

Un développement B débute par la construction d'un model reprenant toutes les descriptions du besoin, en décrivant les principales variables d'état du système, les propriétés que ces variables devront satisfaire à tout moment et les transformations de ces variables par des services. Le model B obtenu constitue une spécification de ce que devra réaliser le système.

Le model B est ensuite raffiné jusqu'à obtenir une implantation complète du système logiciel. Plusieurs raffinements peuvent satisfaire une spécification, le choix des solutions reposant sur des critères divers comme la rapidité du traitement, la précision des calculs ou la simplicité à être démontré. Le raffinement peut également être utilisé comme technique de spécification. Dans ce cas, le raffinement permet

d'inclure petit à petit les détails du problème dans le développement formel. La spécification formelle est alors réalisée progressivement et non pas directement.

Utiliser B dans le développement d'un système est donc :

- Lever toutes ambiguïtés dés l'interprétation du besoin.
- Construire une spécification cohérente et conforme au besoin.
- Elaborer par étapes successives le système logiciel qui réalise la spécification.

3.4.1.3 – Validation formelle de la spécification d'un logiciel

A – Objectifs de la méthode

La méthode de Validation Formelle de la spécification permet au Maître d'œuvre d'avoir au plutôt l'assurance de la complétude et de la cohérence des fonctionnalités portées par le logiciel généralement décrites dans les spécifications du logiciel. Cette assurance est apportée par l'établissement du modèle formel de ces fonctionnalités et la démonstration sous forme de preuves mathématiques du respect des propriétés de sûreté des fonctions du logiciel dans le système.
(Cette démarche est "Hautement Recommandée" pour les logiciels de niveau SIL 4 dans la norme CEI 61508)[3].

B – Démarche

1 – A partir de la documentation du système et du logiciel, obtention du modèle fonctionnel orienté sûreté (chemins fonctionnels et modèle des flux).
2 – inversion des chemins du modèle fonctionnel orienté sûreté, réalisation du modèle formel en langage PVS.
3 – A partir de la documentation du système et du logiciel, identification des propriétés de sûreté à valider et traduire ces propriétés en langage PVS.
3 – A l'aide de l'outil PVS, réalisation de la preuve des propriétés de sûreté sur le modèle formel.

C – Résultats

- Ensemble de remarques, sous la forme de Fiches d'avis, sur la complétude et la cohérence des spécifications
- Modèle formel des fonctionnalités du logiciel, ce type de résultat peut servir de base à la définition des scénarios de Tests de Validation de la SdF du logiciel
- Dossier de Preuve formelle mathématique des propriétés de sûreté respectées par le logiciel.

3.4.2 – La modélisation structurelle marcovienne.

La sûreté de fonctionnement est une qualité essentielle requise pour un système réparable. Ce model est un modèle structurel markovien dédié aux systèmes logiciels. Il permet le calcul (exact) de diverses métriques de sûreté de fonctionnement en tenant compte des éventuels temps de recouvrement de défaillances. Ces indices sont obtenus sous une forme analytique en utilisant les

[3] : c'est une norme utilisée par le département d'état américain pour la validation des logiciels jugés critique

propriétés des processus markoviens. Ce modèle est suffisamment général pour inclure la majorité des modèles structurels markoviens connus et en particulier celui de Littlewood.

3.4.3 – ARLIA WorkShop.

Arlia Work Shop est un outil complet permettant de réaliser une étude de sûreté de fonctionnement d'un spectre large allant des études probabilistes de la sûreté nucléaire jusqu'à des études de disponibilité ou des études incidentielles.

Le logiciel Arlia WorkShop repose d'une part sur les développements réalisés au sein du groupe Arlia [4] : Une instance d'édition et de présentation des arbres de défaillances : SimTree connecté au cœur de calcul Arlia (logiciel de traitement des formules booléennes probabilisées). D'autre part sur les développements réalisés par le groupe Hévéa [5] : une interface d'édition et de présentation des arbres d'événements SimEvent connecté à un cœur de calcul dénommé Hévéa(logiciel encapsulant le code Arlia et traduisant les séquences d'adE[6] en formules booléennes interpretablent par Arlia.

3.4.4 - La méthode AMDEC.

Historiquement, la méthode initiale est appelée Analyse des modes de défaillances et de leurs effets (AMDE). Il s'agit d'une méthode d'analyse préventive de la sûreté de fonctionnement (fiabilité, disponibilité, maintenabilité, sécurité). Développée aux Etats-Unis, dans l'industrie aéronautique, au début des années soixante. Elle a pris son essor en Europe au cours des années soixante-dix dans l'industrie automobile, chimique et nucléaire. La méthode AMDEC a ajouté l'estimation de la dimension critique des risques.

Le principe de la prévention repose sur le recensement systématique et l'évaluation des risques potentiels d'erreurs susceptibles de se produire à toutes les phases de réalisation d'un système.

Les aspects originaux de la méthode sont les suivants :

- L'AMDEC a pour but d'évaluer l'impact ou la criticité des modes de défaillances des composants d'un système sur la fiabilité, la maintenabilité, la disponibilité et la sécurité de ce système

- Appliquée en groupe de travail pluridisciplinaire, elle est recommandée pour la résolution de problèmes mineurs dont on veut identifier les causes et les effets.

- La démarche AMDEC consiste à recenser les modes de défaillance des composants, d'en évaluer les effets sur l'ensemble des fonctions de ce système, d'en analyser les causes.

[4] groupe Arlia : CEA, Dassault, EDF, ELF-EP, LABRI, LADS, SGN, Schneider électrique et technicatome.
[5] Groupe hévéa : ELF-EP, IPSN, IXI, LABRI et Technicatome.
[6] Arbre d'événement

- En phase de conception, l'AMDEC est associée à l'Analyse fonctionnelle pour la recherche des modes de défaillances spécifiques à chaque fonction ou contrainte des composants. Elle peut intervenir à titre correctif pour l'amélioration de systèmes existants

- Cette méthode est qualifiée d'inductive au sens où elle s'appuie, pour l'analyse des défaillances, sur une logique de décomposition d'un système en sous-ensembles successifs pour parvenir au niveau des composants élémentaires. On s'intéresse alors aux défaillances liées au mauvais fonctionnement de ces composants et à leurs répercussions aux niveaux supérieurs du système.

Bien que L'AMDEC ait été employée pour la première fois pour l'analyse de la sécurité des avions. La mise en œuvre s'est longtemps limitée à l'utilisation dans le cadre d'études de fiabilité sur du matériel. Bien qu'ayant subi de nombreuses critiques dues au coût et à la lourdeur de son application, elle reste néanmoins une des méthodes les plus répandues et l'une des plus efficaces. Elle est en effet de plus en plus utilisée en sécurité, maintenance et disponibilité non seulement sur le matériel, mais aussi sur le système, le fonctionnel et le logiciel.

Aussi est-elle maintenant largement recommandée au niveau international et systématiquement utilisée dans toutes les industries à risque, comme le nucléaire, le spatial et la chimie, dans le but de faire des analyses préventives de la sûreté de fonctionnement.

Méthodologie

Avant de se lancer dans la réalisation proprement dite des AMDEC, il faut connaître précisément le système et son environnement. Ces informations sont généralement les résultats de l'analyse fonctionnelle, de l'analyse des risques et éventuellement du retour d'expériences.

Il faut également déterminer comment et à quelle fin AMDEC sera exploitée et définir les moyens nécessaires, l'organisation et les responsabilités associées. Dans un second temps, il faut évaluer les effets des modes de défaillance. Les effets de mode de défaillance d'une entité donnée sont étudiés d'abord sur les composants directement interfacés avec celui-ci (effet local) et de proche en proche (effets de zone) vers le système et son environnement (effet global). Il est important de noter que lorsqu'une entité donnée est considérée selon un mode de défaillance donné, toutes les autres entités sont supposées en état de fonctionnement nominal. Dans un troisième temps, il convient de classer les effets des modes de défaillance par niveau de criticité, par rapport à certains critères de sûreté de fonctionnement préalablement définis au niveau du système en fonction des objectifs fixés (fiabilité, sécurité, etc.).

Les modes de défaillance d'un composant sont regroupés par niveau de criticité de leurs effets et sont par conséquents hiérarchisés. Cette typologie permet d'identifier les composants les plus critiques et de proposer alors les actions et les procédures " juste nécessaire " pour y remédier. Cette activité d'interprétation des résultats et de mise en place de recommandations constitue la dernière étape de l'AMDEC.

Bien que simple, la méthode s'accompagne d'une lourdeur certaine et la réalisation exige un travail souvent important et fastidieux. Une des difficultés est dans l'optimisation de l'effort entre le coût de l'analyse AMDEC (dépendant de la profondeur de l'analyse) et le coût de l'amélioration à apporter. La solution pour surmonter le volume des entités à étudier est de conduire des AMDEC fonctionnelles. Cette approche permet de détecter les fonctions les plus critiques et de limiter ensuite l'AMDEC " physique " aux composants qui réalisent tout ou partie de ces fonctions. La cohérence entre d'une part la gestion des AMDEC et des améliorations préconisées et d'autre part, les différentes versions du système est l'une des autres principales difficultés à résoudre. Aussi, la méthode n'est pas bien adaptée aux projets en temps réel car elle ne permet pas de bien appréhender l'aspect temporel des scénarios.

Néanmoins l'AMDEC fournit :

- une autre vision du système,
- des supports de réflexion, de décision et d'amélioration,
- des informations à gérer au niveau des études de sûreté de fonctionnement et des actions à entreprendre.

3.5 – Conclusion

Dans ce chapitre on a étudié un certain nombre de méthodes utilisées pour assurer la sûreté de fonctionnement des logiciels critiques. On a remarqué que ces méthodes ont chacune un principe différent et elles utilisent toutes des approches différentes, cette variabilité d’approche et de principe est due aux différents types de logiciels qui existe et à leurs différentes orientations, car chaque types de logiciels a ses faiblesses et ses propres domaines d’application c’est pour cela que dans le chapitre suivant nous essaierons de présenter une approche de sûreté de fonctionnement pour chaque type de logiciel.

CHAPITRE 4 :

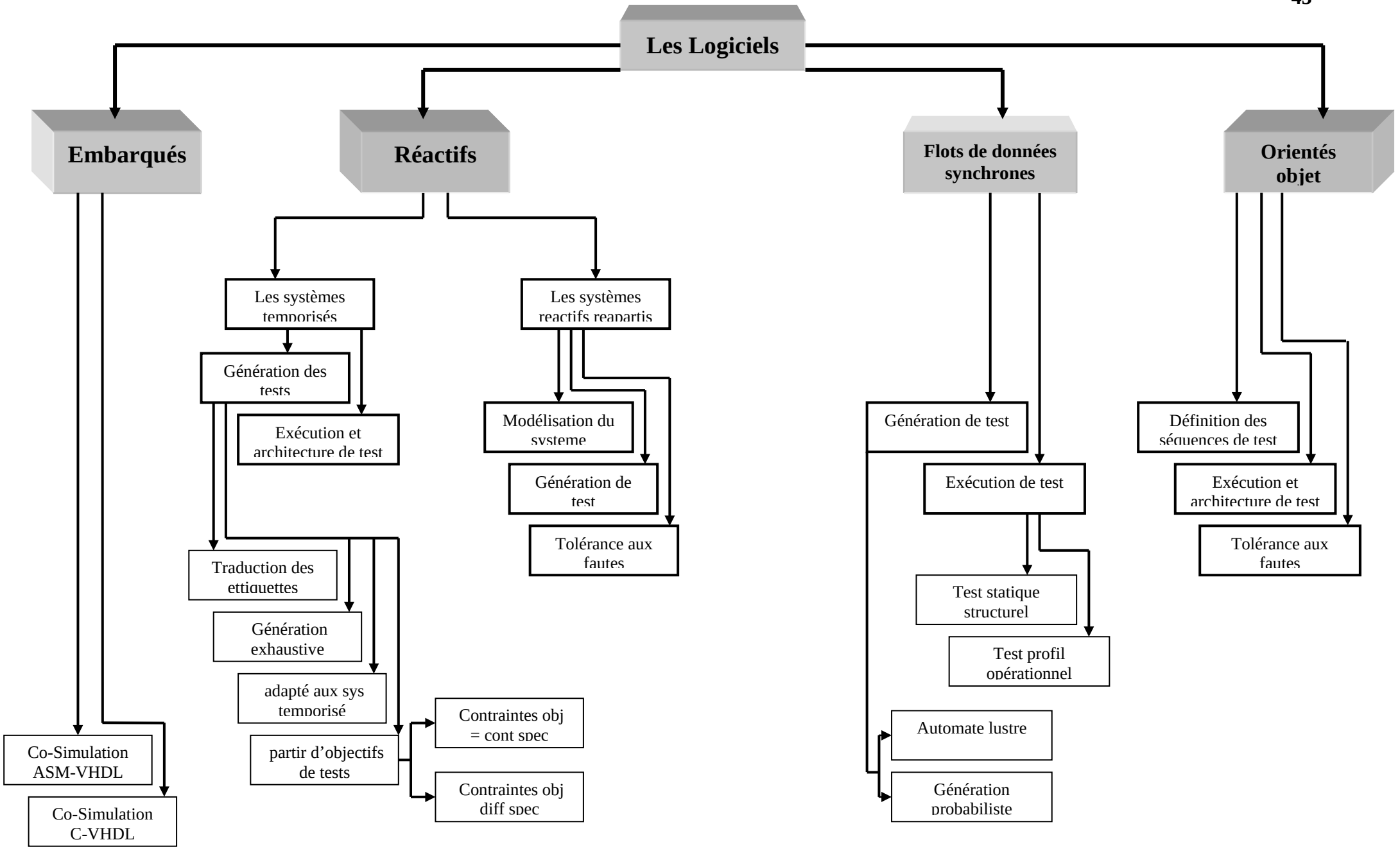
Les différentes approche utilisées pour valider les différentes classes de logiciels.

CHAPITRE 4 : Les différentes approche utilisées pour valider les différentes classes de logiciels.

Comme on l'a vu précédemment plusieurs méthodes et approches sont utilisées pour assurer la fiabilité et la sûreté de fonctionnement des logiciels. Le choix entre ces différentes méthodes dépend du type de logiciel auquel est destinée chaque méthode. Dans ce qui suit on présentera certaines approches utilisées pour les classes de logiciels suivantes :

- Les systèmes embarqués
- Les systèmes réactifs.
- Les systèmes temporisés.
- Les logiciels synchrones.
- Les logiciels orientés objet.
- Et les systèmes repartis commerciaux.

Pour chaque classe de logiciels citée précédemment on a tenter de présenter l'approche la plus utilisé et la plus adapté tandis que pour certains on a donné l'état de l'art et l'état de la recherche pour ces logiciels.


Les Logiciels
Embarqués
Réactifs
Flots de données synchrones
Orientés objet
Les systèmes temporisés
Les systèmes reactifs reapartis
Génération des tests
Exécution et architecture de test
Modélisation du systeme
Génération de test
Tolérance aux fautes
Traduction des ettiquettes
Génération exhaustive
adapté aux sys temporisé
partir d'objectifs de tests
Contraintes obj = cont spec
Contraintes obj diff spec
Co-Simulation ASM-VHDL
Co-Simulation C-VHDL
Génération de test
Exécution de test
Test statique structurel
Test profil opérationnel
Automate lustre
Génération probabiliste
Définition des séquences de test
Exécution et architecture de test
Tolérance aux fautes

4.1 – Les logiciels embarqués[22].

4.1.1 - Introduction

Les technologies récentes d'intégration de circuits permettent la réunion sur une seule puce de la totalité des composantes de systèmes complexe, les applications de téléphonie mobile de télévision numérique et de visiophonie sont des exemples de tels systèmes. En parallèle les exigences de flexibilité exigent d'accroître la programmabilité de ces produits, ce qui rend l'embarcation de processeurs inévitable. Associé à ces processeurs, le logiciel embarqué réclame une conception particulière basée sur une validation approfondie avant la réalisation du système. Dans ce qui suit on donnera un aperçu sur une approche basée sur la co-simulation C-VHDL permettant de valider la fonctionnalité d'un logiciel embarqué dans son environnement matériel réel sans pour autant requérir un model du processeur.

La conception d'un système intégré contenant un ou plusieurs processeurs spécifiques requière l'utilisation d'une conception conjointe matérielle et logicielle.

En effet la particularité de la conception d'un système intégré est que le logiciel embarqué doit être développer et valider avant la réalisation du matériel. Le fait que le processeur réel ne soit disponible pour le développement du logiciel rend ce dernier plus difficile puisqu'on ne peut ni explorer plusieurs solutions ni tester la solution finale. L'intérêt se porte donc à anticiper la validation fonctionnelle en environnement réel. La validation d'un logiciel embarqué se fait en trois phases :

- **La validation interne :** où l'on s'assure de l'exécution correcte d'une grande partie de l'algorithme indépendamment du reste du système
- **La validation de la communication avec l'extérieur :** on s'assure que les informations échangées avec le reste du système sont correctement transmises et interprétées.
- **La validation dans un environnement réel :** on vérifie que le logiciel réagit correctement lorsque le système complet est intégré.

Nous nous intéressons particulièrement à la troisième étape car il existe des méthodes classiques mais très pénalisantes en terme de temps de développement

4.1.2 - Approche classique de la validation logicielle.

Le flot de conception classique de circuit intégré propose une étape de validation complète du logiciel embarqué, apparaissant relativement tard dans le processus de développement. Le flot de conception classique d'un système contenant un processeur et son logiciel embarqué se passe comme suit : A partir des spécifications complètes du système, les descriptions de haut niveau des parties matérielles et logicielles sont produites (manuellement). Pour la partie matérielle un langage tel que VHDL ou verilog est généralement utilisé. La partie logicielle utilise le flot de compilation C

Le logiciel écrit en C est compilé en code assembleur, par un compilateur développé spécialement pour ce processeur. Ce code assembleur est ensuite

chargé dans la mémoire programme du modèle VHDL du processeur, lequel est simulé conjointement avec le reste du système par un unique simulateur VHDL. Cette simulation est appelée simulation au niveau assembleur. Grâce à l'exactitude temporelle des simulations VHDL, la validation du logiciel embarqué atteint un niveau de précision au cycle près. Le prix à payer est une vitesse de simulation relativement faible, puisqu'elle ne dépasse pas quelques instructions par seconde pour un seul processeur

Cette approche de la validation présente un inconvénient majeur : la simulation du logiciel avec le matériel intervient tard dans le flot de développement.

Le logiciel doit évidemment être développé (en C), de même que le compilateur complet doit être développé et validé. De plus, le processeur lui-même doit être disponible, et donc ses spécifications figées (interface et jeu d'instructions). Il est observé que plus la validation intervient tôt dans la conception d'un système, plus une erreur est corrigée rapidement.

Dans ce contexte, la moindre modification des spécifications (processeur,interface, jeu d'instructions) implique de re-générer le modèle du processeur ainsi que le compilateur. Le temps de cycle est alors trop long pour espérer explorer plusieurs solutions.

4.1.3 - La co-simulation matériel-logiciel

L'inconvénient principal du flot de conception classique, précédemment décrit, est une validation tardive du logiciel embarqué, essentiellement due à la nécessité de disposer du modèle matériel réel du processeur embarqué. L'objectif est donc de s'affranchir de l'indisponibilité de ce composant lors de la validation logicielle. Le logiciel serait alors exécuté conjointement avec le matériel, par l'intermédiaire d'un outil de liaison ne faisant pas partie du système final. Le terme de co-simulation, sous-entendu matérielle-logicielle, désigne cette simulation conjointe. Nous distinguerons deux types de co-simulations, selon le niveau auquel elle s'effectue. La co-simulation de bas niveau fait intervenir la forme assembleur (binaire) du logiciel embarqué alors que la co-simulation à haut niveau ne nécessite que le modèle écrit en C du logiciel.

4.1.3.1 - La co-simulation assembleur-VHDL

Cette méthode consiste à substituer au modèle réel du processeur un modèle de simulation pouvant exécuter l'ensemble du jeu d'instructions du processeur. La précision de simulation obtenue est très proche de la simulation réelle du processeur. La vitesse de simulation est également considérablement augmentée par rapport à une simulation de la réalisation du circuit décrite en VHDL. Ou bien en C

La validation complète du logiciel peut être accomplie dès le développement du logiciel embarqué et du compilateur, sans que le modèle réel du processeur ne soit nécessaire. Le processeur sera utilisé par la suite lors de la phase de son intégration dans le système. L'interface matériel-logiciel est à nouveau localisée

entre le code applicatif en assembleur et le décodeur du processeur, implémenté dans le simulateur, aussi le processeur est remplacé par un simulateur. Le jeu d'instructions du processeur définissant strictement cette interface, le comportement du simulateur, vu de l'extérieur, est parfaitement similaire au modèle réel du processeur pour peu que le modèle de simulation soit suffisamment précis.

Avantages et inconvénients

L'avantage essentiel de cette approche est qu'elle ne nécessite pas de concevoir ni de réaliser le modèle matériel réel du processeur, réduisant ainsi considérablement le temps de développement du système complet, et la date de disponibilité du logiciel embarqué. Cela est vrai à condition que le temps de développement du modèle de simulation soit nettement plus court que celui du modèle réel du processeur. Des mesures ont montré qu'un simulateur de jeu d'instructions en VHDL peut atteindre plusieurs milliers d'instructions par seconde (1-10 Mips), alors qu'un simulateur de jeu d'instructions en C peut lui atteindre une centaine de milliers d'instructions par seconde (100 Mips). De plus, la précision temporelle peut être réaliste au cycle près (cycle-true simulation).

Considérons les trois phases de la validation du logiciel décrites plus haut, et analysons l'adaptation de la co-simulation assembleur-VHDL à chacune d'elles. La validation interne, où la plus grande partie du code logiciel est exécutée, est pénalisée par la faible vitesse de simulation. Par contre, la validation de la communication avec l'extérieur requiert une précision, temporelle notamment, que seule la simulation VHDL peut assurer. Enfin la validation en environnement réel, avec le système complet, demande des temps de simulation trop élevés. Partant de la distinction entre validation fonctionnelle et validation temporelle, il apparaît que la phase 1 (validation interne) relève en grande partie de la validation fonctionnelle. Au contraire la phase 2 exige essentiellement un réalisme temporel, même si la fonctionnalité est encore importante. Enfin, la phase 3 exige une validation aussi bien fonctionnelle que temporelle. Il semble donc qu'une partie de la validation du logiciel relève de la validation fonctionnelle, ne nécessitant pas de précision temporelle au cycle près. Au contraire, elle induit une vitesse de simulation très faible. Nous en déduisons que la co-simulation assembleur-VHDL, même si elle est nécessaire pour atteindre la précision temporelle indispensable à la validation finale, n'est pas adaptée à une partie importante de la validation du logiciel embarqué, la validation fonctionnelle. Le fait d'utiliser le langage C pour la spécification et la co-simulation du logiciel devrait permettre une validation fonctionnelle efficace, appliquée plus tôt dans le flot de conception du système.

4.1.3.2 - La co-simulation de haut-niveau C-VHDL

Cette co-simulation C-VHDL fait intervenir le code C du logiciel embarqué, à la place du code machine ou assembleur. Le code C est exécuté en parallèle avec la simulation VHDL du reste du système. Elle ne requiert aucune description du processeur, mise à part la définition des signaux d'interface entre le processeur et le reste du système. La communication entre le logiciel en C et le reste du système en VHDL est assurée par un module spécifique, généré manuellement ou automatiquement.

Avantages et inconvénients

Les avantages d'une telle approche sont multiples. Avec l'application décrite en langage C, la lecture et le débogage fonctionnel sont bien plus aisés qu'avec le code binaire. De plus, cela autorise l'utilisation d'outils de développement logiciel standards. Ainsi, il n'est plus nécessaire de concevoir ni de réaliser l'architecture du processeur, ni le jeu d'instructions. De plus, le compilateur et/ou l'assembleur ne sont pas requis pour la validation fonctionnelle Leur développement peut se faire en parallèle avec la validation de l'application. Les inconvénients sont liés au fait que le niveau d'abstraction du langage C est plus élevé que pour les simulations antérieures. Seule la fonctionnalité de l'algorithme est validée, indépendamment des autres parties (processeur, micro-codage) qui seront validées lors des étapes suivantes. Les contraintes temporelles ne sont absolument pas vérifiées. La communication de données entre deux modes différents (VHDL et C) peut engendrer des problèmes de cohérence (types de données hétérogènes, absence de parallélisme en C). Ces problèmes doivent être résolus par l'adjonction de fonctions spécifiques. En résumé, la co-simulation C-VHDL permet une validation fonctionnelle et ceci très tôt dans le flot de conception du système, avant même d'avoir conçu et développé un modèle matériel du processeur embarqué. Au prix d'une précision de simulation moindre (comparée à la simulation de jeu d'instructions), le temps gagné et la facilité de développement du logiciel au niveau C font de la co-simulation C-VHDL un outil de validation fonctionnelle efficace pour la conception de systèmes complexes.

4.1.4 - Outils existants

De nombreux outils de co-simulation matériel-logiciel existent, aussi bien sur le marché que dans le monde universitaire. Les environnements complets de conception conjointe disposent également de moyens de co-simulation. Si la co-simulation assembleur-VHDL est proposée dans presque tous les environnements considérés, la co-simulation C-VHDL est plus rare.

Seamless de Mentor Graphics et EagleI de Eagle Design Automation sont deux outils de co-simulation matériel-logiciel au niveau assembleur. Seamless de Mentor Graphics fournit une interface de co-simulation pour connecter un simulateur de jeu d'instructions à un simulateur matériel. La principale caractéristique de cet outil est une optimisation des échanges entre les deux parties selon la densité de la communication. Dans le meilleur des cas, Mentor Graphics assure pouvoir atteindre une vitesse de simulation de l'ordre de 100 000 instructions par seconde.

Cossap de Synopsys et SPW de Alta Group Cadence Design Systems sont deux environnements de conception de systèmes à base de processeur de traitement du signal, et incluent la co-simulation au niveau assembleur. Ils partent d'un modèle de description du système comme un ensemble de blocs interconnectés. Certains de ces blocs peuvent être décrits en C ou en VHDL. La validation du logiciel final est effectuée grâce à la simulation du code assembleur par un simulateur de jeu d'instructions, lequel est relié à l'environnement matériel.

Enfin, Coware et Synthesia proposent deux outils de co-simulation matériel logiciel, aussi bien au niveau assembleur qu'au niveau C. Coware utilise les langages de programmation et de description classiques (C, C++, VHDL). Le système est décrit sous forme de *thread*s, c'est-à-dire un ensemble de blocs de code écrits en C, C++ ou VHDL, pouvant être éventuellement exécutés en parallèle. Les interactions entre les blocs (passage de valeurs) ne peuvent avoir lieu qu'au début ou à la fin d'un *threa*d. La partie logicielle est optimisée afin de minimiser le nombre de *thread*s. Dans le cas d'applications à contrôle complexe, un noyau temps réel peut être invoqué. Après compilation du logiciel, il est possible d'effectuer une co-simulation au niveau assembleur. En effet l'environnement Coware rend possible la connexion d'un simulateur de jeu d'instructions au simulateur VHDL. L'interface entre le matériel et le logiciel est directement disponible si le cœur du processeur est en librairie. L'environnement Coware est particulièrement adapté au développement rapide de systèmes embarqués de type traitement du signal. Les possibilités de co-simulation, aussi bien au niveau C qu'au niveau assembleur, autorisent une validation continue tout au long de la conception, et ce dans un environnement unifié. Synthesia propose un environnement de co-vérification matériel-logiciel destiné à la conception conjointe d'un système, décrit en VHDL pour la partie matérielle, et en C (ou C++ et Ada) pour la partie logicielle. Cette co-vérification intervient après le partitionnement du système, dans le but de valider celui-ci avant la réalisation finale. L'ensemble du matériel tourne sur un seul simulateur VHDL, alors que le ou les logiciels sont directement exécutés sur la station de travail. L'approche de Synthesia est parfaitement adaptée à la conception d'applications logicielles en C embarquées. La flexibilité de l'interface entre le logiciel et le matériel (un ensemble de données de types variés) permet une exploration de l'interface finale, ainsi qu'un temps de mise en place réduit. La généricité du code (aussi bien VHDL que C) disponible sous forme de *packages* ou bibliothèques facilite la mise en place de la co-simulation, et autorise la réalisation de configurations variées. Le choix du mécanisme de communication RPC permet la répartition des tâches sur plusieurs machines à travers le réseau.

4.1.5 - Conclusion

Dans cette partie on a présenté un ensemble de solutions disponibles pour la validation dans des logiciels embarqués dans un système complexe. Parmi elles la Co-Simulation C-VHDL s’avère bien adapté à la validation fonctionnelle du logiciel embarqué dans un processeur dédié. Les bénéfices de cette approche par rapport à la simulation matérielle ou la co-simulation assembleur-VHDL qui ont été constatés ont incité à intégrer cette étape de validation purement fonctionnelle au flot de conception de systèmes intégrés, en complément des validations à un plus bas niveau. En particulier, la possibilité de mettre en place un environnement de simulation du système complet avant la réalisation des processeurs est apparue comme primordiale, ainsi que l'utilisation d'outils de développement au niveau du langage C.

4.2 - Les systèmes réactifs.

L'étude des systèmes réactifs se focalise sur les problèmes liés au contrôle des interactions avec l'environnement du système. Dans les logiciels critiques, ces

interactions sont en général nombreuses et complexes; elles sont par ailleurs la plupart du temps soumises à des contraintes de temps de réponse (les systèmes dits ``temps-réel'' en sont un bon exemple). De plus, une hypothèse raisonnable consiste à considérer qu'un système réactif est censé fonctionner en permanence.

Un système réactif est ainsi généralement conçu comme un ensemble de tâches infinies parallèles qui coopèrent à la réalisation d'un comportement complexe.

Les exigences de qualité des logiciels critiques et les considérations précédentes amènent, d'une part à concevoir des langages de programmation spécifiques, et d'autre part à élaborer des modèles de comportement et des méthodes formelles de spécification et de vérification propres aux systèmes réactifs.

Dans le domaine de la programmation, les *langages synchrones* connaissent un succès important, dû essentiellement à la simplicité de leur sémantique basée sur le principe de l'hypothèse de synchronisme, qui stipule que le système réagit instantanément aux stimulations de son environnement. Ce principe permet d'obtenir des programmes compilés très performants, et également d'effectuer des vérifications formelles sur un modèle d'exécution très proche du code exécuté.

Les *systèmes de transitions finis* sont un formalisme bien adapté à la modélisation et à la vérification formelle des comportements des systèmes réactifs.

En ce qui concerne les formalismes logiques de spécification des systèmes réactifs, les *logiques temporelles* s'avèrent les plus appropriées car elles permettent de raisonner globalement sur l'évolution du programme au cours du temps.

Algar et Zheng ont fait une étude sur les logiciels réactifs à temps réel et ont estimés que les processus de validation formelles sont très difficile et coûteux pour cela ils proposent une méthode de test à boites noire pour les systèmes réactifs à temps réel dont les spécification ont été conçus d'après un formalisme basé sur les formalismes de model d'objets temporelles réactif

Dans ce qui suit on décrira deux méthodes de validation la première est destinée aux systèmes temporisé tandis que la seconde c’est une méthode de validation des systèmes réactifs repartis développée par l’équipe de PAMPA **[18]** Pour cela les systèmes de transition étiquetés *labelled transition systems* en anglais, abréviation LTS ont été utilisés.

4.2.1 Les systèmes temporisés **[6]**

Un système temporisé est un système réactif qui génère et subit des événements en respectant des contraintes temporelles bien déterminées. Il peut concerner aussi bien un protocole de transfert de flux multimédia qu’un système de contrôle de processus industriel. Plusieurs communautés s’intéressent à ce type de système.

Des modèles variés ont été proposés. Certains adoptent une approche du temps discret et d'autre préfèrent l'approche de temps continu. Le model d'automates d'Alur et Dill est celui qui a les fondements théoriques les plus solides. Ce model décrit le fonctionnement d'un système à l'aide d'un système fini d'états transitions. Un état spécifie une situation du système et une transition décrit l'émission ou la réception d'un événement. Un événement peut se produire si les contraintes temporelles qui lui sont associées sont satisfaites. Ces contraintes sont des inégalités sur des horloges spécifiques au système. Les horloges peuvent être initialisées sur une transition après l'exécution de l'action étiquetant cette transition.

Une valuation d'horloge a été définie comme une combinaison de valeurs des horloges du système. Le nombre de valuation est naturellement infini. Le système ainsi décrit possède donc un nombre infini d'états. Alur et Dill ont proposé une modélisation équivalente du point de vue fonctionnel : le model des graphes des régions. Toutes les valuations d'horloge sont regroupées dans des ensembles distincts appelés régions d'horloges. Le graphe des régions résultant est un autre système d'états transitions où un état est composé d'un couple (e,r) e : désigne un des états de l'automate temporisé tandis que r désigne la région d'horloge du système.

4.2.1.1 – le test des systèmes temporisés.

Jusqu'au début des années 90 on n'a jamais considéré formellement le facteur temps lors du test : on ne s'intéressait qu'à l'exécution des événements dans l'ordre dictée par la spécification. Depuis la fin de la dernière décennie un certain nombre d'études sur le test de systèmes temporisés a été proposé.

Toutes ces méthodes utilisent un model réduit de spécification. Souvent le model utilisé est un model de graphe des régions modifié en un model discret. Il est évident que la complexité de la génération décroît considérablement mais le pouvoir d'expression et de précision est également réduit. La plupart des méthodes de génération proposée se sont inspirées des méthodes classiques (non temporisée) ce domaine a fait l'objet de plusieurs années de recherches. Ces méthodes utilisent une machine d'états finis à entrée et sortie. Une transition y est étiquetée par un couple d'actions indissociables : une action de l'environnement et la réaction du système.

L'adaptation des méthodes classiques aux systèmes temporisés pose un problème : les délais entre l'entrée et la sortie ne peuvent être spécifiés par des contraintes temporelles.

4.2.1.2 - Génération des tests

Dans cette partie on abordera plusieurs techniques de génération de test et on présentera leurs avantages et inconvénients.

Traduction des étiquettes.

Cet méthode consiste en la traduction des étiquettes (actions du système) d'un graphe des régions en d'autres étiquettes contenant l'étiquette précédente et la région dans laquelle l'action de l'étiquette doit s'exécuter. Le résultat est une machine d'états finis avec «des actions temporelles ». Le graphe des régions étendu ainsi obtenu est ensuite transformé en machine d'états finis d'entrée et sortie. La transformation ne se fait pas en regroupant une entrée et une sortie mais selon les règles suivantes :

- A chaque action d'entrée est ajoutée une sortie fictive ε.
- A chaque action de sortie on ajoute une entrée fictive GetClock dont le rôle est de relever le rôle de la valuation des horloges (son intérêt pour le test est de vérifier que la valuation d'horloge est correcte).
- Chaque action d'écoulement du temps est transformée en sendClockVal(X) /WakeUp qui estime que le testeur demande aux horloges de s'écouler pendant x unités (temps calculé à la volée) à l'issue de laquelle les horloges envoient une sortie WakeUp.

Enfin une méthode de caractérisation des états du système Wp est appliquée sur ce graphe. Lors de l'exécution des tests la sortie fictive est ignorée mais pour l'entrée fictive il suffit de consulter la valeur de la valuation des horloges du testeur, les cas de tests ainsi obtenus doivent être interprétés pour devenir exécutable (extraire de l'étiquette d'une transition l'action à exécuter et le moment ou elle doit s'exécuter).

Cette méthode n'est appliquée que si les hypothèses suivantes sont réunis : Système déterministe, complètement spécifié, un seul état initial, à partir d'un état on ne peut avoir deux transitions étiquetées respectivement par une action d'entrée et une action de sortie.

Génération à partir d'objectifs de test

La génération exhaustive étant très coûteuse, les industriels se contente de dériver des cas de tests ciblés et qui sont guidés par les concepteurs. En fait au lieu de tester la totalité du système, on ne teste que des partie sensibles avec des cas de tests pertinents. Le but de cette partie est donc d'extraire des cas de test exécutables à partir de propriétés requises par l'utilisateur. les contraintes temporelles sur les propriétés peuvent ne pas coïncider avec les propriétés de la spécification. Pour cela deux techniques ont été développées : une technique avec des contraintes temporelles similaires entre la spécification et l'objectif et une autre avec la possibilité d'avoir une différence de contrainte entre la spécification et l'objectif du test.

Contrainte de l'objectif de test identiques aux contraintes de la spécification.

La spécification est traduite en graphe des régions, les objectif de test subissent la même opération. L'objectif de test exprime un ensemble d'opérations finies il a donc la structure d'un graphe acyclique. On génère tous les chemins partant de la racine. Chacun des chemins est recherché sur la spécification. A

chaque fois qu'on en retrouve un, on extrait le chemin (de la spécification) qui l'englobe. ce dernier fera partie des cas de test de ce système.

Contraintes de l'objectif différentes de celles de la spécification

Lorsque les contraintes temporelles des propriétés de l'utilisateur sont différentes de celles de la spécification, l'extraction des cas de test devient plus complexe. Un cas de test est généré de la spécification s'il concerne un comportement requis par l'objectif de test et existant sur la spécification. En plus de sa contrainte temporelle, le comportement de l'objectif de test doit satisfaire celui de la spécification. Pour générer ce cas de test on exécute un produit synchroniser entre la spécification et l'objectif de test. Son but est de générer une structure semblable à celle d'un graphe des régions qui correspond à l'intersection entre la spécification et l'objectif de test.

Génération exhaustive.

L'intérêt de cette partie est de générer un ensemble minimum de cas de tests permettant de garantir la conformité d'une implantation I par rapport à une spécification S. c'est une technique de caractérisation d'états sur un graphe des régions sans aucune transformation préalable. Du fait que le graphe est minimal on garantit qu'on peut trouver pour chaque paire d'états une séquence d'action qui lorsqu'elle est soumise au système à partir de chacun des états, réagit différemment. On remarque qu'à partir d'un état on peut exécuter une action d'entrée où une action de sortie où un écoulement de temps.

- Cas d'une action de sortie AS : on soumet un préambule, ensuite on observe la sortie (qui doit coïncider avec as) et ensuite on exécute un postambule.
- Cas d'une action d'entrée AE : on soumet un préambule ensuite l'entrée et on vérifie que la séquence qui suit est bien accepter par le système.
- Cas d'un écoulement du temps : on soumet un préambule on attend la durée d'écoulement du temps (calculé à la volé) ensuite on exécute le postambule adéquat.

Ce que l'on vient de voir c'est comment exécuter une séquence d'action à partir d'un état. L'opération peut être répéter plusieurs fois, s'il est nécessaire d'avoir plusieurs séquences pour distinguer un état. Les systèmes étudiés sont déterministes, ce qui implique que les séquences existent toujours. On peut noter que cette méthode est actuellement la seule qui opère sur un graphe des régions sans modifications.

Relation de conformité

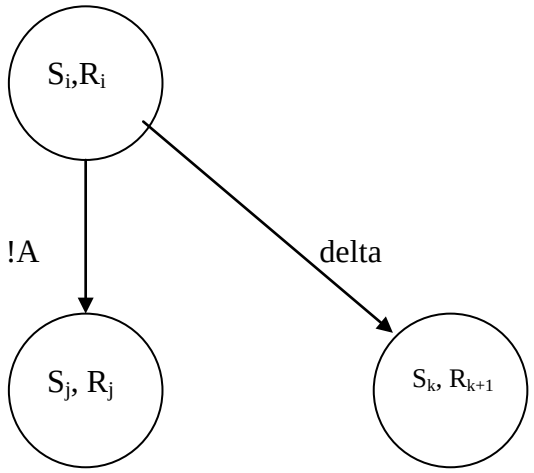
Dans toutes les méthodes de tests de conformité, il est primordial de définir une relation de conformité qui montrerait le degrés de confiance qu'on accorde à la méthode de test. La relation utilisée dans ce cas est une relation d'équivalence de traces temporisées (trace d'actions avec leurs contraintes temporelles). Deux graphes des régions sont équivalents (équivalence trace temporisé) si pour toutes les traces temporisées trouvées dans la spécification, on peut trouver une trace dans

l'implantation ayant les même contraintes temporelles. Pour prouver une telle équivalence, on doit s'assurer que : l'ordonnancement identique des actions entre la spécification et l'implantation, la satisfaction des même contraintes temporelles. Cette dernière se traduit par l'exécution de l'action dans la région horloge qui lui etait prévue. Pour une action d'entrée, il faut soumettre l'action aux deux évaluations qui sont aux bornes de la région d'horloge et pour une action de sortie vérifier qu'elle a bien été reçue à une valuation d'horloge appartenant à sa région d'horloge.

Cas de test adapté aux systèmes temporisés **[23]**

Dans cette partie, on montre que les cas de test temporisé générés par n'importe quelle méthodologie pose un problème dans l'expression des dépassements de délais dans un cas de test. En réalité toutes les méthodes gênèrent des cas de test qui indiquent ce que le système doit faire dans un délai bien déterminé au-delà de ce délai si l'action prévue est exécutée le système est déclaré non conforme. Mais dans le cas où rien ne ce passe, que doit on décider ? Que doit on faire ?

Lorsqu'on observe la structure d'un graphe des régions on remarque que :
Une action d'entrée est toujours intégrée des régions minimales comme décrit dans la figure qui suit



Le système peut renvoyer A à toute valuation de Ri mais lorsque le système passe à R_{i+1} on ne doit considérer que la transition d'écoulement du temps delta.

Une action de sortie est toujours intégrée dans un graphe des régions minimal comme décrit dans la figure précédente (en remplaçant !A par ?B) le système attend B à n'importe quelle valuation de Ri mais lorsque le système passe à R_{i+1} le système n'attend plus cette action.

De cette observation on peut conclure qu'une simple suite d'actions ne peut suffire à tester les aspects temporels et comportementaux d'un système temporisé. Pour cela on définit les deux cas de test suivants :

- Cas de test d'une action de sortie

- Cas de test d'une action d'entrée.

Cas de test d'une action de sortie

Ce test est basé sur l'attente de la réponse du système pendant la durée de R_i deux cas peuvent arriver :

- Si le testeur reçoit l'action à un instant de R_i on garantit qu'elle est correctement implémentée. Un postambule sera exécuté pour atteindre l'etat initial
- Si le testeur passe à R_{i+1} on doit vérifier que le système ne renvoie pas A. on attend pendant la durée R_{i+1}(le système ne doit rien émettre) ensuite on exécute un postambule adéquat.

Comme le système est non déterministe on ne peut pas garantir que les cas suscités se produiront mais il est raisonnable de supposer que pour un nombre élevé d'expérimentation le système couvrira l'ensemble des cas possibles. En conclusion on répète les deux possibilités de test jusqu'à ce que les deux situations se produisent.

Cas de test d'une action d'entrée.

De même on aura deux aspects à tester correspondant respectivement au test de la transition A et delta l'avantage est que l'utilisateur pourra contrôler à tous instant l'émission des actions ce qui implique que cette partie du système est déterministe. Le seul souci avec une action d'entrée est la couverture maximale de la région d'horloge R_i pour toutes ces raison il est suggéré l'utilisation des quatre cas de test suivants (il commenceront tous par un préambule) :

- Lorsque le système atteint l'état (Si, Ri) dans tous les cas on choisit la première valuation d'horloge atteinte de Ri. On soumet l'action A et on poursuit avec un postambule.
- Lorsque le système atteint l'etat (Si, Ri) à la valuation d'horloge fin (Ri) qui est la valeur finale de Ri . de même que précédemment on exécute un postambule.
- On attend que le système passe de Ri à Ri+1 on exécute tout de suite un préambule adéquat.
- On attend que le système passe de Ri à Ri+1 on exécute à la valuation fin (R_{i+1}) un préambule.

L'objectif de ces deux derniers cas de test est de vérifier que les dépassements de délais sont correctement implantés sur la durée totale du délai. L'hypothèse implicite considérée est que les horloges progressent de façon uniforme.

4.2.1.3 – Exécution et architecture de test

L'ISO a proposé une architecture de test pour les protocoles de télécommunications et est utilisée dans de nombreuses applications. Mais elle n'intègre pas l'aspect temporel durant le test.

Couverture maximale de l'espace temps **[3]**.

Plusieurs études proposent de tester un système temporisé en choisissant des moments précis. Mais la garantie de la couverture de l'espace temps n'est pas assurée. Pour résoudre ce problème on adopte le principe du test au plutôt et au plus tard. Ce principe est connu depuis bien longtemps dans la validation des systèmes à temps réel. Dans notre cas pour tester une action A 'une transition Ri→ Rj on exécute deux tests : un test pour la valuation début (Ri) et un autre pour la valuation fin (Ri)

L'originalité de cette solution concerne le calcul de la valuation début (R_i). On calcule les valuations d'horloges d'exécution des actions des préambules de façon à ce que le système arrive à exécuter l'action à tester à une valuation qui soit la plus proche du début de la région d'horloge de l'état de départ de la transition étiquetée par l'action à tester et notée début (initial(a_i)). Le calcul des valeurs des valuations V_k d'exécution de chaque action ak de la séquence d'actions précédente S= a_1,…,a_i se résout en faisant un parcours inverse de cette séquence (éventuellement jusqu'à la racine). Pour chaque entrée ak on choisit la plus grande valeur de valuation V_k (qui est inférieur à celle de V_{k+1}) comme moment d'exécution de a_k

Génération des tests exécutables multiples

Un cas de test temporisé est pertinent s'il couvre au maximum l'espace temps du système. Il est déclaré « succès » si toutes les actions qu'il contient s'exécutent correctement aux valuations extrémités de leurs régions d'horloges pour les actions d'entrée et une valuation d'horloge pour les actions de sortie. Pour un test généré ayant n actions d'entrée on extrait 2^n tests exécutables.

Ce nombre exponentiel ne peut pas satisfaire les industriels du test. Ce nombre pourrait être réduit à 2n pour tester toutes les actions du système avec une couverture temporelle maximale mais en réduisant le test des répercussions des dépendances temporelles entre actions. En résumé chaque action d'entrée est expérimentée avec la valuation de début et de fin de région d'horloges de son état de départ mais avec les valuation d'horloge des autres actions d'entrée fixées à autre valuation choisi par le testeur.

4.2.1.4 – Conclusion

Le développement des tests de système temporisés est un domaine de recherches assez récent et répond à un besoin urgent pour les concepteurs de systèmes à fortes contraintes temporelles (avionique, temps réel, multimédia…). En fait pour garantir la fiabilité et la sûreté de ces systèmes, l'utilisation des descriptions techniques formelles pour le test de conformité répond aux exigences de qualité des systèmes avec un moindre coût en détectant en amont les erreurs de conception et de fournir des outils pour tester la qualité des implantations finales qui peuvent s'écarter des propriétés requise par la spécification. La tendance actuelle est l'utilisation de models de description plus réduit que le graphe des régions. Pour réduire l'encombrement mémoire causé par les graphes des régions. Mais ces models perdent énormément en pouvoir d'expression. Ainsi les systèmes traités sont

parfois moins précis. Cependant les chercheurs préfèrent utiliser le model des graphes des régions pour garder leurs atout majeur : des fondements théoriques solides et une description certaine de description.

La nouvelle formalisation des cas de test temporisée est un résultat important. Elle donne une autre dimension aux cas de tests temporisé, ils sont plus réaliste et le facteur temps est considéré en tant qu'élément à part entière alors qu'avant il était considéré comme une action du système.

4.2.2 – Les systèmes réactifs repartis. **[14][4]**

4.2.2.1 - Définition :

Un LTS est un graphe orienté dont les arêtes appelées transitions sont étiquetés par une lettre prise dans un alphabet appelé événement

Il est usuel de parler d'automate d'états finis pour désigner un système de transitions étiqueté dont l'ensemble des états et celui des événements sont finis. Il s'agit en fait du modèle de machine le plus simple que l'on puisse imaginer. Les LTS sont employés pour modéliser des systèmes réactifs le plus souvent répartis. Dans ce cadre les événements représentent les interactions (entrées ou sorties) du système avec son environnement. On parle alors de système de transitions *entrées-sorties* ou de IOLTS (*input-output* LTS).

Ces systèmes de transitions sont obtenus à partir de spécifications de systèmes réactifs répartis décrits dans des langages de haut niveau comme SDL ou LOTOS. L'association d'un LTS à un programme se fait par l'intermédiaire d'une définition opérationnelle de la sémantique du langage et est en général formalisée sous la forme d'un système de déductions. Pour un langage aussi simple qu'une algèbre de processus (CCS par exemple), la définition de sa sémantique opérationnelle tient en moins de dix axiomes et règles d'inférences ; alors que pour un langage aussi complexe que LDS, cela est plutôt l'affaire d'un document de plus de cent pages.

Pour des raisons de performance, ces sémantiques opérationnelles ne sont jamais mises en œuvre directement ; mais font l'objet de transformations diverses. En particulier, la compacité du codage des états est un facteur déterminant de l'efficacité de la génération des LTS.

Les algorithmes classiques de théorie des langages construisent explicitement des automates d'états finis. Ils sont le plus souvent intégralement stockés en mémoire. Cependant, pour les problèmes qui nous intéressent, la construction (ou la mémorisation) exhaustive des LTS n'est pas toujours nécessaire. Une construction partielle suffit et des stratégies analogues aux évaluations paresseuses des programmes fonctionnels peuvent être employées : seule la partie nécessaire à l'algorithme est calculée.

Les calculs et transformations opérés sur les LTS se résument à des parcours et calculs de points fixes sur les graphes. L'originalité réside dans la façon de les effectuer : par calcul explicite du LTS ou bien implicitement, sans calcul ou stockage exhaustif du LTS.

Les algorithmes classiques de théorie des langages construisent explicitement des automates d'états finis. Ils sont le plus souvent intégralement stockés en mémoire. Cependant, pour les problèmes qui nous intéressent, la construction (ou la mémorisation) exhaustive des LTS n'est pas toujours nécessaire. Une construction partielle suffit et des stratégies analogues aux évaluations paresseuses des programmes fonctionnels peuvent être employées : seule la partie nécessaire à l'algorithme est calculée.

Dans le même esprit il est possible d'oublier certaines parties précédemment calculées du LTS ; et par recyclage judicieux de la mémoire, il est possible d'économiser l'espace mémoire utilisé par nos algorithmes.

La combinaison de ces stratégies de calcul sur des LTS implicites permet de traiter des systèmes de taille réelle même en utilisant des moyens de calcul tout à fait ordinaires.

Ensembles Partiellement Ordonnés

On considère qu'une exécution répartie sur un réseau de processus *I* est faite d'événements atomiques *E*, certains étant observables, d'autre ne l'étant pas. Chaque événement est l'occurrence d'une action ou opérationonsidère habituellement qu'une action a lieu sur un seul et même processus du réseau.

La relation de causalité décrit le plus petit ordonnancement partiel sur les événements que l'on peut déduire du modèle de fonctionnement du réseau de processus que l'on s'est donné. Il a été présenté pour la première fois avec comme hypothèses sur le fonctionnement de l'architecture répartie :

1. Les processus sont séquentiels. Deux événements ayant eu lieu sur le même processus sont ordonnés.

2. Les communications sont asynchrones et points à points. L'émission d'un message précède causalement sa réception.

Tout ce que l'on peut dire est que tout ordonnancement aurait respecté l'ordonnancement causal ; autrement dit, le comportement réel du système est une *extension linéaire* de l'ordre de causalité.

Il n'est cependant ni réaliste ni même utile de chercher à savoir quelle extension linéaire s'est réellement produite. Cela n'est pas réaliste car les architectures réparties existantes n'offrent pas les moyens de synchroniser des horloges locales à chaque processus avec une précision suffisante. Cela n'est pas utile car cet ordonnancement dépend de conditions d'exécution qui ne sont pas

contrôlables ou repérables. Il faut donc considérer que toute extension linéaire de la relation de causalité est un ordonnancement plausible.

La difficulté est dans la combinatoire en général exponentielle dans le nombre d'événements des extensions linéaires d'une relation d'ordre. Il existe cependant une structure intéressante pour représenter l'ensemble les états dans lequel le système a pu se trouver : c'est le treillis des antichaînes de la relation d'ordre. En terme d'exécution répartie ceci correspond à désigner pour chaque processus quel a été le dernier événement qui s'est produit ; cela définit sans ambiguïté un état possible du système. Ce treillis (distributif) peut être représenté sous la forme d'un LTS, avec comme relation de transition, la relation de couverture.

Le test de conformité est un test de type boîte noire. On se donne une spécification d'un système ouvert qui sert de modèle de référence et une implémentation réelle de ce système dont on ne connaît le comportement que par ses interactions avec l'environnement. Un test consiste à alterner la stimulation de l'IUT par des événements émis depuis l'environnement par un testeur, et l'observation des réactions de l'IUT, pouvant conduire à un verdict sur la correction ou non de l'IUT par rapport à sa spécification en fonction d'une relation de conformité. En pratique le test de conformité ne pouvant être exhaustif, les tests permettent de montrer la présence d'erreurs (non-conformité) mais ne permettent jamais de montrer leur absence (conformité). La génération automatique de test consiste à produire automatiquement des cas de tests en fonction de la spécification et de la relation de conformité choisie. La sélection d'un ensemble significatif de cas de tests peut se faire en utilisant des objectifs de test. C'est l'approche suivie actuellement lors de l'écriture manuelle des tests.

4.2.2.2 - L'outil TGV [9]

TGV (Test Generation with Verification technology) est un prototype de génération automatique de tests de conformité. TGV a été développé en commun par l'équipe Pampa et le laboratoire Verimag Grenoble et utilise des bibliothèques de la boîte à outils CADP de Vérimag et de l'Inria Rhône-Alpes. TGV est utilisé par plusieurs laboratoires sur plusieurs études de cas. Un effort important a été entrepris pour le transfert industriel des algorithmes de TGV dans l'outil ObjectGÉODE de Vérilog. Ce transfert est en cours.

Une première version de TGV a vu le jour en 95 lors d'un contrat financé par le STEI[7]. Cette première version développée par Vérimag et Pampa était utilisable sur les graphes d'états de spécifications LDS produits par GÉODE.

TGV a été étendu en 97 pour fonctionner complètement à la volée, à la fois sur des spécifications LDS grâce à sa connexion à ObjectGÉODE, et sur des spécifications LOTOS grâce à sa connexion à l'environnement CADP. TGV est donc relativement indépendant du langage de spécification puisque LDS et LOTOS sont des langages respectivement fondés sur les automates communicants et les algèbres de processus. En sortie, TGV construit un cas de test dans un format

[7] : regroupant Vérilog, Cap Sesa Région Rennes, le CNET Lannion et le Celar de Bruz, Vérimag et Pampa

général qui peut actuellement être traduit dans le langage TTCN, langage de description de tests standard de fait dans le monde des télécom.

En 98, TGV a subi deux modifications majeures. La première est l'intégration d'un nouvel algorithme central permettant de produire des tests de meilleure qualité en limitant les cas inconcluants. La deuxième concerne la prise en compte d'objectifs de tests portant éventuellement sur des actions internes. Ceci est particulièrement important pour mieux cibler les tests, en particulier pour générer des tests dans un contexte. Les objectifs portant auparavant seulement sur des actions observables, ceci a impliqué une modification importante de l'architecture de TGV, les objectifs étant pris en compte directement sur l'IOLTS de sa spécification alors qu'ils ne l'étaient auparavant que sur son comportement observable.

TGV est structuré en couches par l'intermédiaire d'API fournissant les fonctions de parcours d'IOLTS intermédiaires. D'autre part ces différentes couches utilisent des bibliothèques de stockage de graphes de CADP. La première couche spécifique au langage d'entrée permet de connecter TGV soit à l'API du simulateur ObjectGÉODE pour LDS, soit à l'API fournie par le compilateur LOTOS Caesar de la boîte à outils CADP. Cette couche fournit une API de parcours de l'IOLTS de la spécification S. Une deuxième couche utilise cette API et l'objectif de test pour fournir l'API du produit synchrone $PS = TP \times S$. La troisième couche permet le renommage et le masquage d'actions internes fournissant une API sur l'IOLTS PS_{abs} dans lequel toutes les actions internes sont indifférenciées. La quatrième effectue une déterminisation à la volée de l'IOLTS PS_{abs} les primitives de parcours de PS_{vis}. Enfin la dernière couche implémente l'algorithme central qui effectue un parcours de PS_{vis} et synthétise un cas de test *TC*.

4.2.2.3 - Algorithmique

La génération de test implémentée dans l'outil TGV utilise une algorithmique de graphe, en particulier des algorithmes de parcours. La spécificité de TGV est que ces parcours font aussi de la construction: le graphe parcouru n'est pas connu a priori mais construit pendant le parcours. Le calcul à la volée permet d'éviter la construction de certaines parties du graphe. Les parcours sont essentiellement de deux types: en largeur et en profondeur. Ces deux types de parcours ont la même complexité, linéaire dans la taille du graphe. Mais ils ont chacun leurs avantages et inconvénients qui font qu'ils sont adaptés ou non à certaines parties de la génération de tests. Le parcours en largeur est utile pour calculer des plus courts chemins. Par contre il ne permet pas de détecter de boucles. Il est utilisé pour calculer des postambules de test permettant le retour à l'état initial. Inversement le parcours en profondeur permet de détecter des boucles mais ne permet pas de calculer de plus court chemin. Il est aussi très adapté pour synthétiser des sous-graphes, la pile de parcours contenant une séquence depuis l'état initial. Enfin il est très adapté au calcul à la volée, certaines parties du graphe pouvant être coupées au *backtracking*.

4.3 - Test de logiciels à flot de données synchrones

En se basant sur le langage Lustre, des travaux théoriques avaient permis de définir une stratégie de test appropriée aux langages à flot de données synchrones. Cette stratégie est basée sur un test statistique structurel complété par un test déterministe des valeurs limites/spéciales. Elle s'applique à chacune des étapes d'un processus de test progressif : test unitaire et test d'intégration. Le critère de test utilisé est la couverture des transitions d'un automate (complet au niveau unitaire, puis simplifié pour le test d'intégration) produit par le compilateur Lustre. La définition des différents niveaux de test s'inscrit dans le cadre d'une démarche globale montante, qui est guidée par un souci d'optimisation de l'effort de test et de minimisation du coût total. Le critère d'arrêt pour un niveau donné est basé sur la complexité (nombre de transitions) de l'automate à analyser pour définir les entrées de test.

Les travaux qui on démarré en début de 1997 au sein du groupe TSF sur Le test de logiciels synchrones ont permis d'améliorer les algorithmes de sélection des nœuds (sous-programmes Lustre) retenus aux différentes étapes de test (unitaire, intégration). Ce qui a induit à la mise en œuvre ces nouveaux algorithmes sur une étude de cas fournie par Schneider Electric. Il s'agit d'un système d'instrumentation neutronique du niveau source dans un réacteur nucléaire. Cette application a été développée dans l'environnement SCADE commercialisé par Vérilog. Pour ce programme, formé de 36 nœuds Lustre, 4 étapes de test ont été définies - test unitaire, suivi de trois niveaux de test d'intégration; et les jeux de test ont été générés : ils permettent de tester 30 des 36 nœuds. La complexité des 6 nœuds restant ne permet plus de baser la génération des tests sur l'analyse des automates Lustre (plus de 130 états) : pour ces nœuds, un modèle plus amont (planches graphiques SCADE) a été utilisé comme référence pour définir une stratégie de test et générer des jeux de test fonctionnel.

Ce travail s'est terminé par une étude expérimentale visant à évaluer ces jeux de test en terme, d'une part, de pourcentage des branches exécutées dans le code source C généré automatiquement par l'outil SCADE (2600 lignes) et, d'autre part, d'analyse de mutation. En ce qui concerne les branches, les mesures ont été effectuées avec l'outil Logiscope : une couverture de 100% a été observée pour tous les nœuds. L'analyse de mutation, réalisée avec l'outil SESAME, a porté sur un sous-ensemble de trois nœuds (deux au niveau du test unitaire, et un en test d'intégration). L'analyse de mutation consiste à injecter des fautes (appelées mutations) dans le programme et à mesurer l'efficacité d'un jeu de test en terme de pourcentage de mutations qu'il permet de révéler. Sur un total de 220 mutations, une seule n'est pas révélée par le jeux de test. Ces résultats confirment l'efficacité de la stratégie et la complémentarité des entrées de test statistique et déterministe.

D'autres travaux sur la génération probabiliste d'entrées de test ont débouché sur la définition des tests statistiques structurel ou fonctionnel qui reposent sur le principe suivant : les entrées de test sont générées aléatoirement selon une distribution des probabilités sur le domaine d'entrée (appelée profil de test) qui utilise

l'information fournie par le critère retenu. Deux autres approches probabilistes ont été proposées par ailleurs.

Le test aléatoire est basé sur une distribution uniforme sur le domaine d'entrée : son caractère aveugle explique son manque d'efficacité. Le test basé sur un profil opérationnel est généralement utilisé dans un objectif d'évaluation de fiabilité. Pour la recherche de fautes, il présente des limites, telles que la difficulté de définir un profil opérationnel significatif au niveau d'un sous-système, ou encore l'inefficacité vis-à-vis de fautes affectant des fonctions critiques dont la probabilité d'activation est faible en vie opérationnelle (exemple : fonction d'arrêt d'urgence).

4.4 - Test de logiciels orientés objet

Vis-à-vis de la vérification, et en particulier du test, la technologie orientée-objet pose de nouveaux problèmes liés à plusieurs de ses spécificités, telles que : (i) l'architecture éclatée qui caractérise les applications orientées objet, la forte interaction entre objets créée par les liens statiques et dynamiques induits par les relations d'héritage, de polymorphisme, d'agrégation et d'association (relations client-serveur), et l'encapsulation qui ajoute des problèmes de commandabilité et d'observabilité au niveau des objets. Pour résoudre ces problèmes, il faut, d'une part, proposer une approche pour définir les étapes de test (unitaire, intégration) radicalement différente de l'approche classique; relative aux programmes procéduraux et, d'autre part, définir de nouveaux critères de test basés sur des modèles structurels et fonctionnels adaptés à la technologie orientée objet.

En ce qui concerne la définition de critères de test basés sur des modèles adaptés aux logiciels orientés-objet, une première étude avait porté sur une stratégie de test statistique incrémentale ciblant les mécanismes d'héritage et de polymorphisme. C'est ensuite la problématique du test des interactions (relations client-serveur) entre objets concurrents qui a été au cœur des activités de recherche. L'aspect concurrence a conduits à introduire le temps dans la définition de séquences de test, pour prendre en compte le non-déterminisme dû aux communications asynchrones entre objets. L'enjeu consiste à n'exclure pendant le test aucun entrelacement des événements qui peut se produire en vie opérationnelle. Ainsi, pour chaque séquence d'événements à envoyer, des intervalles de temps entre événements doivent être générés selon plusieurs profils de charge qui simulent différents contextes opérationnels. Une technique d'échantillonnage basée sur trois profils caractéristiques a été proposée: des délais longs comparés au temps de réaction du sous-système testé ; des délais courts ; un mélange de délais longs, courts et moyens. Cette technique est intégrée dans une stratégie de test unitaire et d'intégration basée sur l'exploitation des informations contenues dans les dossiers de conception orientée-objet.

Au niveau unitaire, l'accent a été mis sur le test de robustesse dans un objectif de réutilisation, en vérifiant le comportement de petits sous-systèmes interagissant avec un environnement non contraint. L'approche se base sur les modèles comportementaux générés par Fusion : d'une part le modèle du cycle de vie, qui définit les séquences valides d'acceptation et d'émission de messages pour chaque unité ; d'autre part le modèle des opérations, qui précise les traitements des

messages sous forme de pré- et post-conditions logiques. L'environnement de test développé permet de vérifier, par rapport à ces modèles, le comportement des différentes unités soumises à des séquences aléatoires de messages. Outre les éléments de commande et d'observation liés aux échanges de messages avec l'unité, il inclut, lorsque c'est possible, des observateurs sur les post-conditions des opérations. Les expériences de test statistique ont mis en évidence, pour chaque unité, des comportements défaillants liés à la synchronisation avec le simulateur. Indépendamment de ces problèmes, la non conformité au cycle de vie a pu être observée dans le cas de deux unités. Pour l'une d'entre elles, ces expériences ont montré que des hypothèses restrictives sur son environnement d'utilisation sont nécessaires pour assurer le respect de son cycle de vie : cette unité ne peut donc pas être réutilisée dans un contexte quelconque. Des expériences de test d'intégration ont ensuite été faites sur des couples d'unités, puis au niveau de l'application complète (test système). Elles ont permis de conclure que, dans l'environnement plus contraint de la cellule de production, les dysfonctionnements observés unitairement ne peuvent plus se produire car les hypothèses identifiées au niveau unitaire sont respectées. Enfin, dans tous les cas, il a été confirmé que le comportement observé dépend fortement de la façon dont les messages envoyés s'entrelacent avec les traitements des unités, justifiant ainsi l'utilisation de plusieurs profils de charge pour la recherche de fautes.

La tolérance aux fautes

Le développement d'applications tolérantes aux fautes nécessite la définition et l'intégration de mécanismes permettant de mettre en œuvre des caractéristiques méta fonctionnelles (redondance, authentification, chiffrement). L'utilisation de techniques de *développement orienté objet* présente de multiples intérêts, tant au niveau de la conception que de la mise en œuvre. C'est en particulier la propriété de réflexivité de certains langages à objets qui a été mise à profit car elle permet d'observer et de contrôler le fonctionnement interne des objets de manière externe aux objets eux-mêmes. Cette propriété peut se mettre en œuvre en utilisant la notion de *protocole à méta objets*. Un protocole à méta objets permet de lier les objets de l'application (notion de niveau de base) à des méta objets (notion de méta niveau) qui en assurent le contrôle.

La notion de protocole à métaobjets présente un triple avantage. Elle apporte conceptuellement une séparation claire entre l'application elle-même et les mécanismes mis en œuvre à des méta niveaux différents. À chaque méta niveau, il est alors possible de définir les mécanismes en utilisant les propriétés classiques des langages à objets, telles que la spécialisation par héritage, et donc en favoriser la réutilisation. Enfin, l'utilisation de ces mécanismes peut se faire de façon transparente pour les applications.

Cette approche a donné lieu à la conception et à la réalisation d'une architecture répartie permettant le développement d'applications mêlant des mécanismes de réplication, d'authentification et de chiffrement. Le prototype FRIENDS a permis de valider ces idées et de mesurer les performances d'une telle architecture. Les propriétés, les limites et les performances de FRIENDS ont été évaluées sur une application répartie de type bancaire. Les propriétés attendues de

transparence vis-à-vis des applications, de séparation et de composition des mécanismes ont pu être obtenues bien que le protocole à métaobjet utilisé possède certaines limites (conventions de programmation, pas de gestion de l'héritage). En ce qui concerne les performances temporelles, il a été montré qu'un protocole à méta objet ne pénalisait pas de façon significative les performances du système.

Pour lever les limites observées et pour rendre le système compatible avec les standards actuels de développement de systèmes répartis, nous avons développé un protocole à méta objet spécifique. En effet, cette approche est aussi attractive dans un environnement d'exécution CORBA, base actuelle de développement de nombreux systèmes répartis industriels. Elle permet de mettre en œuvre des applications tolérant les fautes sur une telle plate-forme de façon non intrusive. Tout ORB (*Object Request Broker*) du commerce peut donc être utilisé.

Le protocole à méta objet qui a été développé permet un contrôle homogène de la création, de la destruction et de l'invocation de méthodes. Grâce à sa définition sous forme d'interfaces CORBA, il permet l'association dynamique de métaobjets avec les objets de l'application. L'une de ses caractéristiques essentielles est aussi de capturer dynamiquement l'état des objets de l'application.

L'utilisation de la réflexivité *à la compilation* constitue l'originalité du développement de ce protocole. Cette technique consiste en effet à appliquer des règles de transformation de programme sur le code source des objets qui permettent de mettre en œuvre un protocole à métaobjet à l'exécution. Le compilateur du langage d'interface (IDL) a été utilisé pour effectuer l'empaquetage des arguments échangés entre les objets.

Il est important de souligner que la réflexivité à la compilation a, en particulier, permis de traiter le problème de la capture de l'état d'objets CORBA. En effet, lorsque le support d'exécution du langage ne fournit aucune information sur la structure des objets, ce problème ne peut être traité qu'au niveau du langage. Ces travaux s'apparentent aux techniques de sérialisation d'objets Java (obtention de l'état d'un objet grâce à l'interface réflexive de la machine virtuelle Java). Le méta compilateur Open C++ V2 a été utilisé pour les expérimentations de cette approche sur des exemples simples.

Les travaux effectués ont conduit à la définition de l'architecture d'un système CORBA réflexif tolérant les fautes. Pour faciliter le développement de cette architecture, des outils UML ont été utilisés pour la conception du méta niveau et des protocoles répartis de tolérance aux fautes. Une modélisation du protocole à méta objet a été effectuée pour définir une stratégie de test propre à ce type d'architecture.

4.5 - La tolérance aux fautes dans les systèmes commerciaux critiques. [21]

Les impératifs économiques militent en faveur de la réutilisation de composants et surtout de composants sur étagère ou " COTS " (" *commercial off-the-shelf* "), tant au niveau matériel qu'au niveau des logiciels exécutifs. Le problème du surcoût temporel d'une mise en œuvre logicielle de la tolérance aux fautes persiste,

mais est atténué par l'accroissement spectaculaire des performances des processeurs modernes.

Le principal problème de l'utilisation, dans un système critique, de composants COTS à usage général, est que leur processus de développement est régi par les contraintes de " temps jusqu'au marché " et donc peu apte à l'instauration de la confiance vis-à-vis de l'absence de fautes de conception résiduelles ou d'un comportement bien défini en présence de fautes, internes ou externes. Cet aspect est pris en compte, d'une part, par les possibilités de diversification et de partitionnement de l'architecture modulaire, et d'autre part, par l'étude de **mécanismes d'empaquetage** (" *wrapping* ") de systèmes opératoires du commerce visant à confiner les erreurs. Les approches classiques à l'empaquetage ne considèrent que des techniques de filtrage des entrées et des sorties. Dans ce type d'approche, il s'agit de définir en fait des tests d'acceptation de valeur d'entrée ou de sortie sans avoir une connaissance précise du comportement du COTS. C'est la seule approche pour des composants du commerce " boîte noire ", ce qui n'est pas en fait l'hypothèse de travail pour des applications critiques. En effet, l'utilisation d'exécutifs commerciaux se fait en règle générale en partenariat étroit avec les fournisseurs et souvent avec une mise à disposition du code source. Une nouvelle approche a été proposée par les chercheurs du groupe TSF, elle repose sur la notion d'assertions exécutables bâties sur des modèles du comportement du noyau. Elle induit une approche réflexive, approche originale qui tranche avec les approches " boîte noire " classiques. Bien qu'une connaissance intime du composant ne soit pas nécessaire pour établir les modèles, il est nécessaire d'accéder à des données ou à des événements internes du composant pour pouvoir mettre en œuvre les assertions exécutables correspondantes. Cette approche est adaptable aux besoins de l'intégrateur du système et compatible avec les intérêts du fournisseur de COTS.

Au niveau de la mise en œuvre d'algorithmes et de mécanismes de tolérance aux fautes, l'utilisation d'une approche objet a pour avantage la propriété de réflexivité de certains langages à objets ou, plus exactement, l'utilisation de protocoles à métaobjets, permet de rendre les mécanismes de tolérance transparents vis-à-vis des applications et apporte beaucoup de flexibilité par la possible réutilisation des applications et des mécanismes. Dans ce contexte ont été développé le prototype FRIENDS et un protocole à méta objet pour la mise en œuvre d'applications tolérantes aux fautes sur CORBA. Les travaux effectués dans FRIENDS ont montré tout l'intérêt d'une architecture réflexive par rapport à des approches à objet classiques reposant uniquement sur l'héritage tel que dans les projets AVALON/C++ ou ARJUNA. La flexibilité de ce type d'architecture, la facilité de composition de mécanismes ainsi que les performances temporelles ont été étudiées en profondeur par rapport à des systèmes tels que MAUD ou GARF dans lesquels les aspects réflexifs sont plus limités. L'intérêt de cette approche a ensuite été étudié dans le contexte CORBA en essayant de combiner les avantages de ce modèle d'architecture avec les notions de réflexivité.

On peut distinguer trois approches suivies jusqu'alors pour mettre en œuvre la tolérance aux fautes dans des architectures CORBA. D'abord, l'approche par intégration, qui consiste à inclure des mécanismes dans le courtier des requêtes d'objet ou " ORB " (*Object Request Broker*). Cette approche conduit à des ORB

spécifiques et n'est donc pas envisageable lorsque l'on souhaite utiliser des ORB du commerce. La deuxième approche consiste au contraire à développer des services CORBA de tolérance aux fautes sur un ORB standard. Ces services sont utilisables pour la mise en place de stratégies de tolérance aux fautes par les développeurs d'applications. Le principal inconvénient de cette approche est son manque de transparence. La troisième approche consiste à intercepter les communications entre les ORB pour les dérouter vers des composants assurant la tolérance aux fautes. Elle offre la transparence au niveau application, mais est cependant dépendante du système opératoire sous-jacent car le niveau d'interception est très bas (liaisons TCP).

L'originalité de cette approche repose sur une notion d'un empaquetage (" *wrapper "*) bâti sur un modèle du comportement des fonctions internes du micronoyau en l'absence de fautes. Parce que les fonctionnalités du micronoyau sont simples au niveau des spécifications, il est possible de définir des prédicats (ou des pré- et post-conditions) sur leur fonctionnement, à partir de ces modèles. La modélisation peut reposer soit sur une expression formelle du comportement attendu de la classe de fonctionnalité fournie par l'exécutif, soit sur une modélisation comportementale de son fonctionnement. Dans les deux cas, ces modèles sont établis à partir des spécifications. Dans le premier cas, cependant, on obtient une efficacité supérieure alors que dans le second, l'efficacité repose sur le niveau d'abstraction du modèle. La complexité de ce modèle du comportement a une incidence forte sur les performances temporelles.

Les empaquetages ont été développé en suivant cette approche pour le module de synchronisation et pour celui d'ordonnancement. La technique utilisée pour ces deux exemples illustre deux formes différentes pour la définition des empaquetages. Dans le cas de la synchronisation par sémaphore, il est très facile de définir une expression formelle qui lie le nombre d'opérations de prise et de relâchement de sémaphore avec le nombre de tâches bloquées en attente de libération de ressource. Ainsi, un contrôle fin peut être effectué sur le comportement en présence de faute : par exemple, la détection d'un inter-blocage lié à l'occurrence d'une faute. Pour ce qui concerne l'ordonnancement des tâches, nous avons établi un modèle générique de l'ordonnanceur qui lie les tâches bloquées en attente d'exécution dans différentes files d'attente du noyau et la tâche qui s'exécute. Ce modèle est générique dans la mesure où seule la politique d'élection de la tâche active diffère entre différentes politiques d'ordonnancement classiques. Les transitions de l'automate correspondant sont dues non seulement à des appels de fonctions du noyau par les applications (création ou suspension de tâche) mais aussi à des événements internes qu'il est nécessaire de capturer.

La mise en place de tels empaquetages nécessite donc l'interception des appels noyaux, l'interception de certaines fonctions internes ainsi que la visibilité de certaines structures de données. Ce constat a conduits à implémenter ces empaquetages en s'appuyant sur une approche réflexive. La propriété de réflexivité permet en effet de fournir le niveau d'observabilité et de commandabilité nécessaire pour mettre en place les empaquetages. Ces travaux ont conduit à la définition de la notion de *métanoyau* (pour l'exécution des

empaquetages de confinement d'erreur) et de *métainterface* (pour l'accès aux informations internes au noyau). Ces deux notions conduisent à la notion de micronoyau réflexif, permettant ainsi une encapsulation efficace. L'intérêt de ces empaquetages a été illustré au moyen de l'injection de fautes. À titre d'exemple, l'utilisation combinée de l'empaquetage pour la synchronisation et pour l'ordonnancement permet de ramener la proportion de fautes affectant l'application à une valeur inférieure à 0,5%, tant pour les fautes externes qu'internes au noyau.

4.6 – Conclusion

Dans ce chapitre on a présenté les différentes approche qui sont utilisée pour assurer les fonction de fiabilité et de sûreté de fonctionnement, bon nombre de ces méthodes est utilisé à grande échelle tandis que d'autres ne sont toujours que des méthodes académiques qui sont au stade d'essai et d'étude. cependant il existe certains systèmes qu'on ne peut pas classer dans l'une des catégories précédentes car il appartiennent à deux classe de logiciels voir même trois, on peut citer comme exemple les systèmes informatiques incrustés à échéances temps réel strictes ce sont des systèmes embarqué, temporisé voir même orienté objet pour ce type de systeme il n'est pas raisonnable de les faire passer par les trois types de test (probleme de cout et de temp) pour cela on adopte une approche spécifique à l'application de façon à cibler les points les plus critiques ou bien on adoptera une approche de tolérance aux fautes. L'approche proposée s'appuie sur une architecture modulaire configurable selon trois axes: le nombre de canaux (zones de confinement primaires), le nombre de voies (processeurs de traitement) à l'intérieur de chaque canal (zones de confinement secondaires) et le nombre de niveaux d'intégrité qui peuvent être gérés. Les deux dimensions de redondance physique sont gérées par logiciel afin d'autoriser l'utilisation de la diversification (matérielle et/ou logicielle). La mise en place et la gestion de niveaux d'intégrité font appel aux recherches sur la cohabitation de logiciels de criticités multiples.

Sur le plan de la conception de l'architecture pour les aspects liés à la validation), les recherches ont porté d'une part, sur la définition générale de l'architecture et, d'autre part, sur les aspects système ou algorithmiques suivants :

- Les techniques de structuration et d'ordonnancement préemptif de tâches temps réel répliquées permettant de minimiser la jigue des entrées-sorties redondées et d'assurer le déterminisme des traitements en l'absence de faute (sans recours systématique à l'exécution d'un protocole de cohérence interactive) ;

- Les algorithmes de filtrage d'erreurs (destinés à assurer qu'un canal n'est accusé qu'en cas d'erreurs répétitives) et de consolidation des syndromes d'erreur (pour que chaque canal arrive à un même diagnostic de fautes) ;

- Les algorithmes de restauration de l'état de tâches répliquées s'exécutant sur un canal lors de sa réinitialisation pendant le traitement de fautes.

CHAPITRE 5 :

La méthode proposée

CHAPITRE 5 : La méthode proposée

5.1- Introduction.

La phase de test est l'une des phases les plus importantes du cycle de vie d'un logiciel, elle intervient afin de valider le logiciel avant sa mise en œuvre. Plusieurs méthodes et approches ont été développées pour les tests de logiciels. Le choix de ces méthodes dépend principalement du type de l'application. Cependant plusieurs problèmes restent à solutionner tel que les tests d'interactions parallèles, les problèmes d'explosions combinatoires des sequences de cas de tests, l'absence de notion de terminaison,…..etc. Ce rapport traite de ces problèmes et propose des solutions, particulièrement pour le cas des logiciels réactifs à temps réel.

Le terme réactif a été introduit par Harel et Pnueli pour designer des systèmes qui interagissent continuellement avec leurs environnements. Deux propriétés importantes caractérisent ces systèmes réactifs : la synchronisation des stimulants et la synchronisation des réponses. Ces systèmes sont nécessaires, de part leur importance, pour le traitement d'applications exigeant des temps de réponse immédiats, tant du coté interactions externes que du coté interactions internes du système. Comme exemples de ces systèmes le système de monitoring d'un patient ou un chien de garde d'une centrale nucléaire. La conséquence d'une défaillance d'un tel système est extrêmement grave. Les vérifications et les tests doivent alors être conduits à partir des premières étapes de la spécification et de la conception afin de s'assurer d'une implémentation correcte du coté sécurité (fiabilité) et du comportement dépendant du temps.

Plusieurs méthodes ont été développées dont l'approche d'Algar & Zheng, l'approche Nielsen et l'approche de Wesley. Aucune de ces approches ne traite complètement des problèmes posés au niveau des tests comme la définition d'une architecture adaptée aux systèmes réactifs à temps réel.

5.2- Etat de l'art

L'étude des systèmes réactifs se focalise sur les problèmes liés au contrôle des interactions avec l'environnement du système. Ces interactions sont la plupart du temps soumises à des contraintes de temps de réponse (les systèmes dits ``temps-réel'' en sont un bon exemple). De plus, une hypothèse raisonnable consiste à considérer qu'un système réactif est censé fonctionner en permanence. Un système réactif est ainsi généralement conçu comme un ensemble de tâches infinies parallèles qui coopèrent à la réalisation d'un comportement complexe. Les exigences de qualité des logiciels et les considérations précédentes amènent, d'une part à concevoir des langages de programmation spécifiques et d'autre part à élaborer des modèles de comportement et des méthodes formelles de spécification et de vérification propres aux systèmes réactifs.

Plusieurs méthodes de tests des systèmes non temporels à boites noire existent. Ces méthodes différent dans le choix du formalisme utilisé pour la formulation des spécifications des besoins et dans les techniques utilisées pour la génération des cas de tests utilisés à partir des spécifications formelles. La méthode de génération

de test Weyuker's **[27]** utilise des propositions booléenne pour la spécification des besoins. Donat a étendu la méthode weyuker's en utilisant des prédicats logiques pour exprimer les spécifications. Ces notations manquent de puissance d'expression pour spécifier formellement les différents états des données et pour la formulation les cas de test basés sur ces même données. En conséquence ils ne conviennent pas pour le test d'une large gamme des systèmes logiciels **[21]**.

Les méthodes de spécification algébrique sont bien adaptées à la spécification des systèmes orientés objet cependant le style algébrique pur **[28]** ne supporte pas la spécification de l'information d'états. A cause de ses limitations un modèle basé langage qui peut spécifier les changements d'états est généralement utilisé dans les langages de spécification pour les tests à boite noire. Il existe des méthodes rigoureuses pour générer les cas de test à partir des langages de spécifications Z et VDM. Periyasamy a étendu les méthodes Stocks et Carrington pour générer des cas de tests à partir des opérations composites dans les objets Z. Ceci afin de tester la conformité des programmes orientés objet. L'extension temporelle des langages de spécification Z et objets Z a été récemment introduite mais il n'existe pas encore de méthodes formulées pour les utilisés dans la génération des cas de tests pour les systèmes temporisés**[36]**.

Dans le domaine de la programmation, les *langages synchrones* connaissent un succès important dû essentiellement à la simplicité de leur sémantique **[27].** Ce qui permet d'obtenir des programmes compilés très performants et également d'effectuer des vérifications formelles sur un modèle d'exécution très proche du code exécuté. Les *systèmes de transitions finis* sont un formalisme bien adapté à la modélisation et à la vérification formelle des comportements des systèmes réactifs.

En ce qui concerne les formalismes logiques de spécification des systèmes réactifs **[18]**, les *logiques temporelles* s'avèrent les plus appropriées car elles permettent de raisonner globalement sur l'évolution du programme au cours du temps. En général il n'y a pas trop de travaux qui traitent des tests rigoureux des systèmes temporisés. Récemment springintveld, vaandrager et D'Argenio**[25]** ont proposé une méthode de test à boite noire destinée aux systèmes à temps réel. Cette approche est basée sur les automates d'entrées sorties temporisés. Petitjean et Fouchal ont optimisé cette méthode**[25]** en supprimant certaines modélisations jugées non nécessaires. Alagare et Zheng ont proposé une méthode de test à boite noire**[26]** pour les systèmes réactifs à temps réel en utilisant un formalisme de spécification orienté objet.

Si ces méthodes ont permis d'améliorer théoriquement les tests à boite noire pour les systèmes réactifs à temps réel, elles demeurent neamoins difficilement implementables tel que l'approche de test de Fouchal. D'autres approches ont tenté d'utiliser les automates temporisés. Un automate temporisé (AT) est un graphe représentant le système temporisé pendant son exécution**[30]**. Pour modéliser le temps dans un système temporisé un ensemble d'horloge est alors associé à l'automate. Chaque horloge est représentée par une valeur réelle non négative qui s'incremente de façon strictement monotone. A l'état initial, toutes les horloges sont mises à zéro. (Les horloges peuvent être remises à zéro à n'importe quelle transition) Pour exécuter une action étiquetée par une transition, toutes les horloges

du système doivent satisfaire la contrainte de transition. Une variante de cette approche utilise les automates temporisés à entrée sortie **[29]** qui sont une extension des automates temporisés dans lequel les événements sont partitionés en entrée et sortie.

Si on tente de modéliser un système à temps réel avec un tel modèle l'automate temporisé sera grand et complexe. Il sera impossible de construire un tel automate en pratique. De plus l'approche A.T n'est pas orientée objet et par conséquent les instances d'un modèle ne peuvent pas être insantié avec un tel formalisme.

Le formalisme TROM utilise le concept d'objet pour modéliser les systèmes réactifs à temps réel **[6,11]**. Ce qui permet de modéliser les différentes instances des objets et les interactions entre les différents objets, tout en modélisant les contraintes de communication entre ces mêmes objets. Chose qui n'est pas aisée à réaliser avec les autres formalismes (IOTA, réseaux de pétrie….). La méthode d'Algar et Zheng est la seule méthode de test validée et basée sur ce formalisme. De ce fait elle offre la possibilité de modéliser les systèmes réactifs temps réel de façon souple et complète sans pour autant générer des problèmes d'explosions combinatoires. Cependant cette méthode possède certaines limites dues à la définition même du formalisme TROM. D'où l'intérêt de notre travail qui consiste à étendre cette méthode à fin de prendre en charge la communication entre les objets d'une même classe pour tester cette interaction.

Dans la section suivante on présentera l'approche d'Algar et Zheng. la section 3 présentera l'extension proposée à cette méthode. Une architecture de test adaptée à la méthode étendue est présentée dans la section 4 et est complétée par un model de faute à la section 5.

5.3 – Description de la méthode d'Algar et Zheng [26]

La méthode de test d'Algar et Zheng est une méthode de vérification des systèmes réactifs temps réel qui utilise un ensemble de tests rigoureux qui complète un processus de vérification formelle. Elle se base sur la génération de cas de tests à partir de la spécification pour tester la conformité de l'implémentation vis à vis de la spécification formelle du système.

C'est une méthode qui offrira ; après une vérification formelle des propriétés de sécurité ; une protection contre les erreurs dues aux contraintes de ressources qui pourraient interdire le comportement temporel dans certains cas dans l'environnement opérationnel du logiciel.

La méthode d'Algar et Zheng **[26]** que nous décrivons ci-après car intéressant directement notre approche utilise le formalisme TROM. Toute spécification est ensuite transformée en un automate à grille à partir duquel sont extraits les cas de tests en utilisant une couverture d'états et de transitions. Dans le cas où un système est complexe il est partionné en sous systèmes, des cas de test sont générés pour chacun d'entre eux. Nous illustrons ce modèle à travers le problème de traversée d'un chemin de fer. Ce problème est considéré comme le

bench mark de référence des chercheurs de la communauté des systèmes à temps réel.

TROM est un formalisme orienté objet introduit dans **[35]** pour la modélisation des systèmes réactives à temps réel. Bien que la sémantique TROM est basée sur un système temporisé à transitions étiquetées il existe une différence significative entre TROM et TA et entre TROM et TIOA.

Dans le formalisme TROM une entité réactive est formalisée comme une classe réactive générique encapsulant la structure et les fonctionnalités dépendante du temps des objets de la classe. Tous les objets de la classe réactive ont le même comportement.

La classe réactive peut inclure les attributs du type de base ou du type de données abstraites. Plusieurs objets peuvent être instanciés à partir de la classe réactive. Il existe d'autres différences significatives qui sont **[32]** :

- Une classe est paramétrée avec des types de port. Un objet instancié à partir de cette classe peut avoir un nombre fini de ports de n'importe quel type. Les objets ayant le même comportement peuvent interagir dans différents contextes.

- Si un événement e déclenche un événement f et P(t) est le prédicat de contrainte de temps associé à f alors la sémantiques requière que l'événement f se produise pendant l'intervalle de temps durant lequel p(t) est vrai sinon l'objet entre dans l'état désactivé et l'événement f est désactivé.

- Deux objets dans le système communiquent par échange de messages à travers leurs ports compatibles. A n'importe quels moments quelques objets du système peuvent interagir entre eux tandis que d'autres peuvent être occupés par des activités internes. En conséquence il y a un parallélisme dans le système composé d'objets TROM.

- Les événement sont partitioné en entrée (?), sortie (!) et événement internes. Un événement interne est initialisé et contrôlé par un objet et n'a pas d'effet sur les autres objets du système. d'où il est décrit d'une manière plus explicite dans le formalisme TROM que TA et TIOA.

- Un événement entrant n'a pas de contraintes temporelles. Un événement peut se produire durant un état il cause un changement d'état.

5.3.1 - Sémantique opérationnelle.

Dans un système, les objets réactifs communiquent par echange de messages. Un message transmis d'un objet à un autre dans le système est appelé signal et il est représenté par le tuple (e_i, p_i, t_i) dénotant qu'un événement e_i se produit à un temps t_i sur un port p_i . l'état d'un TROM à tout moment t_i , est le tuple (s, $a^{\rightarrow}$, r) tel que l'état courant s est un état courant simple du TROM $a^{\rightarrow}$ est le vecteur d'assignement et r est le vecteur des réactions. L'étape de calcul du TROM se produit lorsqu'un objet ayant l'état (s, $a^{\rightarrow}$, r) reçoit le signal (e_i, p_i, t_i) et qu'il

existe une spécification de transitions qui peut changer l'état du TROM. Le calcul c d'un objet TROM A est la séquence qui peut être infinie d'alternance d'états et de signaux
$OS_0 \xrightarrow{(ei, pi, ti)} OS_1 \xrightarrow{(ei, pi, ti)} \ldots\ldots$ le calcul système réactif peut ne pas se terminer par conséquent le calcul est généralement une séquence infinie.
Le comportement du TROM est l'ensemble de tous les calculs de l'objet TROM A notés par Comp(A). Le calcul d'un sous système est une séquence infinie de séquences d'états systèmes et de signaux qui affecte le changement d'états. Le comportement du système réactif est l'ensemble de tous les calculs du produit synchronisé de la machine correspondant aux objets du système.

5.3.2 - Problème de traversée d'un chemin de fer.

Plusieurs trains traversent des passages à niveaux de façon indépendante et simultanée. Le train traverse le passage entre 2 et 4 unités de temps après avoir envoyé le signal au contrôleur. Ce dernier est également informé lorsque le train quitte le passage 6 unité après avoir envoyé le premier message. Le contrôleur lancera la procédure de fermeture de la barrière une unité de temps après avoir reçu le message d'approche du premier train. Il devra surveiller ses écrans au cas ou il recevra un message d'approche provenant d'un autre train. Le contrôleur ordonnera l'ouverture de la barrière une unité après avoir reçu le message du dernier train qui a quitté la barrière. La procédure de fermeture de la barrière durera une unité de temps. La barrière doit se fermer une unité de temps après le signal du contrôleur et doit s'ouvrir 1 à 2 unités après avoir reçu le signal du contrôleur. La figure 1 montre la spécification du système train contrôleur barrière

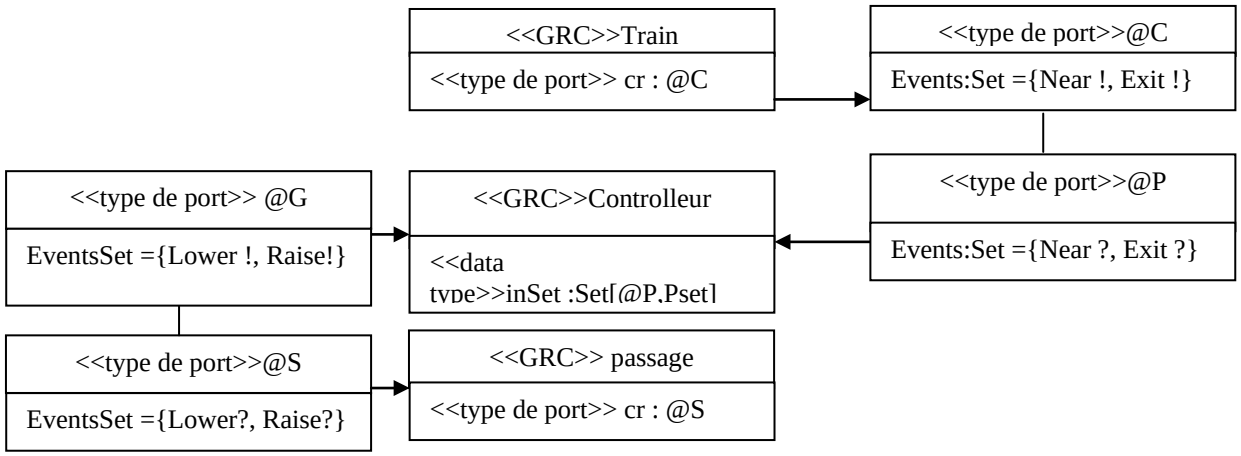


Figure 1 : diagramme des classes des entités contrôleur train barrière

5.3.3 - Concepts fondamentaux d'horloges et régions d'horloges.

Pour générer les régions d'horloges Algar et Zheng ont utilisé les propriétés introduites par Alur et Dill**[31]**. Le modèle de temps est défini sur un ensemble réel non négatif. Une évaluation d'horloge sur un ensemble d'horloges C est une carte v qui assigne à chaque horloge $c \in C$ une valeur dans R^+. L'ensemble des évaluations d'horloge est noté par V(C).

- La contrainte de temps est un prédicat qui entraîne une évaluation d'horloges.
- Si m est une constante de temps m $\in$ Q^+ alors V(C)(x)<= m et V(C)(x) >=m sont des contraintes de temps.
- Une région d'horloge (α) est une classe d'équivalence de valuation d'horloges provoquée par la relation d'équivalence $\cong$ **[26].**
- Une région d'horloge α' est dite successeur temporel d'une région d'horloge α si pour chaque v $\in\alpha$ il existe un nombre positif t $\in$R tel que v+t $\in\alpha$'.

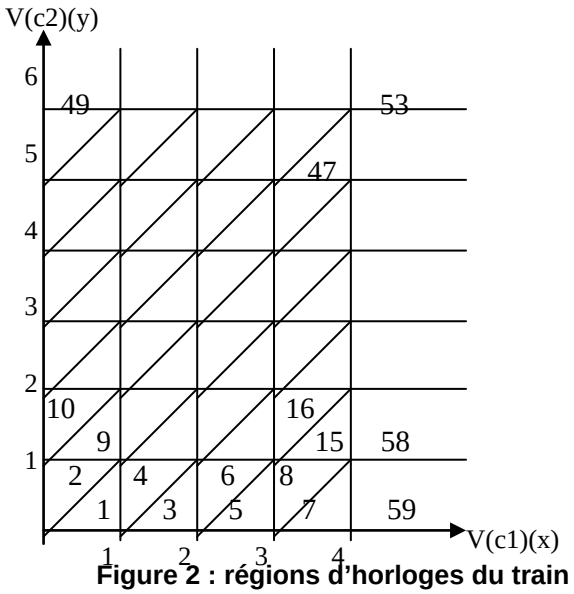


Figure 2 : régions d'horloges du train

5.3.4 - Définition d'un automate à grille.

Chaque objet TROM « A » possède une taille de grille appelée granularité d= 1/k tel que k>1 est le nombre d'horloge dans A. Lorsque i k = 1 on choisit d=1/2. L'automate à grille $G_d(A)$ correspondant à la machine d'état TROM A est l'automate (Θ, θ_0, L , T)tel que :

- Les états Θ = { <s, v'> / <s, v> est un état étendu de A et v' = v+ ka, v' < ∞ , k>0}
- L'état initial θ_0, = $<s_0, v_0>$l'etat initial étendu de A
- L'étiquette de transition est définie par L = ε U {d}
- La fonction de transition T est définie par Θ * L → Θ défini par
  T(<si, vi>,e) = $<s_{i+1}, v_i>$ si e$\in\varepsilon$
  T(<si, vi>, d) = <si, vi+d>

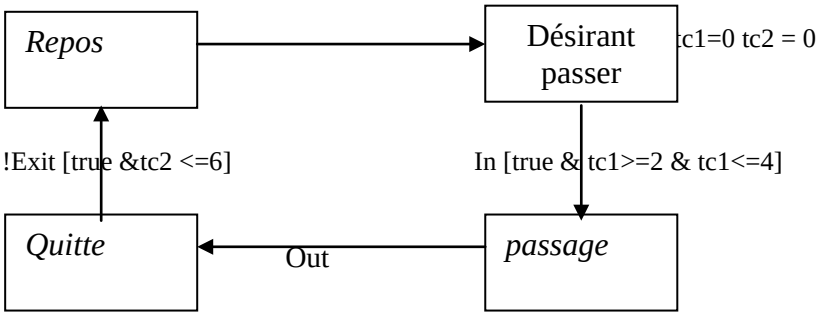


Figure 3 : Diagramme d'état du train utilisé pour la construction de l'automate à grille

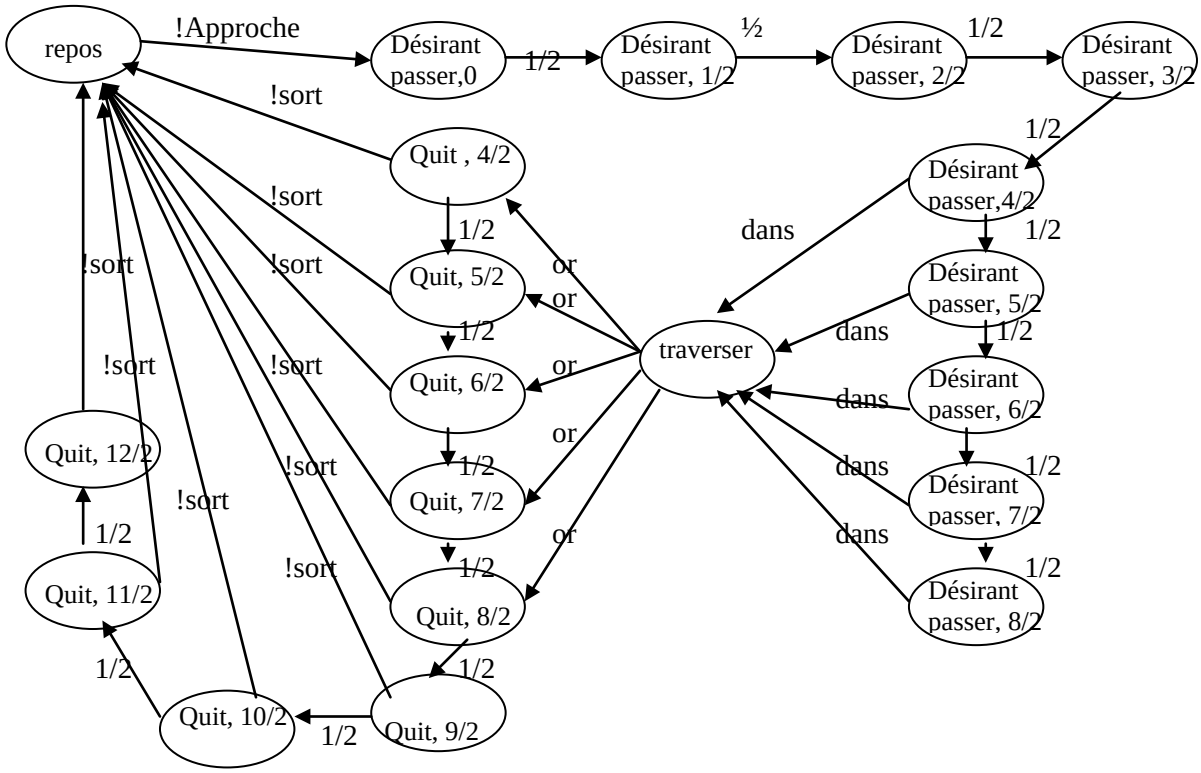


Figure 4 : Automate à grille du train.

5.3.5 - Méthodologie de test des systèmes réactifs à temps réel.

Soit P un programme qui subit un test de conformité pour une spécification A. Le théorème d'homéomorphisme**[26]** montre qu'il est suffisant de tester la conformité du programme P à la spécification $G_d(A)$

A - Algorithme GA : construction de l'automate à grille.

L'algorithme accepte la spécification des classes TROM et délivre la spécification de l'automate à grille.

L'algorithme choisit la granularité d du TROM avec k >1 cependant si la classe TROM possède seulement une horloge la granularité choisit est ½.

Pour chaque contrainte de temps TC_i = {Ri, e, $|t_i, t_j|$, {} } dans la spécification de classe TROM les étapes suivantes sont effectuées :

Etape 0 initialisation :

ES événement de TROM ; SS états de TROM ; TS spécifications de transition de TROM.

Etape 1

Trouver les Spécifications de transition du TROM contrainte par Tc_i .

Ri : <si,sj> ; e (condition du port) ; condition permise⇒ poste condition.

Etape 2.

Calcul des points de grille : <si, v'+0>, <s_i, v'+d>, <s_i, v'+2d>,….., <s_i, v'+md>, ou m est le plus grand entier positive pour lequel md = t_j ;

Etape 3.
Ajouter l'événement interne d à ES, enlever si de SS et ajouter les points de grille calculés dans l'étape 2 à SS

Etape 4.
Effectuez les changements suivants sur TS :

- Pour n tel que 0<=n<= t_j augmenté par t chaque temps, ajouter les transitions (incrément de temps) à TS : <<si, v'+n>, <si, v'+n+d> ; d(true) ; true ⇒ true ;
- Enlever les transitions de TS : Ri <si, sj> ; e (condition de port> ; condition activée ⇒ post condition
- Pour n ti <= n <= tj augmenter par d chaque temps ajouter la transition (contrainte de temps) à TS : << si, v'+n>, <sj, v'+n> ; e (condition de port) ; condition activée ⇒ poste condition ;

B - L'algorithme TC : génération de cas de tests à partir des automates à grille

L'algorithme génère un ensemble de séquences de tests basées sur une couverture d'états. Une séquence de test basée sur une couverture de transition est générée puis ajoutée à la couverture d'états. L'ensemble des cas de tests résultant possède les propriétés de minimalisation et d'exhaustivité.
Les cas de tests générés à partir de l'automate à grille sont présentés à la figure 5.

C- Test de système.

Pour réduire la complexité de teste du système ce dernier est partitioné de manière à ce que les cas de test du système puissent être générés à partir des cas de test des composants individuels dans la partition.

La couverture minimale C pour un graphe G = <V, E> résumée à partir de l'architecture du système réactif (figure 6) fournit une partition du système telle que tous les états du système communiquent avec au moins un état de la couverture. Cette couverture est donnée par l'algorithme de Greedy**[34].**

!Approche	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans
!Approche .1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or
!Approche .1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2
!Approche .1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2
!Approche .1/2.1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2.1/2
!Approche .1/2.1/2.1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2.1/2.1/2
!Approche .1/2.1/2.1/2.1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2.1/2.1/2.1/2
!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2.1/2.1/2.1/2.1/2
!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2.1/2.1/2.1/2.1/2.1/2
!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2	!Approche .1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2.dans.or.1/2.1/2.1/2.1/2.1/2.1/2.1/2.1/2

Figure 5 : Cas de test générés pour l'objet Train

Dans une partition, il existe une interaction entre chaque couple d'objets. Il est alors suffisant de tester les paires d'objets et de rassembler les séquences de tests de toutes les paires afin d'avoir la séquence du composant. Dans le formalisme TROM, il n'existe pas de communication entre les objets d'une même classe : l'interaction se fait entre objets de classes différentes. Il est possible de composer le produit machine synchronisé de deux objets en utilisant l'algorithme GA afin de

déduire l'automate à grille de la machine de produit synchronisé. Cet algorithme est donné dans **[26]**

exemple

Un système composé de cinq trains, deux contrôleurs et deux passages est modélisé par le graphe suivant

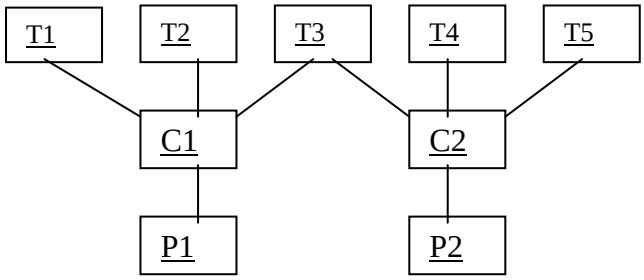


Figure 6 l'architecture du système Train contrôleur barrière

La couverture de notre système est C={C1, C2} ce qui nous permet de partionner le système en deux

$\prod 1$ = {T1, T2, T3, C1, G1}, $\prod 2$ = { T3, T4, T5 , C2, G2}

Comme dit précédemment les cas de test du système sont l'union des cas de tests des deux partitions

la partition $\prod 1$ peut être partagée en sous partitions {T1,C1,P1}, { T2,C1,P1}, {T3,C1,P1}l'interaction de chacun de ces ensemble se fait de manière suivante :

Ti→ C1 → P1. Il suffit donc de considérer l'interaction des paires (T1, C1) et (C1, P1) et de générer les cas de test pour chaque paire. Enfin on génère le cas de test de la composition des deux paires. Les cas de test du système vont correspondre à l'union de tous les cas de test.

5.4 - Notre contribution Complément à la méthode d'Algar et Zheng.

Le formalisme TROM comme définit dans **[25]** ne prend pas en charge l'interaction entre les objets d'une même classe. Cette restriction pose un problème au testeur au cas ou l'interaction entre ces objets est pertinente. Dans ce qui suit, on propose une extension au model TROM afin de pouvoir prendre en charge la communication entre les objets d'une même classe. Afin de mieux mettre en valeur ce problème on propose une extension au problème de train vu précédemment.

5.4.1 - Extension proposée au problème du train.

Les trains doivent ralentir avant de passer par le passage à niveau pour cela chaque train doit prévenir les trains qui utilisent la même voie pour qu'ils réduisent leur vitesse à leur tour. A sa sortie du passage à niveau le train doit revenir à sa vitesse initiale. Il devra en informer les autres trains pour qu'ils en fassent de même.

Le train envoie la demande de ralentissement 2 unités de temps avant l'envoi du message d'approche au contrôleur.

Le train ralentira 1 unité de temps après l'envoi du message de demande de ralentissement.

Le train ralentira 1 unité de temps après l'arrivée d'une demande de ralentissement.

Le train envoie la demande d'accélération 1 unités de temps après l'envoi du message de sortie.
Le train accélérera 1 unité de temps après l'envoi du message de sortie.
Le train accélérera 1 unité de temps après l'arrivée de la demande d'accélération.

5.4.2 - Nouvelle définition du formalisme TROM.

Tous les objets d'une même classe ont les mêmes propriétés. La différence réside dans les valeurs que prendront ces propriétés. Tous les objets d'une même classe ont le même type de port

L'ensemble des objets constitue la classe réactive. Comme on étudie les objets les uns indépendamment des autres on considérera que tous les objets ont le même comportement mais son exécution diffère dans le temps.

Deux instances d'un même objet communiquent par échanges de messages à travers leurs ports. Le résultat de cette communication est une transition synchronisée. Certains objets peuvent communiquer entre eux tandis que le reste des objets peut effectuer des opérations internes

Les événements sont partitionnés en entrées ? Sorties ! et événements internes. L'événement interne est initié et contrôler par l'objet lui-même mais peut avoir un effet sur les objets de la même classe.

Les événements d'entrées d'un objet proviennent soit de l'environnement extérieur soit des objets des autres classes soit des autres objets de la même classes avec lesquelles il interagit par conséquent un événement d'entrée n'est pas contraint par le temps. Un événement quel que soit son type peut se produire durant un état il causera le changement de cet état et déclenchera un événement futur contraint par le temps.

Toute instance d'objet réactif générique possède un port de communication avec les autres instances de l'objet (en plus de ses ports traditionnels) des attributs et une fonction qui assigne les événements aux ports.

La classe réactive générique a un type de port, des attributs et une fonction qui assigne les événements au type de port.

La classe réactive s'associe à une machine d'états finis qui définit le comportement dynamique de tous les objets de la classe. La machine d'états est associée à des ports dont le type est défini dans la CRG, des attributs dont le type a été inclus dans le GRC, des assertions logiques sur les attributs et des contraintes temporelles

Un événement externe d'entrée ou de sortie ne se produit que dans une instance de port spécifique. Un événement interne ne peut subvenir que dans un port nul. L'événement étiquette les transitions entre les états. Les assertions logiques

dans les attributs spécifient la condition du port, la condition activée, et la poste condition de chaque transition. Les contraintes temporelles sont associées aux transitions pour décrire la réponse aux stimulants. Le model abstrait des systèmes réactifs inclus les instances du model réactif générique, avec les instances de chaque type de port du model correspondant. Les objets d'un sous systèmes communiquent en utilisant un mécanisme de passage de message synchrone.

5.4.3 - Les objets réactives

Un objet générique réactif est un 8 tuple (P, E, S, X, L, ϕ, $\wedge$, Y) tel que :
P est un ensemble fini de types de ports.
E : est un ensemble fini d'événements qui inclus l'événement muet Tick
S un ensemble fini d'états
X est un ensemble fini d'attributs typés.
L est un ensemble fini de traits introduisant les types de données abstraits utilisés dans X.
ϕ : est un vecteur de fonction (ϕ_s, ϕ_{at}) tel que ϕ_s : S → 2^S associe à chaque etat S l'ensemble des etats qui peut etre vide est appelé substates et ϕ_{at} : S → 2^x associe à chaque état S un ensemble d'attributs qui peut être vide et appelé ensemble d'attributs actifs.
$\wedge$: est un ensemble fini de spécifications de transitions
Y un ensemble fini de contraintes temporelles.
Le langage de description des classes réactives générique dérive directement de la description formelle.

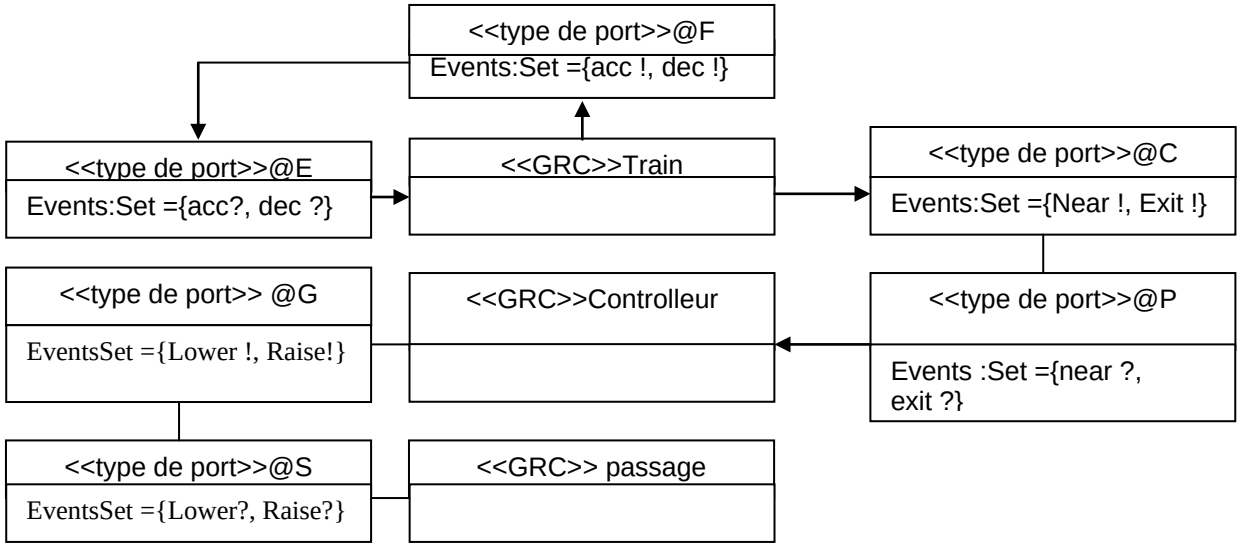


Figure 7 : diagramme des classes des entités contrôleur train barrière.

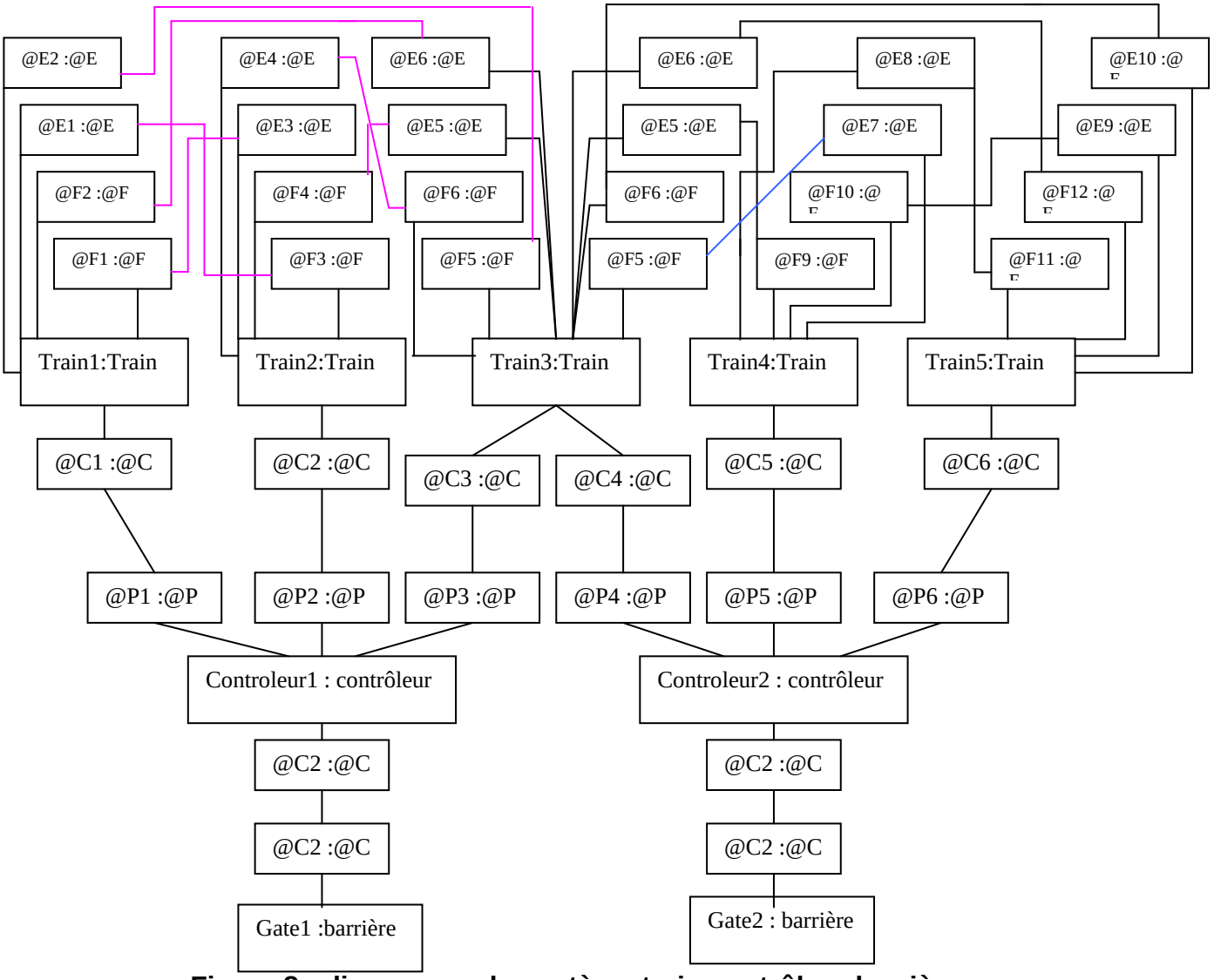


Figure 8 : diagramme du système train contrôleur barrière.

La figure 4 nous présente le diagramme de classe du système trains-controleurs-barrieres. L 'ensemble des trains 1,2 et 3 communiquent entre eux car ils traversent la même barrière. L'ensemble des trains 3,4 et 5 communiquent entre eux aussi. On remarque que les trains 1,2,3 et 4 possèdent chacun 04 ports de communication (02 ports d'entrée et 02 ports de sortie) qui leur permettent de communiquer avec deux trains chacun. Le train 3 possède 08 ports car il doit communiquer avec les quatre trains.

La figure qui suit présente le diagramme d'états de l'objet train.

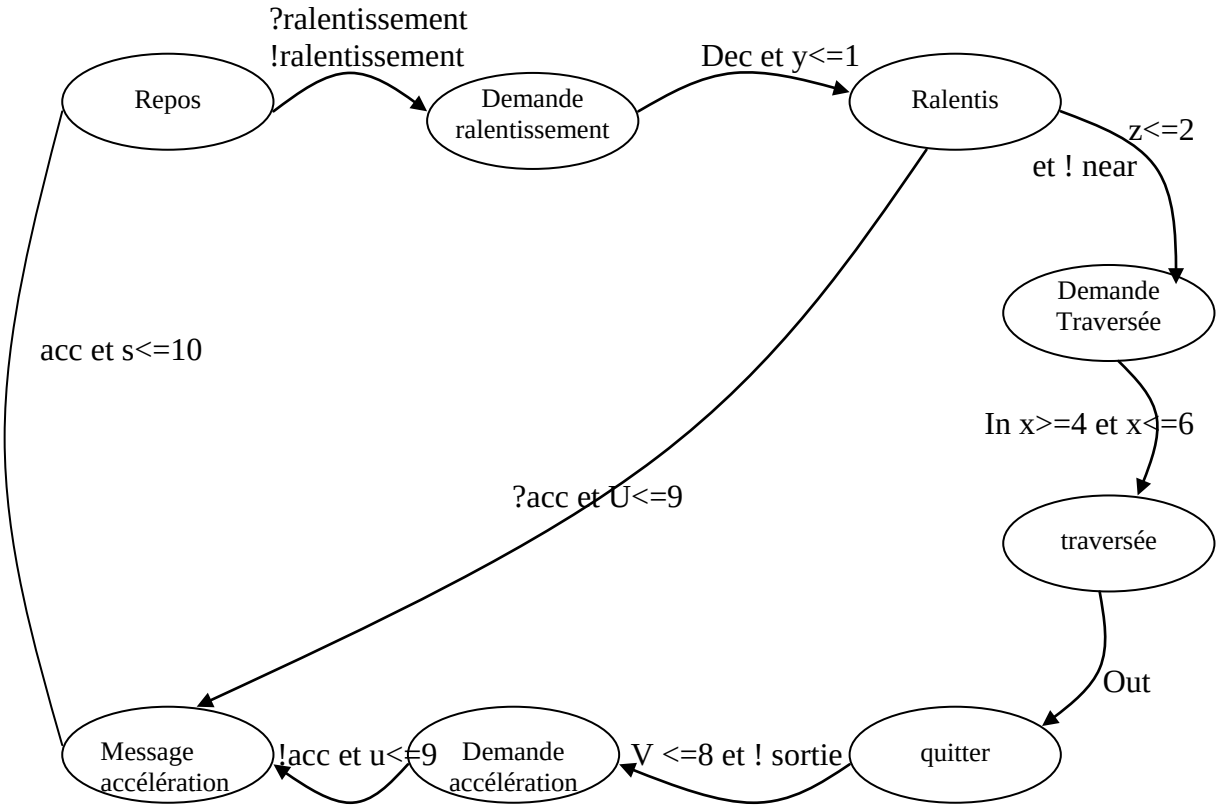


Figure 09 : digramme d'état d'un train

On remarque que ce graphe modélise deux comportements différents de l'objet train. Ces deux comportements sont cependant dépendant l'un de l'autre.

Pour la génération de l'automate à grille on utilisera le même algorithme que celui utilisé par Algar & Zheng. Comme on a deux comportements différents il est préférable de générer deux automates à grille (chaque automate modélisera un comportement).

5.4.4 - Génération de l'automate à grille.

La génération de l'automate à grille se fera en utilisant l'algorithme précédemment décrit.

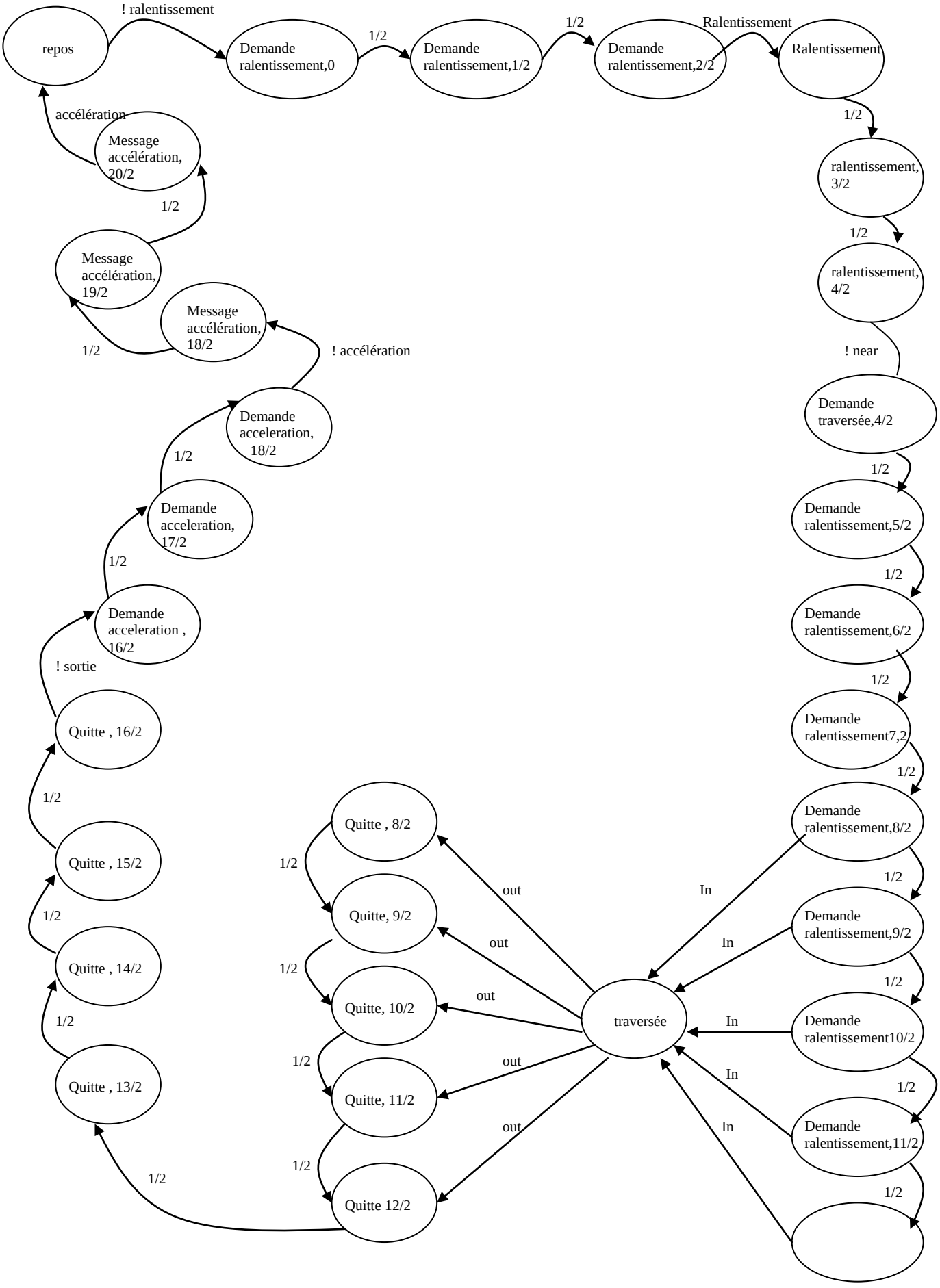

repos
! ralentissement
Demande ralentissement,0
1/2
Demande ralentissement,1/2
1/2
Demande ralentissement,2/2
Ralentissement
Ralentissement
1/2
ralentissement, 3/2
1/2
ralentissement, 4/2
! near
Demande traversée,4/2
1/2
Demande ralentissement,5/2
1/2
Demande ralentissement,6/2
1/2
Demande ralentissement7,2
1/2
Demande ralentissement,8/2
1/2
Demande ralentissement,9/2
1/2
Demande ralentissement10/2
1/2
Demande ralentissement,11/2
1/2
accélération
Message accélération, 20/2
1/2
Message accélération, 19/2
1/2
Message accélération, 18/2
! accélération
Demande acceleration, 18/2
1/2
Demande acceleration, 17/2
1/2
Demande acceleration , 16/2
! sortie
Quitte , 16/2
1/2
Quitte , 15/2
1/2
Quitte , 14/2
1/2
Quitte , 13/2
1/2
Quitte , 8/2
1/2
Quitte, 9/2
1/2
Quitte, 10/2
1/2
Quitte, 11/2
1/2
Quitte 12/2
out
out
out
out
out
traversée
In
In
In
In
In

Figure10 : Automate à grille du train qui envoie la requête.

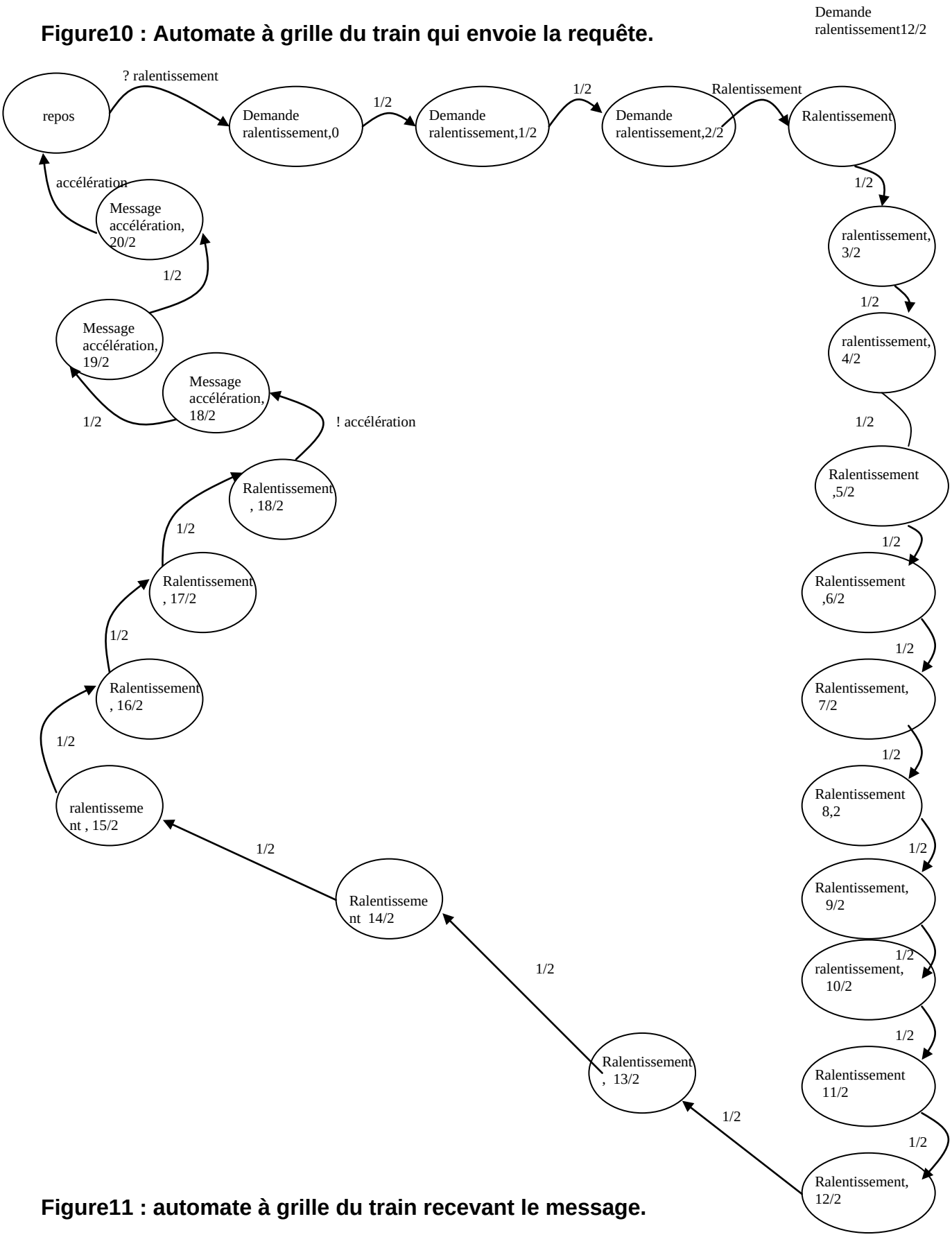


Figure11 : automate à grille du train recevant le message.

La génération des cas de tests se fera en utilisant les algorithmes de couverture d'états et de transition définis dans **[26].** Les cas de tests générés sont les suivants :

!ralentissement
!ralentissement,1/2
!ralentissement,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,
!ralentissement,1/2,1/2,ralentissement,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2, !sortie
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2, !sortie,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2, !sortie,1/2,1/2
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2, !sortie,1/2,1/2, !acceleration
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2, !sortie,1/2,1/2, !acceleration,1/2,
!ralentissement,1/2,1/2,ralentissement,1/2,1/2,!near,1/2,1/2,1/2,1/2,in,out,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2, !sortie,1/2,1/2, !acceleration,1/2,1/2

?ralentissement
?ralentissement,1/2
?ralentissement,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,
?ralentissement,1/2,1/2,ralentissement,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2 ,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2 ?acceleration
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2 ,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2 ?acceleration,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2 ,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2 ?acceleration,1/2,1/2
?ralentissement,1/2,1/2,ralentissement,1/2,1/2,1/2,1/2,1/2,1/2 ,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2,1/2 ?acceleration,1/2,1/2,aceleration

Figure 12 cas de test de l'objet train

La couverture de chaque automate à grille générera un ensemble de séquences de cas de tests qui servira à effectuer les tests unitaires.

5.4.5 - Test du système.

La meilleure approche pour tester un système est de générer les cas de tests à partir de la composition des automates d'états synchronisés. Cette approche possède deux inconvénients :

- L'automate à grille résultant de la composition de quatre ou cinq automates sera énorme, surtout si le système possède un grand nombre d'horloges.

- Il y aura une explosion combinatoire des séquences de cas de tests générées, ce qui rendra impraticable le test du système.

Pour palier à ces deux inconvénients, l'architecture du système sera partitionnée de telle manière que les cas de tests de tout le système seront générés à partir des cas de tests des différentes partitions.

A - L'algorithme de partition.

Le système est composé d'un ensemble d'objets et de canaux de communications reliant les objets qui interagissent entre eux. L'architecture du système peut être modélisée par un graphe où les arêtes modélise les ports de communication et les états les objets. La couverture vertex d'un tel graphe est un sous ensemble d'états de ce graphe qui vérifie la condition suivante :
Chaque état du graphe est adjacent à au moins un état de la couverture vertex. La couverture vertex minimale est une couverture qui ne contient aucune autre couverture. Si on arrive à trouver la couverture vertex minimale C d'un graphe G = <V, E> construit à partir d'une architecture d'un système réactif alors C satisfait les propriétés suivantes :

$\forall\ c_i, c_j \in C < c_i, c_j > \notin E$

$\forall\ c \in C\ \ \Pi_c = \{c'/c' \in V \wedge <c, c'> \in E\} \cup \{c\}$

Puisque chaque arête modélise un port de communication alors la collection $\Pi=\{\Pi_c/\ c \in C\}$ offre une partition de l'architecture du système qui satisfait la condition suivante: tous les objets $\Pi_c - \{c\}$communiquent avec l'objet c donc le fait de tester chaque partition Π_c séparément et d'assembler les différents cas de tests pour tester la totalité du système est justifié.

Le problème de la couverture vertex est de trouver la couverture minimale dans G. En général c'est un problème NP complet cependant l'algorithme de Greedy peut trouver une couverture optimale. La complexité d'un tel algorithme est $\Theta(n)$. Cet algorithme sera utilisé pour partitioner, cependant certaines modifications y ont été apportées.

Lors de la sélection des états il serait plus intéressant d'éliminer les états qui sont en communication directe avec un état déjà sélectionné et qui appartiennent à la même classe que cet état.

Justification.

D'après la définition d'un objet TROM tous les objets d'une même classe ont le même comportement de ce fait ils sont tous reliés aux même objets des autres classes autrement dit : Il n'existe pas deux objets O1 et O2 appartenant à la classe C1 tel que O1 est relié à un objet O3 de la classe C2 et que O2 ne possède aucun lien qui le relie à cet objet.

Partant de ce principe, on peut dire que pour tout objets Oi Oj de la classe Ck reliés par un canal de communication on ne devra sélectionner que l'un des deux objets lors de la création des partitions.

Algorithme de Greedy modifié.

Entrée : le graphe G= (V,E)
Sortie couverture d'état de C
Tant que ($E \neq \phi$) faire
{ $\forall v \in V$ calculer d(v) le degrés d'états de v
choisir un état w de degrés maximum faire
si existe pas un etat v tq v et w apprtient à la même classe et existe pas une
transition e tq e relie w à v

- $C \leftarrow C \cup \{w\}$
- $V \leftarrow V - \{w\}$
- $E \leftarrow E - \{(w,x) / x \in V) \wedge ((w,x) \in E)\}$

}

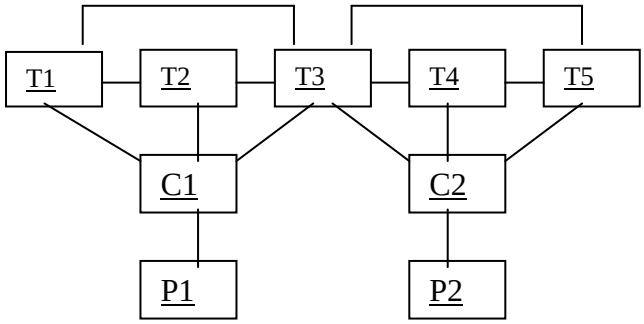


Figure 6 l'architecture du système Train contrôleur barrière

Si on prenait comme exemple notre système étudié train contrôleur passage. On obtient les deux résultats qui suivent :
Avec l'algorithme original :
La couverture vertex est { T3, T1, T5, C1, C2} donc les partitions due système sont
(T1, T2, T3, T4, T5, C1, C2)
(T1, T2, T3, C1)
(T3, T4, T5, C2)
(T1, T2, T3, C1, B1)
(T3, T4, T5, C2, B2)

avec l'algorithme modifié on aura :
la couverture vertex est T3, G1, G2 et les partitions sont les suivantes :

Π 1 (T1, T2, T3,T4,T5, C1, C2)
Π 2 (T1, T2, T3, C1, B1)
Π 3 (T3, T4, T5, C2, B2)

B - Les cas de tests pour les composants d'une partition.

Tous les objets d'une partition interagissent entre eux dans le sens ou il existe une interaction entre toutes les paires d'objets de la partition. Il est suffisant de tester les paires d'objets et de mettre ensemble toutes les séquences de tests pour obtenir la séquence de test de la partition.
La justification apparaît dans **[30]**

Comme on avait défini une interaction entre les objets d'une même classe et que tous les objets d'une même classe ont les même caractéristiques (même comportement) donc il y aura forcément un effet action réaction au moment de l'interaction (chaque objet possède deux automates à grille) Donc quand le premier objet stimulera le second ce dernier devra réagir sans pour autant agir de la même manière. Donc on peut composer le produit de deux objets, générer l'automate à grille correspondant à la machine des produits synchronisés et enfin générer les cas de tests en utilisant les algorithmes de couverture d'états et de transitions.
Prenons comme exemple la partition Π2 générée précédemment :

Π 2 (T1,T2,T3,C1,B1) l'interaction des composants de Π2 peut se faire de la façon suivante : T1→ T2, T1→ T3 , T1→C1 → B1. on peut considérer l'interaction entre les paires d'objets :
T1 → T2, T1→T3, T1→C1 et C1→B1. du moment ou T1→T3 et T1→ T2 généreront les mêmes cas de tests il est suffisant de ne traiter que l'un des deux cas.
On générera l'ensemble de cas de tests τ_{t1t2} relative à la paire T1,T2 à partir de l'automate à grille du produit synchronisé de façon similaire seront générés $\tau_{t1c1}\tau_{c1b1}$ qui seront assemblés pour l'obtention de $\tau_{t1t2c1b1}$ qui est l'ensemble des cas de test de la partition Π2.

Les cas de test du système seront obtenus par l'union des cas de tests des 03 partitions.

C - Algorithme de dérivation de la machine de produit synchronisé.

Entrée : machine d'états finis $M_1 = (S_1,E_1,T_1)$ et $M_2 = (S_2,E_2,T_2)$, $E_1 \cap E_2$ l'ensemble des événements partagés
Sortie : machine de produit synchronisé : M = (S ,E,T) ou $S \subseteq$ {si, s'i) / si $\in$ S1 et si' $\in$ S2} E= E1∪E2 $\forall$ (si, si') $\in$ S il existe au moins une transition entrante ou bien sortante

Etape 1 initialisation de NU (ensemble des états explorés) , S ensemble des états de machine produit, T ensemble des transitions

Etape 2
Tant que NU $\neq \varnothing$ faire
- NU = NU – {(si, s'i)}

- $S = S \cup \{(s_i, s'_i)\}$
- Pour chaque événement partagé e qui se produit dans si et si' faire : si ($s_i \xrightarrow{e} s_j$) et ($s'_i \xrightarrow{e} s'_j$) alors $NU = NU \cup (s_j, s'_j)$, $T = T \cup \{ (s_i,s'_i) \xrightarrow{e} (s_j,s'_j)$ la garde et l'action de cette transition sont la conjonction des garde et actions des transitions d'origine.
- pour chaque événement interne e se produisant dans s_i faire : si ($s_i \xrightarrow{e} s_j$) alors $NU = NU \cup (s_j, si')$, $T = T \cup \{ (si,s'i) \xrightarrow{e} (sj,s'i)$ la garde et l'action de cette transition sont la garde et l'action de la transitions d'origine.
- pour chaque événement interne e se produisant dans si' faire si ($s'i \xrightarrow{e} s'j$) alors $NU = NU \cup (s_i, s_j')$, $T = T \cup \{ (si,si') \xrightarrow{e} (si,s'j)$ la garde et l'action de cette transition sont la garde et l'action de la transition d'origine.

fin while
fin algorithmes

Cependant il serait intéressant d'avoir une architecture de test automatisable qui effectuera l'ensemble de ces tâches de façon autonome sans intervention du testeur du moins avec un minimum d'interventions.

5.5 - Architecture de Test : notre approche.

5.5.1 - Position du problème.

Comme décrit précédemment, l'étude des différentes approches de tests a montré que celles-ci n'étaient pas totalement implémentables.

Notre travail de recherche s'est alors orienté dans une première phase à une tentative d'optimisation de ces approches. Notre première idée a été d'inclure la méthode de génération des régions d'horloges telle que définie dans l'approche (Fouchal & Petit Jean) **[1,7]** et l'appliquer conformément à la méthode d'Algar. Ceci pour diminuer la complexité de la spécification, et à la même occasion rendre le processus de test plus rapide et beaucoup moins complexe.

Les conclusions de notre étude ont montré que cette approche ne pouvait être valide car n'utilisant qu'un nombre restreint de régions d'horloges.

Aussi nos recherches se sont focalisés sur la conception d'une architecture de test, adaptée au modèle d'Algar et Zheng, mais associée à un modèle de fautes qui guide le testeur durant la phase de test.

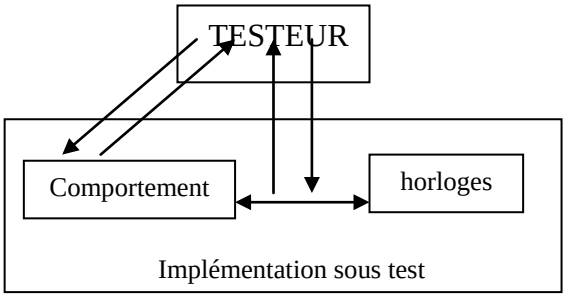
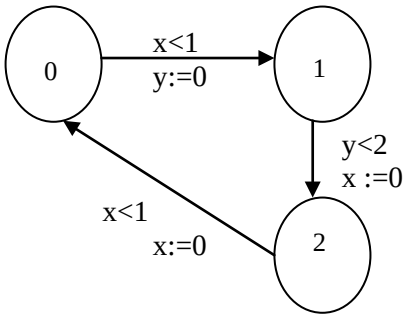


Figure 13 : Architecture de Test

Dans cette architecture l'implémentation sous test (IST) est composée de deux parties : La partie comportement qui exprime la partie contrôle du système et une partie horloge qui contient toutes les horloges du système. Le testeur a pour rôle de contrôler la réponse de l'IST après l'envoi de stimulants ; ceci à partir de deux points de contrôle. Le premier, attaché à la partie comportement, permet au testeur d'injecter les entrées et à partir de laquelle il recevra et contrôlera les sorties. Le deuxième point de contrôle, rattaché à la partie horloges de l'IST, permet au testeur d'interroger les horloges pour prendre connaissance des valeurs ce qui fera que l'IST répondra immédiatement. Considérons la spécification donnée par la figure 8 et l'implémentation de cette même spécification donnée dans la figure 9



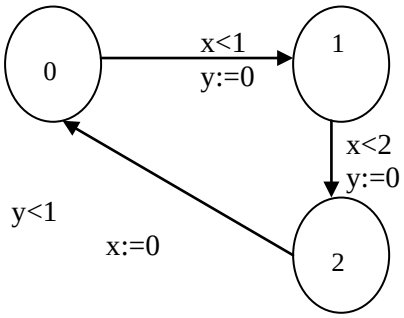


Figure 14 : la spécification **figure 15 : l'implémentation**

A première vue il semblerait que le comportement de l'implémentation se rapproche de celui de la spécification parce que les contraintes expriment les même délais. D'un point de vue fonctionnel l'implémentation est conforme aux spécifications. Cependant si on désire effectuer le test avec l'architecture précédente pour la transition entre les états 1 et 2, on devra vérifier la valeur de l'horloge x. Il est clair que dans la spécification cette dernière est remise à zéro contrairement à l'implémentation. Les zones d'horloges atteintes par la spécification et l'implémentation après l'exécution de cette transition ne sont donc pas identiques ; le résultat du test sera négatif donc l'implémentation n'est pas conforme à la spécification, donc un test avec une architecture pareille peut nous induire à un verdict erroné

La source d'une telle erreur est liée au fait qu'à tout moment on essaie de comparer les valeurs d'horloges contenues dans l'IST à celles contenues dans la spécification. Supposons que cette comparaison soit possible et significative : c'est une restriction de taille sur la structure de l'implémentation. Ceci implique d'une part que l'IST doit contenir des horloges i.e. que son comportement temporel est gérer par des minuteurs et d'autre part que ses horloges sont identiques à celles exprimées dans la spécification i.e. que les différentes horloges dans l'implémentation et la spécification doivent avoir les même noms et la même sémantique.

En d'autres termes une telle architecture n'est utilisables que si on décide que le comportement temporel de l'application doit être exactement identique à celui exprimé par la spécification. De plus un inconvénient de taille majeur est que cette approche ne pourra jamais être utilisée pour un test à boite noire car ne connaissant

pas la structure interne du système on est dans l'incapacité de récupérer les valeurs des horloges internes de ce même système.

5.5.2 Une architecture plus souple adaptée à notre système.

Notre cadre d'étude s'intéresse aux systèmes réactifs à temps réel. Pour pouvoir tester de tels systèmes on doit les partitionner. Pour chaque partition, on teste l'interaction entre les paires d'objet qui interagissent de façon directe. Pour cela on modélise la spécification de notre système sous forme d'un automate à grille décrivant les actions d'entrée et de sortie de chaque objet ainsi que les délais temporels à respecter pour l'accomplissement de chaque événement d'entrée ou de sortie (les messages échangés entre les objets de notre système) et les actions non contraintes par le temps.

L'architecture de test de notre système se base sur le modèle décrit précédemment et auquel on a apporté certaines modifications pour palier aux inconvénients cités précédemment.

L'idée clé est de tester le comportement temporel ainsi que l'interactions entre les objets quelque soit la façon dont ils ont été implémentés. Ce comportement est représenté, dans la spécification (automate à grille), par des contraintes d'horloges, des échanges de messages et des actions à effectuer. Notre but est de vérifier si l'implémentation (les objets) accepte les entrées à tout moment en respectant la spécification, et si les sorties ont lieu au moment attendu. En conséquence et parce que les seules informations temporelles et interactionnelles dont nous disposons sont liées à la spécification, nous devons traiter le comportement dans notre test comme une partie du testeur (figure 10).

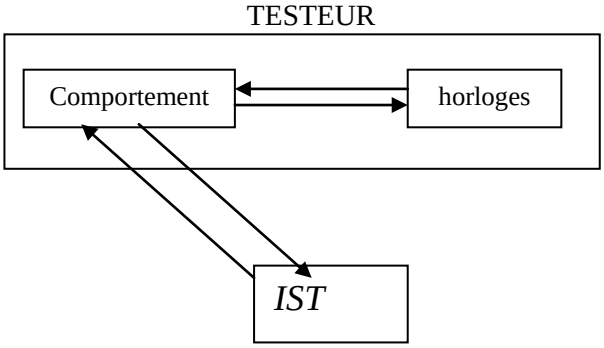


Figure 16 architecture proposée

Cette architecture définit l'espace d'interaction de chaque objet i.e. : avec quels objets interagit chaque objet de notre système. Ceci permet au testeur d'avoir une meilleure connaissance du comportement de chaque objet et donc de simplifier l'opération de test (identification des états de sortie)

Implications.

Comme les horloges font partie du testeur, toute action utilisant les horloges devra être exécutée par le testeur. La partie comportement du testeur (comme la partie comportement de l'IST dans l'architecture de test précédente) pourra

interroger la partie horloge à tout moment afin de connaître la valeur d'une ou de plusieurs horloges. Elle pourra également recevoir la réponse de façon instantanée (La communication est maintenant interne au testeur et non pas entre le testeur et l'implémentation).

D'autre part, les actions exécutées sur les horloges n'impliqueront plus l'implémentation. En effet le testeur, dont on sait qu'il possède les horloges, exécutera les reinitialisations indépendamment de l'IST.

5.5.3 - Etapes d'exécution du test.

1) Avant l'exécution du test, le testeur devra synchroniser les horloges de la spécification (dont il dispose) avec ceux de l'application.

2) Il soumet ensuite, le signal d'entrée à la partie comportementale de l'IST. Ceci, au moment où les évaluations d'horloges correspondront à la zone d'horloge à laquelle appartient le cas de test.

3) Une fois la réponse reçue, le testeur relève les valeurs des horloges (dont il dispose) et vérifie si le symbole reçu est le même que celui attendu ou prévu. De même qu'il vérifie aussi si la valeur d'horloge se trouve dans la zone d'horloge spécifiée dans le cas de test. (I.e. pas les horloges de l'IST mais les horloges du testeur).

Le test porte sur les événements suivants :

- Pour la transition dont l'étiquette contient un symbole de sortie : vérifier si la transition mène vers la région d'horloge désirée et vers l'état désiré de l'objet.
- Si la transition contient un symbole d'entrée : vérifier que le changement d'état provoqué par le symbole d'entrée est conforme à la spécification.
    - Vérifier que le symbole va être accepté pour toutes les évaluations d'horloges de la région en cours.
    - Vérifier que l'état actuel peut réellement être atteint par le système
- Pour un événement non contraint par le temps : vérifier que le changement d'état causé par cet événement est conforme à la spécification.

5.6 - Modèle de fautes

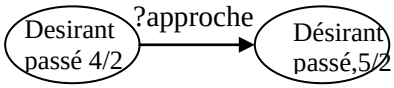
Le modèle de fautes est l'un des aspects importants d'un processus de test. Il permet de montrer les erreurs auxquelles est confrontée la personne qui contrôle la conformité d'une implémentation donnée vis à vis d'une spécification donnée.

Nos présentons les principales erreurs à détecter dans les systèmes réactifs à temps réel. Nous illustrons chaque type d'erreur par un petit exemple basé sur une implémentation erronée. Enfin nous expliquons brièvement la façon de détecter ces erreurs en utilisant notre architecture de tests décrite précédemment.

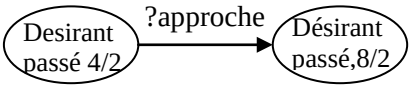
1 - Zone d'horloge inaccessible

L'erreur d'inaccessibilité**[36]** à une région d'horloge se produit si dans la spécification on peut atteindre l'états *désirant passé*,4/2 modélisant la région d'horloge α_{15} à partir de l'état repos modélisant la région d'horloge en utilisant

l'événement ! approche tandis que dans l'IST on atteint l'état (*désirant passé*, 2/2) modélisant la région α_{56} par exemple



la spécification



l'implémentation erronée

Dans ce cas l'erreur qui s'est produite n'a pas d'effet immédiat cependant elle peut induire en erreur pour le reste du processus car le fait d'être dans l'état désirant passé,2/2 signifie que l'on peut franchir le passage dans l'immédiat où dans au maximum 1 et ½ unités de temps ce qui n'est pas le cas avec l'état désirant passé,8/2. La solution pour détecter ce genre d'erreur est d'injecter deux cas de test successifs et d'étudier la réaction de l'IST

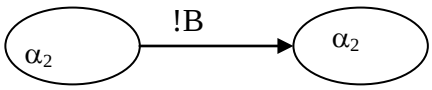
- Pour le premier cas de test on aura comme résultat l'état désirant entrée. Sans avoir la valuation d'horloge correspondante ce qui signifie que pour le testeur ce sera 5/2 mais pour l'implémentation c'est en réalité 8/2. L'erreur n'apparaît pas au testeur mais s'il exécute la 2^{eme} cas de test (le suivant). Il en résultera un blocage de l'IST car on est pas dans la bonne région d'horloge pour injecter. c'est pour cela que le testeur devra fixer un délais de temps limite pour l'attente de réponse de l'IST

2- erreur de sortie temporisé

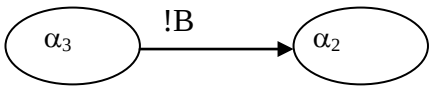
Cette erreur de sortie temporisée **[25]** se produit si :

- il existe une transition dans la spécification étiquetée par ! B. Ce qui signifie que l'action B doit se produire dans la région Z correspondante à l'état de départ de la transition.
- et si la valuation d'horloge obtenue, immédiatement après l'action !B dans l'implémentation n'appartient pas à z ou bien si l'implémentation ne répond pas avant la fin des délais fixés

De manière pratique ce genre d'erreur peut être généré par le fait que la spécification et l'IST n'utilisent pas le même ensemble de compteurs ou alors qu'elles n'ont pas le même ensemble de contraintes sur les actions de sorties. Généralement les délais sont déterminés par la spécification et dépendent de la taille de la zone d'horloge (ils sont différents pour chaque horloge).



Spécification



implémentation

Dans notre spécification, c'est l'envoi du message B qui générera le passage vers l'état prochain. Cet état correspond à la zone α_2. Par contre, dans l'IST, c'est l'état modélisant la région α_3 qui se transformera à l'envoi de B. Cette faute est un cas particulier de ce qu'on a défini précédemment. Du fait qu'après l'erreur générée, le comportement de l'IST rejoint celui de la spécification, on considère que c'est une faute acceptable car elle n'influe en aucun cas sur le comportement futur de l'IST.

3 - erreur de transfert de symbole.

L'erreur de transfert de symboles **[25]** se produit quand la transition ne mène pas au même état (de l'objet) (dans la transition et la spécification). Ce type d'erreurs facilement repérable par l'état de sortie délivré, signifie que l'implémentation est erronée.

4 - Erreur de symbole de sortie.

L'erreur de symbole de sortie **[36]** se produit quand les transitions de la spécification et de l'implémentation ne sont pas étiquetées avec le même symbole de sortie.
Le testeur détecte cette erreur en injectant un cas de test. en sortie, le testeur reçoit une réponse de l'IST différente de celle spécifiée dans la spécification.

5 - Erreur de symbole d'action.

L'erreur de symbole d'action se produit quand l'action non contrainte par le temps de la spécification et de l'implémentation sont différents pour une même transition. Généralement ce genre d'erreur est causé par un événement interne qui est mal géré par l'objet lui-même. Il est alors modélisé par un changement d'état imprévisible de l'objet. Ce dernier ne correspondant pas à la spécification, ou bien sera bloqué lors de l'exécution des cas de tests.

6 - Erreur de symbole d'entrée.

L'erreur de symbole d'entrée se produit quand l'objet reçoit un message erroné d'un autre composant. Ce qui provoquera soit un changement d'état non conforme à la spécification soit un échec de l'objet (signifiant que l'objet va se bloquer).On remarque que les symptômes sont les même que ceux des cas d'erreurs 1 et 3. Il est à remarquer que le principe de test repose sur les paires d'objets. l'erreur est alors identifiée à la sortie du premier composant et à l'entrée du deuxième ; ce qui facilite la détection de l'erreur.

5.7 - Conclusion.

Nous avons présenté dans ce document une approche de test complétée par une architecture de test pour les systèmes logiciels réactifs à temps réel. L'approche de test permet de tester l'interaction entre les objets d'une même classe. L'architecture est basée sur le principe que ce qui nous intéresse réellement est le respect des contraintes temporelles par l'IST et non la concordance entre les valeurs d'horloges de l'IST et de la spécification. Pour cela nous avons en premier lieu présenté les différentes méthodes existantes pour la spécification des systèmes réactifs à temps réel. Parmi ces méthodes nous avons retenu l'approche d'Algar et Zheng qui est l'une des seules méthodes validées qui traite à la fois l'aspect temporel et réactif. Cependant nous avons relevé certaines insuffisances telles que l'absence d'une architecture de test appropriée. Pour cela, nous avons étudié l'une des architectures validées mais qui demeure mal adaptée aux tests à boite noire et qui ne solutionne pas le problème d'évaluation d'horloges. Aussi nous avons proposé une architecture de test associée à un modèle de fautes. Cette architecture résout le problème de correspondance des évaluations d'horloges de l'IST avec ceux de la spécification. Cependant il reste à valider notre approche par un ensemble de simulations et de tests.

CONCLUSION GENERALE

Dans cette thèse nous avons traité des problèmes de fiabilité des logiciels, spécialement ceux orienté pour les systèmes réactifs temps réel. Pour cela nous avons défini en premier lieu la notion de fiabilité tout en présentant les différentes approches et outils utilisés pour assurer cette dernière. Comme la notion de fiabilité n'est pas absolue et que les besoins en matière de fiabilité diffèrent d'un type d'application à un autre, nous avons présenté les différentes approches et méthodes utilisées, d'une part, et nous les avons classées d'après le type de logiciel auquel elles sont destinées, d'autre part.

Nous avons conclu notre étude par la proposition d'une approche de test destinée aux systèmes réactifs temps réel. Nous avons opté pour cette classe de logiciels car elle est la plus utilisée dans les différents domaines. De plus, il n'existe pas suffisamment d'approches de tests qui ciblent à la fois l'aspect temporel et l'aspect réactif. Ce n'est que récemment que la communauté scientifique s'est intéressée à l'aspect temporel des tests. Notre approche qui s'est inspirée de la méthode d'Algar et Zheng vise à compléter cette dernière en testant l'interaction des objets d'une même classe. Nous avons complété cette méthode par une architecture de test qui lui est adaptée. Cette architecture se base sur le principe que ce qui nous intéresse réellement concerne le respect des contraintes temporelles par l'IST et non la concordance entre les valeurs d'horloges de l'IST et de la spécification. Afin de diagnostiquer les différents cas d'erreur qui peuvent être détectés lors du processus de test, nous avons développé un modèle de fautes qui englobe tous les cas d'erreurs qui peuvent survenir lors d'un processus de test en définissant leurs origines.

Cependant et pour compléter ce travail, une validation est en cours via le développement de modules de simulation afin de tester l'efficacité de notre approche.

Pour nos futurs travaux de recherches nous envisageons de nous intéresser au problème suivant:

Du moment qu'on utilise un système réactif, qu'est ce qui nous prouve que le logiciel qu'on utilise ne réagira pas à certains stimulants non connus par l'utilisateur et qui le pousseront à accomplir des fonctions indésirables et inconnues par ce dernier ?.

BIBLIOGRAPHIE

[1] Bousfiha et al
RAPPORT d'ACTIVITES de RECHERCHE 1998
Université de Technologie de Compiègne
Département Génie Informatique
Centre de Recherches de Royallieu

[2] CIQS compétences sécurité et sûreté de fonctionnement
2002
www.ciqs.fr

[3] A.EN-Nourani, H.Fouchal and R.Dssouli
Test derivation for timed systems
Report LERI-97-03-01, LERI-RS (Université de Reims) and DIRO (Université de Montreal Canada)
1997

[4] www.essi.fr/~Hugues/gl/cours6/sld041.html

[5] D.ESSAME
Tolérance aux fautes dans les systèmes critiques : application au pilotage des lignes de métro automatisées
Rapport LAAS No98414
Doctorat, Institut National Polytechnique, Toulouse, 15 Septembre 1998, N°1450, 190p.

[6] Hacéne Fouchal
Document de synthèse des travaux de recherches
Habilitation à diriger des recherches
Université de Reims Champagne-Ardenne
2002

[7] W.M.Gentleman
If softwar quality is perception how do we measur it ?
Softwar engenieering laboratory
National research council canada july 1996

[8] Paul Jacquet, Yves Ledru, Xavier Nicollin, Marie Laure Potet
Exposition "Logiciel Critique"
15 Janvier 1995 – 15 Avril 1995

[9] Thierry Jéron, Pierre Morel, César Viho
TGV un outil de génération automatique des tests de conformités
2001

[10] James Ledoux
Sur la modélisation structurelle markoviennes en fiabilité du logiciel
Rapport de recherches INRIA N° 2714 Novembre 1995

[11] LIGERON S.A. fiche technique avril 2000
www.Vti_bin/shtml.exe/fichesdf.htm

[12] Limnios et al
RAPPORT d'ACTIVITES de RECHERCHE 1998
Université de Technologie de Compiègne
Département Génie Informatique
Centre de Recherches de Royallieu

[13] Sandra Manzi
sûreté et sécurité des logiciels complexes:
école nouvelle d'ingénieur en communication
2000

[14] www.-mediatheque.imag.fr/mediatheque.imag/exposition/logiciels.critiques/section3-7.html

[15] www.atelierb.societe.com/page_B/fr/methb-01.html
2002

[16] M. Ahmed Nacer
Les systèmes éprouvés : fiabilité validation et tests de logiciels.
Support de cours C.E.R.I.S.T 2001

[17] P. G. Neumann
forum des risques
1995

[18] Vérification et tests des logiciels repartis
2001
www.irisa/pampa/theme/verif

[19] William E.Perry
Effective methodes for softaware testing
Second edition 1999

[20] A.Rauzy
Arlia-Workshop 2.0 user's manual. Technical Report
LaBRI/MVTSI/Aralia/TR98-10, LaBRI, 1998

[21] RAPPORT d'ACTIVITES de RECHERCHE 1996-1999
Groupe TSF
Tolérance aux fautes et sûreté de fonctionnement informatique
LAAS

[22] Carlos Valderrama, François Naçabal, Fabiano Hessel, Pierre Paulin, Ahmed Jeraya
Co-simulation C-VHLD pour la validation fonctionnelle de logiciels embarqués.
2000

[23] Hacéne Fouchal
Adapted test cases for timed systems.
Journal of electronics and computer science, 1(4), December 2001.

[24] J.C. Laprie
Sûreté de fonctionnement informatique
Rapport LAAS CNRS 01326 Juillet 2001

[25] Eric Petitjean, Hacéne Fouchal
« A realistic architecture for timed testing »
Université de Reims Champagne-Ardenne
Moulin de la Housse BP 1039 51687 Reims Cedex 2, France
2001

[26] V.S. Alagar, M.Zheng
« A Rigorous method for Testing Real-Time reactive systems »
Department of computer science Concordia University
Montreal, Quebec H3G 1M8, Canada
2000

[27] Brian Nielsen and Arne Skou
« Automated test generation from timed automata »
April 2001

[28] Genie logiciel et tests logiciels
DESS CCI IOANNIS Parissis
2002

[29] Sébastien Salva, Eric Petitjean,
« A simple approach to testing Timed Systems »
Université de Reims Champagne-Ardenne
Moulin de la Housse BP 1039 51687 Reims Cedex 2, France
2000

[30] M.Zheng
Automated generation of test suites from formal specifications of Real Time Reactive Systems
PhD thesis, Department of computer science, Concordia university, Montreal Canada 2003

[31] Rajeev Alur and David L.Dill
« A theory of timed automata. »
25 April 1994

[32] Robert V. Binder., Addison-Wesley,
« Testing Object-Oriented Systems :Models, Patterns, and Tools »
October 1999.
ISBN 0-201-80938-9.

[33] A.En-nouaary H.Fouchal R.Dssouli a.Elqortobi
« Timed Test Cases generation based on state characterization technique »
Decembre 1998

[34] D.Harel, A. Pnueli
« On the development of reactive systems. In logic and models of concurrent systems »
1985

[35] V.S. Alagar, R. Achuthan, D. Muthiayen
« TROMLAB : A software development environment for *Real-Time Reactive Systems »*
2001

[36] Brian Nielsen
PhD thesis « Specification and test of real time systems »
Departement of computer science
AALBORG University Danemark
April 2000