INSTITUT POLYTECHNIQUE DE GRENOBLE

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

THESE

pour obtenir le grade de

DOCTEUR DE GRENOBLE INP

Spécialité : " Informatique "

préparée au Laboratoire de Conception et d'Intégration des Systèmes (LCIS)

dans le cadre de **l'Ecole Doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatiques (MSTII)**"

présentée et soutenue publiquement par

Thi Quynh BUI

14 Octobre 2009

SERVICE DE DIAGNOSTIC EN LIGNE POUR LES APPLICATIONS A BASE DE COMPOSANTS LOGICIELS

Directeurs de thèse :

M. Michel DANG et Mme Oum-El-Kheir AKTOUF

JURY

Mme Claudia RONCANCIO	, Président
Mme Catherine DUBOIS	, Rapporteur
M. Daniel HAGIMONT	, Rapporteur
M. Michel DANG	, Directeur de thèse
Mme Oum-El-Kheir AKTOUF	, Co-directeur de thèse
M. José DA GRACA MARTINS	, Examinateur

Remerciements

Je tiens tout d'abord à remercier profondément M. Michel DANG et Mme Oum-El-Kheir AKTOUF, pour avoir encadré cette thèse. J'exprime également tout mon respect pour leur accueil plus que sympathique depuis mon arrivée en France. Je les remercie pour leur soutien, leur disponibilité, leur confiance à mon égard et leur regard précieux pendant mes années de travail. Leur rigueur intellectuelle, leur très grande compétence, leurs encouragements et leur gentillesse ont été déterminants pour ces travaux.

Je remercie également les personnes qui ont accepté d'évaluer mon travail et m'ont fait l'honneur de participer à mon jury de thèse :

Mme Claudia RONCANCIO, Professeur à Grenoble INP,
Mme Catherine DUBOIS, Professeur à l'ENSIIE,
M. Daniel HAGIMONT, Professeur à l'INP Toulouse,
M. José DA GRACA MARTINS, Président Directeur Général de EF Tecnologias,
M. Michel DANG, Professeur à Grenoble INP,
Mme Oum-El-Kheir AKTOUF, Maître de conférences à Grenoble INP

Mme Catherine DUBOIS et M. Daniel HAGIMONT ont accepté de consacrer une partie de leur temps précieux pour évaluer mon travail en tant que rapporteurs. Je leur témoigne tout mon respect et ma reconnaissance.

Mes remerciements vont également aux directeurs successifs du laboratoire LCIS, Mme Chantal ROBACH et M. Eduardo MENDES, et à l'équipe CTCYS du laboratoire. J'ai eu l'occasion de me lancer dans une ambiance agréable avec l'aide, le soutien et la bonne humeur de tout le laboratoire. Merci pour les apports à la fois humains et scientifiques qui ont fortement participé à la mise en œuvre de ce travail.

Je voudrais aussi remercier tous mes professeurs, depuis l'école primaire jusqu'à l'université, qui m'ont apporté leur savoir – faire et donc contribué de manière indirecte mais importante, à la réalisation de cette thèse.

Je dois aussi beaucoup à mes parents, dont les bénédictions m'ont suivie tout au long de cette thèse. Je leur exprime toute mon admiration, mon affection et ma gratitude.

Je tiens tout spécialement à remercier mon mari Minh Duc NGUYEN et tous les membres de ma famille, pour leur grand amour, leur confiance et compréhension et surtout pour leur support sans condition depuis toujours …

Je salue et remercie sincèrement tous mes ami(e)s qui ont toujours été là quand il le fallait, et qui ne se sont jamais lassés d'apprécier les hauts et de supporter les bas de ma vie.

Enfin, cette thèse est dédiée à ma petite fille qui naîtra dans quelques semaines…

SOMMAIRE

LISTE DES FIGURES

LISTE DES TABLEAUX

Chapitre 1

INTRODUCTION

Ce chapitre donne un aperçu du contexte et des objectifs de ce travail de thèse. Tout d'abord, nous introduisons le contexte général de cette thèse, et les objectifs de nos travaux. Ensuite, nous résumons les principales problématiques traitées et nos contributions. Nous décrivons enfin l'organisation et la structure de ce rapport.

1.1. Contexte

L'informatisation croissante de divers domaines de la vie courante entraîne une exigence forte sur la qualité des services délivrés par les systèmes informatiques. La sûreté de fonctionnement constitue un des critères les plus importants de la qualité globale d'un système informatique. Elle inclut en effet des propriétés telles que la fiabilité, la disponibilité, la sécurité, *etc*. qui confèrent à un système l'aptitude à délivrer un service dans lequel l'utilisateur peut avoir confiance, et à s'assurer que cette confiance est justifiée [LAP96].

Depuis plusieurs années, le développement du logiciel s'oriente vers une nouvelle approche permettant d'en simplifier la conception et la maintenance. Cette approche a pour objectif la construction d'applications en intégrant des briques logicielles bien définies et autonomes, appelées *composants* [HEI01]. La notion classique de développement d'applications par écriture de code a été remplacée par l'assemblage de composants préfabriqués. A l'image des systèmes matériels, les applications ainsi développées ne sont plus des blocs monolithiques mais le regroupement de "pièces" bien définies. La nature composable des applications permet alors, en cas de besoin de maintenance, de repérer et de remplacer plus facilement les pièces concernées. En outre, les applications logicielles actuelles deviennent de plus en plus distribuées et opèrent dans des environnements hautement dynamiques. Dans ce contexte, la sûreté de fonctionnement de ces applications est indispensable pour permettre la livraison de services corrects aux utilisateurs du système. Or, l'utilisation du développement à base de composants pose de nouveaux défis à la sûreté de fonctionnement et ouvre la voie à de nouveaux domaines de recherche.

L'analogie avec les systèmes matériels pousse naturellement à explorer les solutions développées et éprouvées pour les systèmes matériels et à analyser leur adéquation et leur adaptation aux applications logicielles à base de composants. Ainsi, parmi les approches proposées pour la sûreté de fonctionnement des applications logicielles à base de composants, nombreuses sont celles basées sur la réplication des composants et le masquage des fautes.

Les services fournis sont alors garantis corrects bien que certaines fautes puissent corrompre le fonctionnement de quelques composants applicatifs. De telles solutions peuvent néanmoins être coûteuses à cause de la réplication des composants. Une alternative intéressante aux méthodes de réplication serait l'utilisation des techniques de diagnostic, dont le principe est de mettre en place des tests inter-composants afin de détecter et localiser les composants défaillants. De telles techniques ont été utilisées avec succès par le passé pour localiser les éléments défaillants dans des systèmes matériels. A notre connaissance, nos travaux sont parmi les premiers travaux visant à assurer la sûreté de fonctionnement des applications logicielles à base de composants, en utilisant une approche de tests inter-composants et de diagnostic en ligne.

Il est important de noter ici que le diagnostic seul a pour unique objectif de détecter et localiser les composants défaillants. Pour permettre la mise en œuvre de mécanismes complets de tolérance aux fautes, il doit impérativement être complété par des techniques de reconfiguration. Cet aspect n'est pas abordé dans nos travaux.

1.2. Objectifs de la thèse

Pour favoriser la productivité et simplifier le développement, le programmeur ne doit se préoccuper que de la logique métier de son application. Les autres services, souvent qualifiés de non-fonctionnels, et nécessaires pour l'exécution de l'application, doivent être réalisés séparément par des spécialistes qualifiés dans des domaines spécifiques comme les transactions, la sécurité, la persistance, *etc*. Le diagnostic de composants défaillants, qui fait l'objet de cette thèse, doit être traité comme une *propriété non-fonctionnelle*. Il doit considérer des critères de transparence, de séparation et composabilité, de visibilité, de réutilisabilité et de portabilité afin de décharger le développeur le plus possible de la mise en œuvre des mécanismes liés à la sûreté de fonctionnement.

Les différentes approches de tolérance aux fautes dans les applications à base de composants peuvent être classifiées selon leur niveau d'intégration dans le système. Comme nous le verrons au chapitre 2, ce sont principalement des approches de niveau système, des librairies, des approches d'intégration ou des services. L'analyse de ces différentes approches selon les critères ci-dessus nous a permis d'en déterminer les avantages et les inconvénients quand à la mise en place du diagnostic de composants. Il ressort de notre analyse que l'approche service est le mieux adaptée au contexte de nos travaux. En effet, cette approche permet d'ajouter élégamment des mécanismes de tolérance aux fautes à une application à base de composants, de façon transparente, portable et configurable. Cette approche offre également une très bonne séparation entre les mécanismes du diagnostic et l'application.

En outre, les composants logiciels s'exécutent dans des environnements complexes qui peuvent évoluer dynamiquement. Les applications construites à base de composants doivent s'adapter de façon autonome aux changements de disponibilité des composants. Notre objectif est donc de proposer un service de diagnostic *flexible* qui tient compte des évolutions des applications.

Notre travail de thèse vise à développer une solution de diagnostic *générale*, et indépendante le plus possible des applications et de leurs modèles de composants. En procédant à l'abstraction des éléments techniques et des concepts liés à chaque modèle de composant logiciel, il devrait être possible de définir une approche qui favorise la réutilisation et qui peut être exploitée dans différents contextes.

Notre approche est mise en œuvre sous forme d'un service de diagnostic DISCO (**DI**agnosis **S**ervice for **CO**mponent-based applications). Le rôle de ce service est de prendre en charge la détection et la localisation des composants fautifs durant l'exécution de l'application. Dès que le processus de diagnostic est terminé, le système est capable d'isoler les composants défaillants, en ignorant leurs sorties et en lançant une opération de reconfiguration.

1.3. Résumé de nos contributions

Le service de diagnostic proposé fournit la possibilité de détecter et de localiser les composants fautifs durant l'exécution de l'application. Il se base sur des tests inter-composants en ligne.

La première étape de nos travaux a été d'étudier la problématique du test en ligne de composants logiciels en vue de proposer des solutions générales, pouvant s'appliquer à tout modèle de composant logiciel. La principale difficulté a été ici de faire interférer le moins possible le test avec la fonctionnalité du composant. Par la suite, nous avons proposé une architecture pour le service de diagnostic DISCO en vue de son implémentation. Cette architecture est générale, et indépendante le plus possible des applications et de leurs modèles de composants. En outre, elle est flexible et s'adapte à l'évolution dynamique des applications à base de composants.

Les réalisations pratiques de cette thèse ont concerné l'implémentation d'une plate forme d'expérimentation en utilisant le modèle de composant OSGi [OSGi]. Cette réalisation avait pour objectif la validation de nos propositions et l'évaluation expérimentale du service DISCO. Nous avons choisi le modèle de composant OSGi car il fournit un environnement d'exécution dynamique qui permet d'installer et désinstaller des composants à tout moment, et d'assembler des interfaces et des composants de manière dynamique. Cette caractéristique permet une réalisation plus souple et flexible du service DISCO.

Nous avons ensuite porté notre service de diagnostic sur un cas d'étude industriel qui consiste en un système de gestion de données de capteurs hétérogènes. Ce travail nous a permis d'expérimenter concrètement le portage et la réutilisation du service DISCO.

1.4. Organisation du document

Ce document est organisé en 5 chapitres principaux :

- Après le chapitre 1 qui est une introduction générale de nos travaux et des objectifs de cette thèse, le chapitre 2 présente les principaux concepts, en relation avec le contexte scientifique de cette thèse. Il s'articule autour de 3 thèmes principaux. Tout d'abord, la notion de programmation par composants ainsi que les principaux modèles de composants logiciels utilisés de nos jours sont présentés. Ensuite, les concepts de base de la tolérance aux fautes et du diagnostic sont présentés, ainsi que les principales approches de la tolérance aux fautes dans les applications à base de composants logiciels. Enfin, les travaux existants sur le test de composants logiciels sont décrits.
- Le chapitre 3 présente le test inter-composant en ligne. Il explique comment tester un composant pour servir le processus de diagnostic. Il montre des exemples assez simples de mise en œuvre des tests en ligne.

- Le chapitre 4 décrit notre service de diagnostic DISCO, son architecture, l'implémentation détaillée de ses composants, ainsi que la procédure de diagnostic.
- Le chapitre 5 illustre la validation expérimentale du service DISCO avec la technologie de composant OSGi et montre les résultats d'expérimentation obtenus. L'objectif est de mesurer les surcoûts du diagnostic et d'analyser ses performances pour montrer son apport éventuel dans la tolérance aux fautes des applications à base de composants. Il présente ensuite le portage du service DISCO sur un système de gestion de données de capteurs hétérogènes.

Enfin, le chapitre 6 dresse le bilan des travaux menés durant cette thèse et en présente les principales perspectives.

Chapitre 2

CADRE DES TRAVAUX ET ETAT DE L'ART

Nous présentons dans ce chapitre les principaux concepts, en relation avec le contexte scientifique dans lequel se déroulent nos travaux. Nous présentons tout d'abord la notion de programmation par composants, et les principaux modèles de composants logiciels. Ensuite, nous introduisons les concepts de base et les principales approches de la tolérance aux fautes dans les applications à base de composants logiciels. Nous les présentons et les comparons en fonction de critères que nous aurons préalablement définis. Enfin, cette évaluation nous permet de situer notre approche par rapport à celles présentées.

Cet état de l'art s'articule autour de 3 éléments principaux : les modèles de composants logiciels, la tolérance aux fautes dans les applications à base de composants et le test de composants logiciels.

2.1. Les modèles de composants logiciels

La complexité croissante des applications informatiques et leur évolution rapide ont motivé depuis une dizaine d'années l'apparition du concept de logiciel "orienté composant", succédant à l'approche "orientée objet" qui avait considérablement amélioré l'analyse, la conception et le développement des systèmes logiciels. Cette nouvelle révolution dans l'ingénierie informatique vise à concevoir et développer des applications par assemblage de composants logiciels pré-fabriqués et pré-testés [HEI01].

Un composant est réutilisable, c'est-à-dire qu'un même composant peut être employé dans la construction de différentes applications, et sa réutilisation ne nécessite pas, *a priori*, de connaître son implémentation. L'approche à composants est fondée sur l'idée que le développement et l'assemblage de composants peuvent être réalisés de façon totalement séparée par des acteurs différents et dans des emplacements différents.

Le bénéfice de l'approche à composants du point de vue économique est la réduction du temps et du coût de développement et la spécialisation des acteurs intervenant dans le cycle de vie du développement [BAS00]. En plus, elle offre des solutions pratiques à certaines limitations d'approches précédentes, telle que l'approche orientée objet par exemple. Par rapport à l'approche orientée objet, l'approche à composants promeut notamment l'emploi de la composition comme mécanisme de réutilisation, le support pour la livraison et le

déploiement indépendant des composants ainsi que la prise en compte de l'existence d'aspects non-fonctionnels.

Avant de passer en revue les principaux modèles de composants logiciels existants, nous présentons ci-dessous quelques définitions de base.

2.1.1. Définitions

Il n'existe pas une définition consensuelle de la notion de composant logiciel. Cependant, une des définitions les plus souvent citées dans la littérature est celle que donne Clemens Szyperski [SZY02] :

"*Un composant logiciel est une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites au contexte. Un composant logiciel peut être déployé indépendamment et est sujet à une tierce composition*".

Plus précisément, un composant est considéré comme étant une entité logicielle possédant des interfaces bien définies et ayant des dépendances exprimées clairement. Il peut être composé et déployé indépendamment dans un système. Deux types d'interfaces sont généralement présents dans la plupart des modèles de composants. Ce sont les *interfaces fournies* et les *interfaces requises* qui permettent à un composant de communiquer avec son environnement. Chaque interface fournie est un ensemble d'opérations que le composant fournit à d'autres composants, tandis que chaque interface requise est un ensemble d'opérations nécessaires au composant pour effectuer ses opérations. Les noms de ces interfaces peuvent être différents selon le modèle de composant. Les composants peuvent également avoir d'autres types d'interfaces comme nous le verrons dans les modèles présentés ci-après.

Un modèle de composant définit les caractéristiques des composants, de leur composition et est accompagné par un support d'exécution. Dans la suite, nous étudions quelques modèles de composants logiciels. Notre objectif dans ce travail est d'utiliser le concept de test inter-composant en ligne pour déterminer l'état (correct ou défaillant) des différents composants de l'application. De ce point de vue, les aspects qui interviennent dans la définition des interactions entre composants et de l'exécution de ceux-ci nous intéressent particulièrement. C'est pourquoi, nous focalisons l'étude des modèles de composants existants sur les points suivants :

- Définition du composant et de ses interfaces : nous présentons le développement du composant et ses frontières avec le monde extérieur.
- Composition des composants : il s'agit de l'interaction entre les composants ; nous montrons comment les composants sont assemblés pour former des applications.
- Framework du composant : c'est l'environnement d'exécution du composant qui supporte l'exécution des composants en gérant les interactions entre composants et l'invocation des différents services fournis par les composants. L'environnement d'exécution est parfois comparé à un mini-système d'exploitation [BAC00] car il est chargé de gérer des aspects divers tels que le cycle de vie des instances de composants ou bien leurs propriétés non-fonctionnelles.

Ainsi, connaître la définition d'un composant et de ces interfaces nous permet de savoir comment mettre en place le test d'un composant. L'analyse de l'interaction entre les

composants (ou composition des composants) est nécessaire pour développer la notion de test inter-composant en ligne. Enfin, l'environnement d'exécution du composant est important pour la mise en œuvre des tests inter-composants et du diagnostic en ligne des éléments défaillants.

2.1.2. Etude de quelques modèles de composants logiciels

Dans cette partie du rapport, nous regardons quelques modèles de composants logiciels en étudiant la définition du composant et de ses interfaces, la composition des composants et le framework de composant pour chaque modèle. De nombreux modèles de composants logiciels ont été proposés. Nous nous limitons dans cet état de l'art à l'étude de quelques modèles libres parmi les plus connus et les plus utilisés, de manière à présenter les principales utilisations possibles des composants logiciels. Les modèles étudiés sont : .NET, EJB, CCM, Fractal, OSGi.

2.1.2.1. Le modèle de composant .NET

Le modèle de composant .NET [NET] a été développé pour le domaine des applications réparties, qui peuvent être développées avec différents langages de programmation (C# [HEJ03], VB.NET [BAR02], ASP[LIB03], ADO.NET [SCE02], *etc.*). Ce modèle a été introduit par Microsoft et se lie donc au framework .NET, qui est un environnement de développement et d'exécution reprenant, entre autres, les concepts de la machine virtuelle de Java, via le CLR (*Common Language Runtime*).

2.1.2.1.1. Définition du composant et de ses interfaces

Les unités de base des applications .NET sont des composants qui sont appelés *assemblages (ou assembly)*. Un assemblage contient un ensemble de fichiers nécessaires à son exécution : des fichiers d'implémentation, de ressources et de descriptions. Les fichiers de ressources peuvent être des pages HTML, des images, des vidéos ou des sons. Le fichier de descriptions, appelé *Manifest*, fournit les informations telles que le numéro de version de l'*assemblage*, les méthodes importées et exportées ainsi que les dépendances de modules[1]. Le déploiement d'un composant consiste à tout simplement copier les fichiers qui le constituent vers l'emplacement de destination.

2.1.2.1.2. Composition des composants

Les applications sont déployées par la copie des assemblages qui les constituent dans un répertoire spécifique, le GAC (Global Assemblies Cache), ou dans l'arborescence utilisateur. Les fichiers exécutables sont, en général, répartis sur un ou plusieurs fichiers physiques. Dans ces fichiers, les éléments nécessaires pour le déploiement simultané des composants sont regroupés. Le modèle .NET permet de gérer dans le GAC plusieurs versions du même fichier et de rediriger à la demande les appels destinés à un assemblage, vers une version plus récente si cela est souhaité par le concepteur de l'application. En effet, les méta-données rendent tous les composants auto-descriptifs, ce qui permet de connaître leur version. Cela a pour conséquence que toute application installée est automatiquement associée aux fichiers faisant partie de son assemblage.

[1] Ces informations sont appelées des méta-données.

2.1.2.1.3. Framework du composant

Le framework .NET (voir Figure 1) comprend des services et l'environnement d'exécution. Les services se présentent sous forme d'un ensemble de classes appelé *Framework Class Library* (FCL). Ils fournissent des fonctionnalités pour les principaux besoins des développeurs. L'environnement d'exécution comprend :

- Un moteur d'exécution CLR qui est l'équivalent de la machine virtuelle Java. Il permet de compiler le code source de l'application en un langage intermédiaire MSIL (*Microsoft Intermediate Language*). Lors de la première exécution de l'application, le code MSIL est à son tour compilé à la volée en code spécifique au système grâce à un compilateur JIT (*Just In Time*).
- Un environnement d'exécution d'applications et de services web, appelé ASP.NET.
- Un environnement d'exécution d'applications lourdes, appelé *WinForms*.

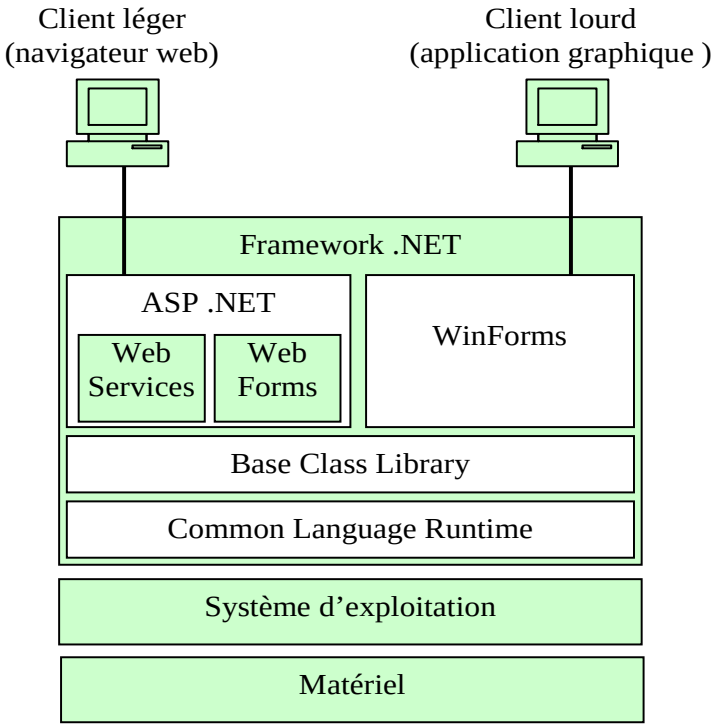


Figure 1. Framework .NET [NET]

Ce modèle a pour principal domaine d'application des applications Web ou des services Web. Il se lie étroitement au framework .NET et supporte plusieurs langages de programmation. Il fournit plusieurs outils pour développer des applications afin de réduire l'effort de développeur.

2.1.2.2. Le modèle de composant EJB (Entreprise JavaBeans)

Sun Microsystem a introduit la première spécification du modèle de composants *Entreprise JavaBeans* (EJB) [EJB] en 1998. Ce modèle a pour objectif de construire des applications de type client-serveur en utilisant une architecture trois tiers [ROM02] : un tiers de présentation, un tiers applicatif, et un tiers de données (Figure 2). Le tiers de présentation correspond à l'affichage, la restitution sur le poste de travail et le dialogue avec l'utilisateur. Le tiers applicatif réside dans le serveur et implémente la logique applicative. Le tiers de

données concerne les bases de données contenant les informations persistantes de l'application. Le modèle EJB est spécifiquement dédié à la construction du tiers applicatif. Les EJB sont implémentés comme un noyau de la plate-forme J2EE (*Java 2 Enterprise Edition*) [J2EE].

La spécification EJB inclut un ensemble de conventions et un ensemble de classes et d'interfaces constituant l'interface de programmation des composants.

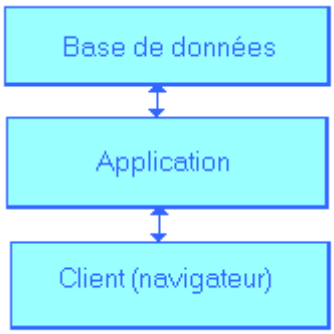


Figure 2. Architecture trois tiers

2.1.2.2.1. Définition du composant et de ses interfaces

Un composant EJB comprend une unique interface fonctionnelle fournie appelée interface *remote.* Le modèle EJB ne permet pas de décrire de façon explicite les interfaces requises du composant. L'implémentation du composant est réalisée par une classe Java qui doit implémenter en plus des méthodes de l'interface *remote*, un ensemble de méthodes permettant de gérer le cycle de vie des instances du composant EJB (création, destruction, recherche, *etc.*).

Les composants EJB peuvent être de trois types : *session*, *entité et message*.

- Les EJB session proposent des services à leur appelant. Ils peuvent être avec ou sans état.
  - Sans état : les composants EJB session sans état peuvent être utilisés pour traiter les requêtes de plusieurs clients.
  - Avec état : les composants EJB session avec état conservent un état entre les appels. Un composant EJB de ce type est associé à un seul client. Par conséquent, il ne peut pas être partagé.
- Les EJB entité représentent des données résidant dans la base de données et qui sont donc persistantes. Chaque enregistrement de la base de données est chargé dans un composant EJB entité pour pouvoir être manipulé.
- Les EJB message sont proches des composants EJB session mais supportent une communication de type asynchrone. Ils sont introduits depuis la version 2.0 du modèle EJB.

2.1.2.2.2. Composition des composants

Le développeur fournit les composants EJB sous la forme de fichiers exécutables Jar. En pratique, les fichiers Jar contiennent l'ensemble des interfaces et des classes d'un ou de plusieurs composants et un descripteur de déploiement.

Développer une application à partir de composants EJB de base consiste à créer de nouveaux composants responsables d'instancier et connecter les composants de base. La composition se fait à travers le langage Java. Les composants dont l'implémentation réalise la composition sont typiquement des composants de type session avec état.

2.1.2.2.3. Framework du composant

Les instances de composants EJB s'exécutent dans des conteneurs comme montrés sur la Figure 3. Les conteneurs sont logés dans un serveur et sont responsables de gérer les composants EJB. La responsabilité la plus importante d'un conteneur d' EJB est de fournir un environnement dans lequel les EJB peuvent fonctionner. Les conteneurs d' EJB logent les EJB et les rendent disponibles à des clients distants. Essentiellement, les conteneurs d' EJB agissent en tant qu'intermédiaires invisibles entre le client et les EJB. Ils sont responsables de relier des clients aux EJB, d'exécuter la coordination des transactions, de fournir la persistance, de contrôler le cycle de vie, *etc.*

Les entités externes au serveur qui appellent un composant EJB ne communiquent pas directement avec celui-ci. En effet, l'accès à un composant EJB par un client se fait obligatoirement via le conteneur. Ainsi, l'avantage de passer par le conteneur est que celui-ci peut utiliser les services qu'il propose et libérer ainsi le développeur de cette charge de travail. Ceci permet au développeur de se concentrer sur les traitements métiers proposés par le composant.

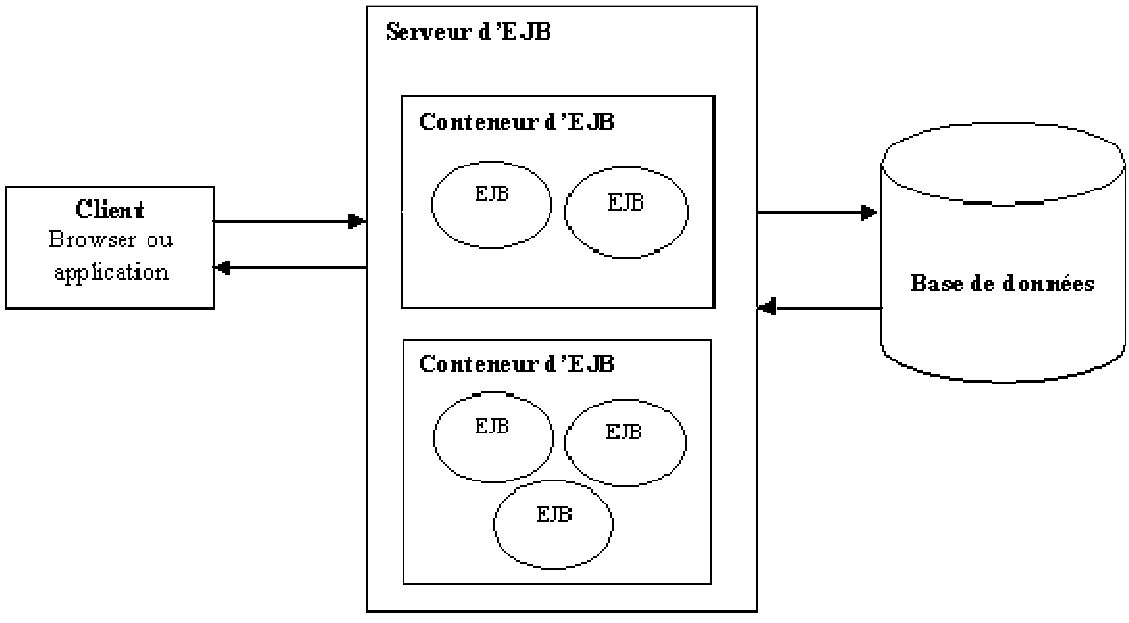


Figure 3. Exécution des composants EJB [EJB]

En bref, ce modèle de composant est utilisé pour le domaine des applications clients-serveurs, côté serveur. La définition d'un composant EJB a une seule interface fournie et pas d'interface requise. Un composant EJB est implémenté en Java.

2.1.2.3. Le modèle de composant CCM (CORBA Component Model)

Le modèle à composants de CORBA (CCM) [OBJ99] a été proposé par l'OMG (*Object Management Group*) en 1999. Il ajoute une couche au dessus de l'intergiciel[2] CORBA [VIN97] permettant de définir l'architecture d'une application distribuée sous forme de composition d'instances de composants.

Ce modèle supporte l'implémentation de composants dans différents langages. Ceci est rendu possible au moyen d'un langage abstrait de description d'interfaces (IDL - *Interface Definition Language*) et d'un langage appelé CIDL (*Component Implementation Description Language*) qui permet de décrire les implémentations ; et de l'infrastructure de communication commune proposée par l'intergiciel, appelée ORB (*Object Request Broker*). Le support de multiples langages d'implémentation constitue un avantage de ce modèle de composant. Cependant, il est nécessaire de tout réaliser de façon abstraite, puis de passer par des étapes de compilation afin de traduire les descriptions abstraites vers des langages concrets.

2.1.2.3.1. Définition du composant et de ses interfaces

Le composant CCM comprend un ensemble de *ports* (Figure 4), qui définissent la manière dont il communique avec d'autres composants :

- Facette : c'est une interface que le composant offre à ses clients.
- Réceptacle : c'est une interface que le composant utilise.
- Source d'événements : c'est un point de production d'événements.
- Puits d'événements : c'est un point de consommation d'événements.

De plus, les interfaces ont une cardinalité associée qui peut être *simple* ou *multiple* et qui contraint le nombre de connexions qui peuvent être réalisées vers une instance du composant.

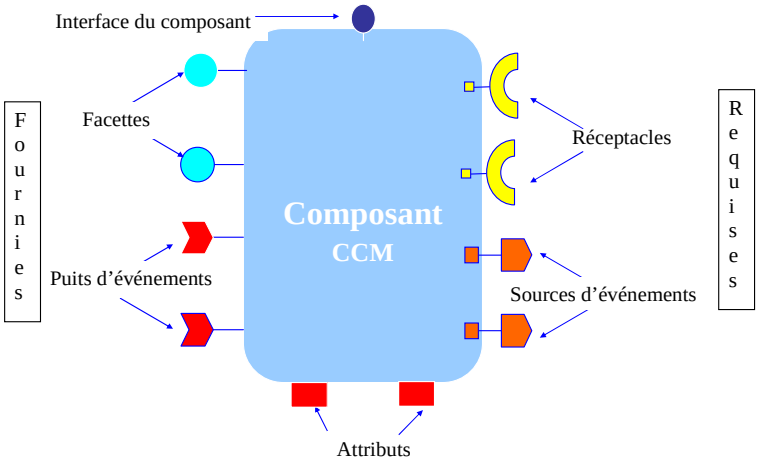


Figure 4. Les ports d'un composant CCM

[2] Dans ce mémoire, les termes "intergiciel" et "middleware" sont utilisés de manière interchangeable.

Les instances de composants sont créées à travers des fabriques appelées *homes* (ou maisons) et une fois créées, leur cycle de vie est géré par un conteneur.

2.1.2.3.2. Composition des composants

Les ports des composants permettent de connecter entre eux les composants qui interagissent. Typiquement, un réceptacle est conçu pour être connecté à une facette et un puits d'événement est conçu pour être connecté à une source d'événements. La description d'une composition, appelée *Assembly*, est réalisée de façon déclarative. Cette description est en XML [XML] (fichier .cad : *Component Assembly Descriptor*).

2.1.2.3.3. Framework du composant

Les caractéristiques non-fonctionnelles sont appliquées à travers des conteneurs. Il s'agit essentiellement de caractéristiques similaires à celles décrites pour le modèle EJB.

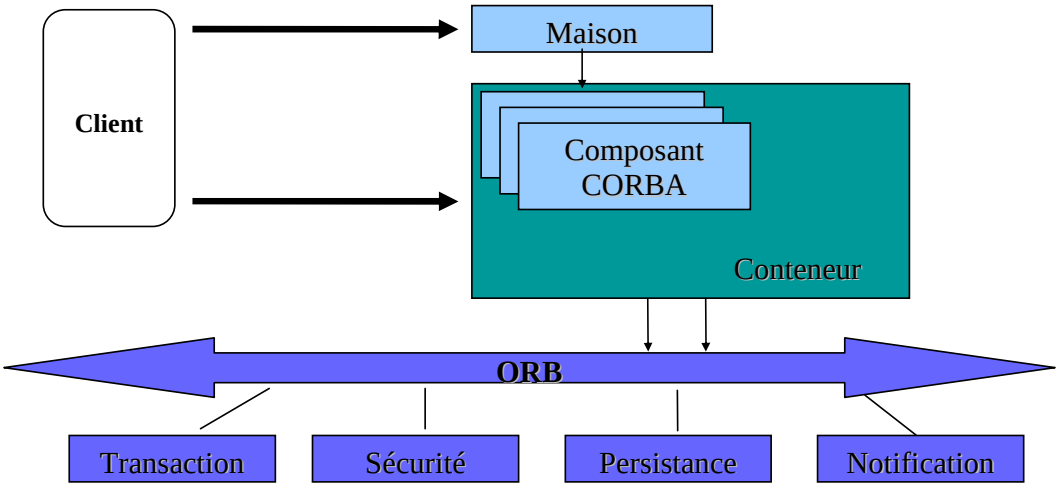


Figure 5. Architecture du conteneur des composants CCM [OBJ99]

Les conteneurs fournissent l'environnement d'exécution pour les composants CORBA. Un conteneur est un cadre pour l'intégration des transactions, de la sécurité, des événements et de la persistance au comportement d'un composant pendant l'exécution. Un conteneur fournit les fonctions suivantes pour ses composants :

- Toutes les instances du composant sont créées et contrôlées pendant l'exécution par son conteneur.
- Les conteneurs fournissent un ensemble standard de services à un composant, en permettant au même composant d'être accueilli par différentes implémentations du conteneur.

En bref, ce modèle de composant CCM supporte plusieurs langages de programmation. Il est adapté pour les applications réparties. La composition des composants est réalisée de manière déclarative durant la phase de développement.

2.1.2.4. Le modèle de composant Fractal

Le modèle de composant Fractal a été introduit par France Telecom R&D et l'INRIA en 2002. Fractal est un framework de composition [BCS02] qui définit un modèle de composants

"générique" car il n'est pas orienté vers un domaine d'application particulier. Il définit la spécification et les implémentations dans différents langages de programmation tels que Java, C, C++, SmallTalk, *etc*. Le framework contient un ensemble d'interfaces de programmation (API) dédiées à la réalisation d'aspects tels que la définition de classes de composants, des fabriques, la reconfiguration dynamique des instances, la composition hiérarchique et l'introspection.

Un composant Fractal comprend un *contrôleur* et un *contenu*. Le contrôleur implémente un nombre variable d'interfaces de contrôle dont certaines sont définies dans la spécification du modèle de composant. Le contrôleur contrôle le contenu d'un composant qui peut être du code fonctionnel ou bien être un ensemble d'autres composants, car ce modèle de composant est récursif. Lorsque le contrôleur agit sur le contenu, il joue un rôle similaire à celui du conteneur des modèles EJB et CCM.

2.1.2.4.1. Définition du composant et de ses interfaces

Un composant Fractal est défini comme étant une entité d'exécution qui possède une ou plusieurs interfaces. Une *interface* est un point d'accès au composant et est classée selon son type, sa cardinalité et sa contingence. Il existe deux types d'interfaces : les interfaces *serveurs* correspondant aux services fournis par le composant, et les interfaces *clients* correspondant aux services requis par le composant. La cardinalité d'une interface spécifie le nombre d'interfaces que le composant peut avoir, elle peut être simple (*singleton*) ou multiple (*collection*). La contingence d'une interface (*obligatoire* ou *optionnelle*) informe de la présence ou de l'absence des opérations fournies ou requises d'une interface à l'exécution.

En général, un composant Fractal est constitué de deux parties (voir la Figure 6) : (1) une membrane qui possède des interfaces fonctionnelles et des interfaces permettant l'introspection et la configuration du composant ; (2) un contenu qui est composé d'un ensemble fini de sous-composants.

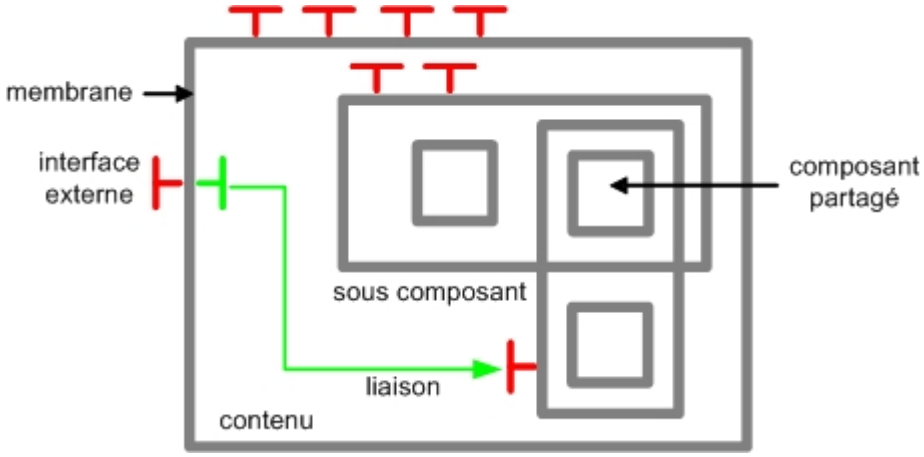


Figure 6. Exemple de composant Fractal [FRA]

Les interfaces d'une membrane peuvent être classifiées en interfaces externes ou interfaces internes. Les premières sont accessibles de l'extérieur du composant, alors que les secondes sont accessibles par les sous-composants du composant. La membrane d'un composant est constituée d'un ensemble de contrôleurs.

2.1.2.4.2. Composition des composants

La composition des instances d'un composant Fractal est réalisée à travers différentes interfaces de contrôle dont *AttributeController*, qui permet de réaliser la configuration d'une instance, *BindingController*, qui permet de réaliser la connexion des instances et *ContentController*, qui permet de créer des composants composites.

La composition peut être réalisée de façon déclarative à travers le langage *ADL Fractal* [ADL] qui permet de définir des composants et de composer des instances de ces derniers. La composition visuelle est supportée dans un outil appelé *FractalGUI* [FGUI].

2.1.2.4.3. Framework du composant

Lorsque le contrôleur agit sur le contenu, il joue un rôle similaire à celui du conteneur. La spécification Fractal définit différents contrôleurs :

- Le contrôleur d'attributs permet de configurer les attributs d'un composant.
- Le contrôleur de liaisons permet de créer/rompre une liaison primitive entre deux interfaces de composants.
- Le contrôleur de contenu permet d'ajouter/enlever des sous-composants au contenu d'un composant composite.
- Le contrôleur de cycle de vie permet de contrôler les principales phases comportementales d'un composant, par exemple, le démarrage et l'arrêt de l'exécution du composant.

Le modèle de composant Fractal est un modèle générique, car il n'est pas orienté vers un domaine d'application particulier. Il supporte plusieurs langages de programmation. La composition des composants est réalisée de façon déclarative durant la phase de développement.

2.1.2.5. Le modèle de composant OSGi

OSGi [OSGi] est un modèle de composants proposé en 1999 par l' OSGi™ Alliance. Il définit un modèle et un framework de composants permettant la livraison de services administrés dans des réseaux résidentiels ou autres types d'environnements restreints (voitures, *etc.*). Une caractéristique importante de ce modèle est qu'il permet de tenir compte les évolutions dynamiques des applications.

Dans OSGi, les services sont livrés et déployés dans des unités appelées *bundles*. Un bundle correspond à un fichier JAR contenant des classes Java et des ressources. Notons qu'un bundle représente l'unité de déploiement associée au modèle OSGi, et est utilisé pour conditionner les composants. Les composants sont appelés *services* dans la spécification.

2.1.2.5.1. Définition du composant et de ses interfaces

Un composant ou un service est une classe Java qui implémente une ou plusieurs interfaces. La définition des interfaces du composant et de sa classe d'implémentation est identique à un programme Java classique. Chaque bundle peut posséder un ou plusieurs composants qu'il exporte. Le fichier manifeste associé à chaque bundle décrit l'ensemble des propriétés du bundle et les services requis et fournis par les composants qu'il contient. Nous voyons donc que d'une manière générale, la description des fonctionnalités fournies et

requises par un composant donné, n'est pas déclarée explicitement, et est cachée dans l'implémentation. Un bundle peut également contenir une classe spécifique appelée *activateur*. Cette classe définit deux méthodes `start()` et `stop()` qui sont appelées au démarrage et à l'arrêt des bundles, respectivement. La méthode `start()` enregistre les différents composants inclus dans le bundle, localise et obtient les références vers les composants requis.

2.1.2.5.2. Composition des composants

Au cours du développement des applications OSGi, la phase de composition n'existe pas comme une étape explicite. La plateforme de service OSGi permet de changer dynamiquement les relations entre composants sans avoir à arrêter puis relancer le système. Elle propose une architecture de service qui permet aux composants de se rechercher, se trouver et collaborer à la volée. L'application se construit donc progressivement au fur et à mesure que les composants apparaissent ou disparaissent. Autrement dit, le concept d'application n'apparaît que dans la phase d'exécution et peut changer dans le temps.

2.1.2.5.3. Framework du composant

Le framework OSGi s'exécute au dessus de la machine virtuelle (JVM). Le déploiement et l'exécution des services OSGi se font au dessus du framework OSGi (Figure 7).

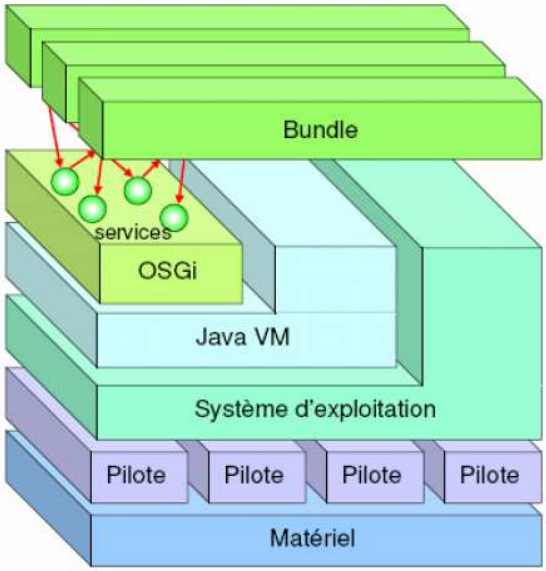


Figure 7. L'environnement d'exécution des composants OSGi [OSGi]

Le framework fournit un certain nombre de fonctionnalités de base et des mécanismes d'administration permettant de réaliser l'installation, l'activation, la désactivation, la mise à jour et la désinstallation des bundles de façon continue. Lorsque le bundle est actif, il peut publier ou découvrir des services et se lier avec d'autres bundles à travers un registre de services fourni par la plate-forme.

La Figure 8 présente les différents états du cycle de vie d'un bundle :

- *Installed* : le bundle est juste installé au sein du framework OSGi mais la résolution des dépendances n'est pas encore réalisée.
- *Resolved* : le bundle est installé et la résolution des dépendances est juste réalisée.

- *Starting* : le bundle est en train de démarrer. Cet état correspond à un état transitoire entre les états *Resolved* et *Active*.
- *Active* : le bundle est démarré avec succès et tous les services qu'il exporte sont disponibles pour les autres bundles.
- *Stopping* : le bundle est en train de s'arrêter. Cet état correspond à un état transitoire entre les états *Active* et *Resolved*.
- *Uninstalled* : le bundle est désinstallé et ne peut plus passer à un autre état.

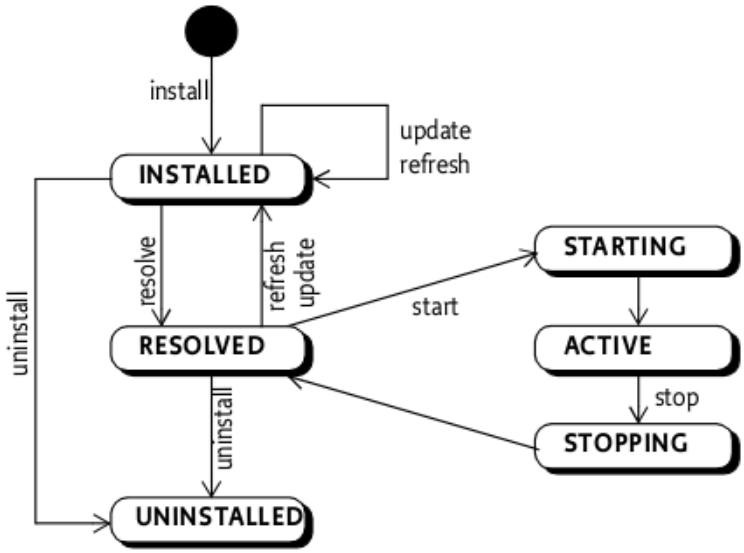


Figure 8. Cycle de vie d'un bundle OSGi [OSGi]

En bref, il est important de noter que la composition des composants OSGi se fait de manière dynamique et que les applications sont construites de façon incrémentale, au fur et à mesure que les composants sont installés ou désinstallés.

2.1.3. Synthèse sur les modèles de composants

Dans les paragraphes précédents, nous avons abordé l'approche de programmation par composants. Nous avons commencé par présenter des définitions de base, ensuite nous avons brièvement décrit quelques modèles de composants. Un comparatif résumé des différents modèles de composants étudiés (.NET, EJB, CCM, Fractal, OSGi) du point de vue de leurs avantages et de leurs inconvénients, est présenté dans le Tableau 1.

L'étude de ces différents modèles de composants nous a permis de voir comment les composants sont implémentés, leurs interfaces, la composition des composants et leur environnement d'exécution. Il est important de noter que les services des composants ne sont pas toujours définis explicitement. Par exemple, les services requis et fournis sont cachés dans l'implémentation des composants dans le cas du modèle OSGi. Par conséquent, l'assemblage des composants par un tiers est une opération délicate. Cependant, ce modèle de composant fournit une caractéristique très intéressante qui permet de prendre en compte les évolutions dynamiques des applications, c'est-à-dire qu'une application peut se construire de manière incrémentale au fur et à mesure que les composants s'installent ou se désinstallent. Un autre aspect à noter est que certains modèles de composants sont très spécialisés par rapport à leur

domaine d'application (EJB et .NET) tandis que d'autres modèles sont plus généraux (Fractal, CCM, OSGi).

Notre objectif de travail est de proposer des solutions générales, pouvant s'appliquer à tout modèle de composant logiciel. Donc, dans les chapitres 3 et 4 suivants, les descriptions de travail s'appuient sur un modèle de composant général, dans lequel la seule contrainte concerne l'existence d'interfaces requises et d'interfaces fournies. Toutefois, pour les aspects expérimentaux, nous avons utilisé des modèles concrets de composants pour montrer notamment la faisabilité de notre approche sur des modèles de composants différents. Parmi les modèles libres et généraux, le modèle Fractal, de par sa récursivité, nous a paru peu approprié à une étude préliminaire du problème du test inter-composant et du diagnostic. Nous avons donc utilisé deux modèles, à savoir CCM et OSGi. De par leur prise en main relativement facile et l'approche service qui y est très présente, ces modèles nous fournissent un cadre approprié pour nos travaux. Un atout supplémentaire du modèle OSGi concerne la composition dynamique.

Tableau 1. Résumé des modèles de composants

	.NET	EJB	CCM	Fractal	OSGi
Avantages	- Nombreux services supportés - Plusieurs langages	- Facilité de prise en main - Disponible	- Général - Disponible - Spécification claire - Plusieurs langages	- Général - Disponible - Plusieurs langages	- Dynamique - Général - Disponible
Inconvénients	- Solution propriétaire - Lourdeur du framework	- Une seule interface fournie - Spécifique au développement serveur	- Lourdeur du framework	- Composant récursif - difficile à appréhender pour le test	- Manque de clarté dans la définition des interfaces fournies et requises

Après avoir passé eu revue les principaux modèles de composants logiciels et sélectionné ceux qui nous paraissent les mieux adaptés à nos travaux, nous poursuivons ce chapitre de l'état de l'art en étudiant les principales approches de tolérance aux fautes existantes pour les applications à base de composants logiciels.

2.2. Tolérance aux fautes dans les applications à base de composants logiciels

Cette section débute par une brève présentation des concepts de base de la sûreté de fonctionnement. Quelques approches de tolérance aux fautes mises en œuvre dans les applications à base de composants logiciels sont ensuite décrites. Cette section se termine par la présentation de l'approche de diagnostic en ligne que nous projetons de développer dans le cadre de nos travaux.

2.2.1. Concepts de base de la sûreté de fonctionnement

"*La sûreté de fonctionnement d'un système informatique est la propriété qui permet à ses utilisateurs de placer une confiance justifiée dans le service qu'il leur délivre*" [LAP96]. Le *service* délivré par un système est son comportement tel que perçu par son ou ses utilisateurs; un *utilisateur* est un autre système (humain ou physique) qui interagit avec le système considéré.

Les notions attachées à la sûreté de fonctionnement sont regroupées en trois grandes classes : *attributs*, *entraves* et *moyens*, comme cela est illustré sur la Figure 9 ci-dessous.

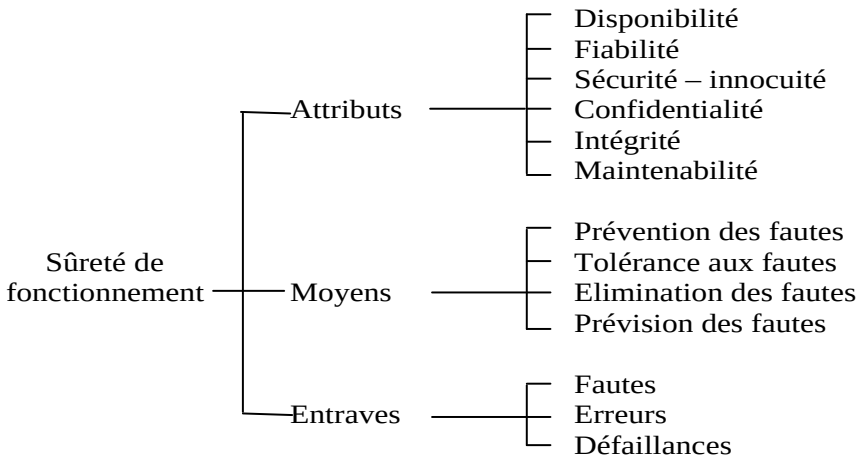


Figure 9. Attributs, entraves et moyens de la sûreté de fonctionnement [LAP96]

Les *attributs* de la sûreté de fonctionnement permettent d'exprimer les propriétés qui sont attendues du système, et d'apprécier la qualité du service délivré :

- La disponibilité évalue la capacité du système d'être prêt à rendre le service, elle est requise pour tout système, bien qu'à des degrés variables.
- La fiabilité cherche la continuité du service.
- La sécurité-innocuité est définie par la non-occurrence de conséquences catastrophiques pour l'environnement du système. Elle est requise pour des systèmes que l'on qualifie de critiques.
- La confidentialité exprime l'absence de divulgations non-autorisées de l'information.
- L'intégrité permet la non-occurrence d'altérations inappropriées du système.
- La maintenabilité est l'aptitude aux réparations et aux évolutions.

Les *entraves* à la sûreté de fonctionnement sont les circonstances indésirables, causes ou résultats de la non sûreté de fonctionnement. Trois concepts – *faute*, erreur, *défaillance* – sont nécessaires et suffisants pour les exprimer :

- Une *défaillance* du système survient lorsque le service délivré par le système ne correspond pas à la fonction attendue du système.
- Une *erreur* est la partie de l'état du système qui est inexacte et par conséquent susceptible d'entraîner une défaillance.

- Une *faute* est la cause adjugée ou supposée d'une erreur.

Dans la pratique, fautes, erreurs et défaillances peuvent s'enchaîner sans fin comme suit :

... –> faute –> erreur –> défaillance –> faute –> ...

Plus précisément, la défaillance d'un composant est considérée comme une faute pour le système qui l'englobe. Cette faute peut entraîner une erreur, auquel cas la faute est *active*, sinon elle est *latente*. Une erreur est aussi *latente* tant qu'elle n'a pas été reconnue comme telle, autrement dit tant qu'elle ne s'est pas propagée. La défaillance survient lorsque le service délivré est affecté.

Les *moyens* de la sûreté de fonctionnement sont les méthodes et les techniques permettant de fournir au système l'aptitude à délivrer un service conforme à l'accomplissement de sa fonction, et de donner confiance dans cette aptitude. Le développement d'un système sûr de fonctionnement passe par l'utilisation combinée de l'ensemble de ces méthodes qui peuvent être classées en quatre catégories principales :

- *Prévention des fautes* : comment empêcher l'occurrence ou l'introduction de fautes.
- *Tolérance aux fautes* : comment fournir un service à même de remplir la fonction du système en dépit de la présence de fautes.
- *Elimination des fautes* : comment réduire la présence (nombre, sévérité) des fautes.
- *Prévision des fautes* : comment estimer la présence, la création et les conséquences des fautes.

Les approches de prévention des fautes, d'élimination des fautes et de prévision des fautes ayant d'inévitables limitations d'un point de vue opérationnel lors de l'utilisation d'un système, celle de la tolérance aux fautes s'avère nécessaire pour tout système devant satisfaire certain attributs de la sûreté de fonctionnement (disponibilité, fiabilité, *etc.*). Dans le cadre de nos travaux, nous nous intéressons plus particulièrement à la tolérance aux fautes des applications développées à partir de composants logiciels.

La tolérance aux fautes est un processus qui met en œuvre quatre étapes distinctes, mais il n'est pas obligatoire qu'elles soient toutes présentes :

- *Détection* : permet de découvrir l'existence d'une erreur ou d'une défaillance.
- *Localisation* : permet d'identifier le point précis (dans l'espace ou le temps) où l'erreur (ou la défaillance) est apparue.

Ces deux étapes constituent la phase du *diagnostic*.

- *Isolation* : est l'action de confiner l'erreur pour éviter sa propagation à d'autres parties du système.
- *Réparation* : permet de remettre le système en état de fournir un service correct à l'aide des techniques de *remplacement* de composants, de *recouvrement* ou de *compensation* d'erreurs.

Ces deux dernières étapes constituent la phase de *reconfiguration*.

La compensation d'erreur par *vote majoritaire* conduit à un *masquage* de faute. Sa durée est plus faible que celle d'un remplacement de composant ou d'un recouvrement d'erreur.

C'est pourquoi, de nombreuses techniques de tolérance aux fautes reposent sur l'introduction d'éléments redondants appelés aussi répliques, dans un but de compensation d'erreur. On parle alors de *réplication passive*. La principale difficulté de cette approche réside dans la minimisation de cette redondance pour obtenir une fiabilité maximale tout en respectant les impératifs de coût et de disponibilité des ressources et en s'attachant à limiter la complexité du système résultant.

Il est possible de réduire, voire de supprimer totalement le recours à la réplication des composants, en dotant ceux-ci de capacités supplémentaires leur permettant de prendre en charge la phase du diagnostic. C'est cette approche que nous avons développée dans nos travaux par l'introduction de la notion de *composant logiciel testable en ligne*. Notre démarche est donc totalement différente des approches classiques de réplication des composants. Elle s'inspire en cela des travaux de diagnostic en ligne de composants matériels (processeurs) interconnectés par un réseau mais est, à notre connaissance, un travail original pour le cas des composants logiciels. C'est pourquoi, il nous a paru nécessaire de présenter, dans cet état de l'art le principe de base du diagnostic en ligne tel que proposé pour les composants matériels (voir paragraphe 2.2.4). Mais avant cela, nous avons jugé utile de compléter cet état de l'art par des exemples d'approches de tolérance aux fautes qui ont été introduites pour les systèmes à composants logiciels.

2.2.2. Quelques approches de tolérance aux fautes pour les applications à base de composants logiciels

L'utilisation des composants logiciels dans le développement d'applications informatiques étant relativement récente, peu de travaux ont été publiés sur les aspects de tolérance aux fautes concernant ce type d'applications. Nous décrivons ci-dessous les principaux travaux que nous avons rencontrés et qui concernent principalement les composants CCM. Comme nous le verrons, des différences importantes existent entre toutes les approches développées. Elles sont essentiellement dues à la couche du système dans laquelle sont intégrés les mécanismes de tolérance aux fautes, ainsi qu'à la technique d'intégration utilisée entre l'application et les mécanismes de tolérance aux fautes.

Après avoir décrit les travaux de tolérance aux fautes dans les applications à base de composants, nous cherchons à évaluer leurs caractéristiques sous l'angle de plusieurs critères. Ceci nous permet alors d'en dresser une comparaison.

2.2.2.1. Classification des approches de tolérance aux fautes dans les applications à base de composants

Les différentes approches de tolérance aux fautes dans les applications à base de composants peuvent être classifiées selon la couche du système dans laquelle sont intégrés les mécanismes de tolérance aux fautes. Comme le montre la Figure 10, nous pouvons distinguer :

- Des approches de niveau *système* : où les mécanismes sont intégrés dans les couches les plus basses, et qui ont accès à un certain nombre d'informations de niveau système.
- Des approches de niveau intermédiaire (*middleware ou intergiciel*) : qui sont les approches par *intégration*, qui, comme leur nom l'indique, intègrent les mécanismes de tolérance aux fautes dans l'intergiciel ou le serveur de composants.

- Et des approches de niveau applicatif :
  - Les *services/librairies*, qui implémentent certains mécanismes de tolérance aux fautes, à charge de l'utilisateur de les appeler et lier à son application.
  - Et enfin les systèmes *réflexifs* ou par *héritage*, qui se placent au niveau du *langage*. Ils utilisent des mécanismes du langage, réflexivité ou héritage afin de lier les mécanismes de tolérance aux fautes et l'application.

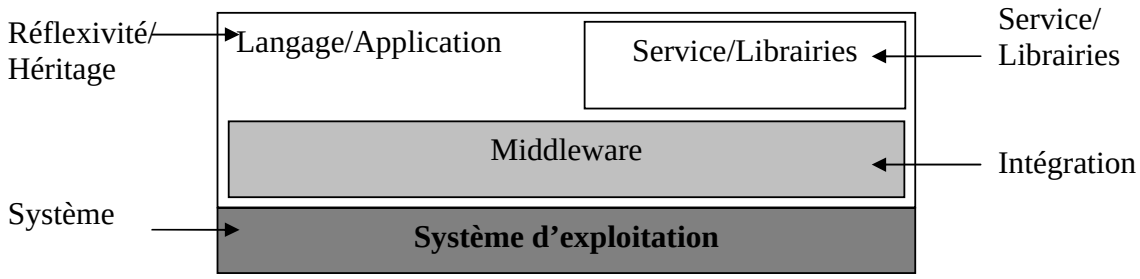


Figure 10. Différentes couches d'intégration de la tolérance aux fautes

Ci-dessous, nous décrivons plus en détail quelques approches développées dans chacun de ces niveaux.

2.2.2.2. Approches de niveau système

Les approches dites "système" proposent d'intégrer les mécanismes de tolérance aux fautes dans les couches basses du système que sont le noyau ou le système d'exploitation. Le fait d'être au coeur du système permet à ces mécanismes d'accéder à des informations qui sont cachées aux niveaux d'abstraction plus élevés. Le projet DECOS (Dependable Embedded Components and Systems) [DEC] qui est développé à l'université de Vienne est un exemple de tolérance aux fautes au niveau système.

DECOS est un projet qui développe une technologie de base pour une architecture distribuée intégrée afin de réduire le temps de développement, le coût de production et de maintenance et d'augmenter la fiabilité des applications distribuées embarquées, dans beaucoup de domaines d'applications.

Le projet DECOS est basé sur l'architecture temps déclenchée (Time Triggered Architecture - TTA) [KOP02] qui supporte une implémentation transparente de la tolérance aux fautes. Le TTA est prévu pour des applications temps réel critiques telles que la commande d'un avion ou d'un système de freinage dans les automobiles. La réplication active, qui permet la détection des erreurs (comme la duplication en vue de la comparaison par exemple) et le vote majoritaire qui permet de masquer les erreurs, sont les techniques de tolérance aux fautes les plus appropriées pour la satisfaction de ces conditions.

Dans le TTA, le mécanisme de tolérance aux fautes est implémenté dans une couche dédiée à la tolérance aux fautes (Figure 11). Un logiciel d'application peut fonctionner ainsi dans ce système avec ou sans tolérance aux fautes sans aucune modification préalable. Le mécanisme de tolérance aux fautes est transparent à l'application dans le TTA.

L'unité de tolérance aux fautes FTU comporte trois nœuds (redondance modulaire triple - TMR) [LAL01] qui fonctionnent en parallèle et présentent leurs résultats à un électeur qui prend une décision à la majorité (Figure 11).

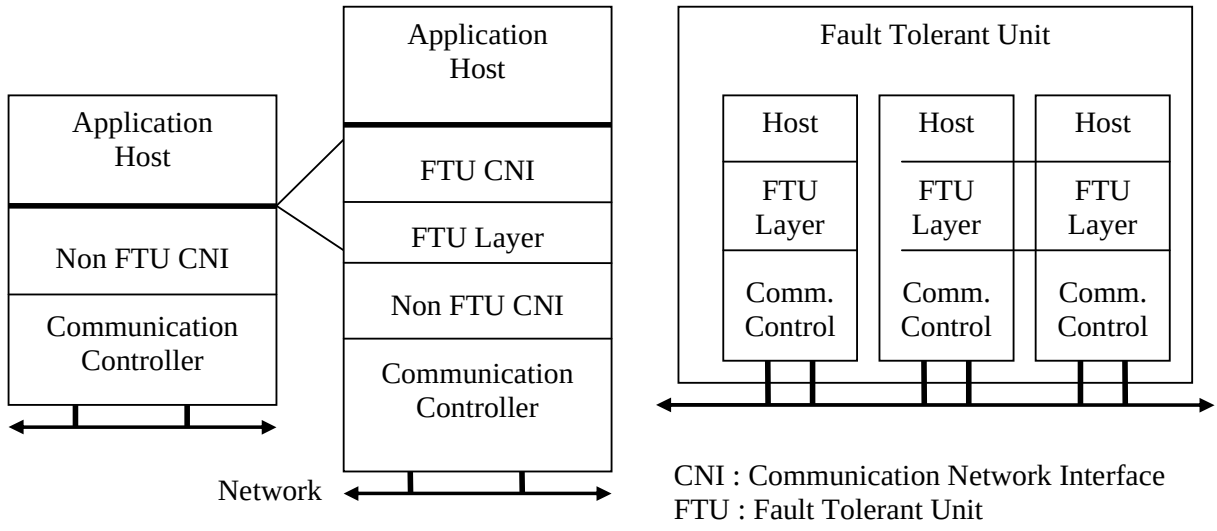


Figure 11. Expansion d'un nœud n'ayant pas de tolérance aux fautes à un nœud avec tolérance aux fautes et redondance triple modulaire dans le TTA [DEC]

Les trois répliques du logiciel de l'hôte fonctionnent de manière synchrone sur trois processeurs différents et produisent leur sortie simultanément (en fonction de la précision de la synchronisation d'horloge) à leur CNI de FTU. La couche FTU distribue les messages aux autres nœuds de la grappe. La couche FTU d'un nœud vote sur les messages entrants et présente le résultat de la majorité à son hôte à l'instant de la livraison. Périodiquement, chaque hôte doit sortir son état interne pour la même procédure de vote par d'autres nœuds du FTU. Un nœud doit attendre la réception d'un état interne voté avant de pouvoir participer à une application.

L'utilisation de cette approche de tolérance aux fautes de niveau système est transparente aux applications, mais elle est fortement liée à l'architecture temps déclenchée qui nécessite un fonctionnement synchrone.

2.2.2.3. Approches de niveau intermédiaire ou approches par intégration

Le principe de ces approches consiste à intégrer les mécanismes de tolérance aux fautes dans l'intergiciel ou le serveur de composants. Cette approche est utilisée dans le serveur de composants K2 développé par la compagnie iCMG [ICM]. Ce serveur fournit une infrastructure (côté serveur) pour le développement, l'assemblage, le déploiement, et la gestion des composants CORBA. Ainsi, le mécanisme de tolérance aux fautes est intégré dans le serveur de composants K2 (Figure 12).

Le serveur de composants K2 permet le développement et le déploiement des composants CORBA. Il fournit aussi des possibilités de gestion de la QoS (qualité de service) pour répondre aux exigences opérationnelles des applications critiques telles que la tolérance aux fautes, l'équilibrage de charge, les performances, la fiabilité, l'extensibilité, et la haute disponibilité.

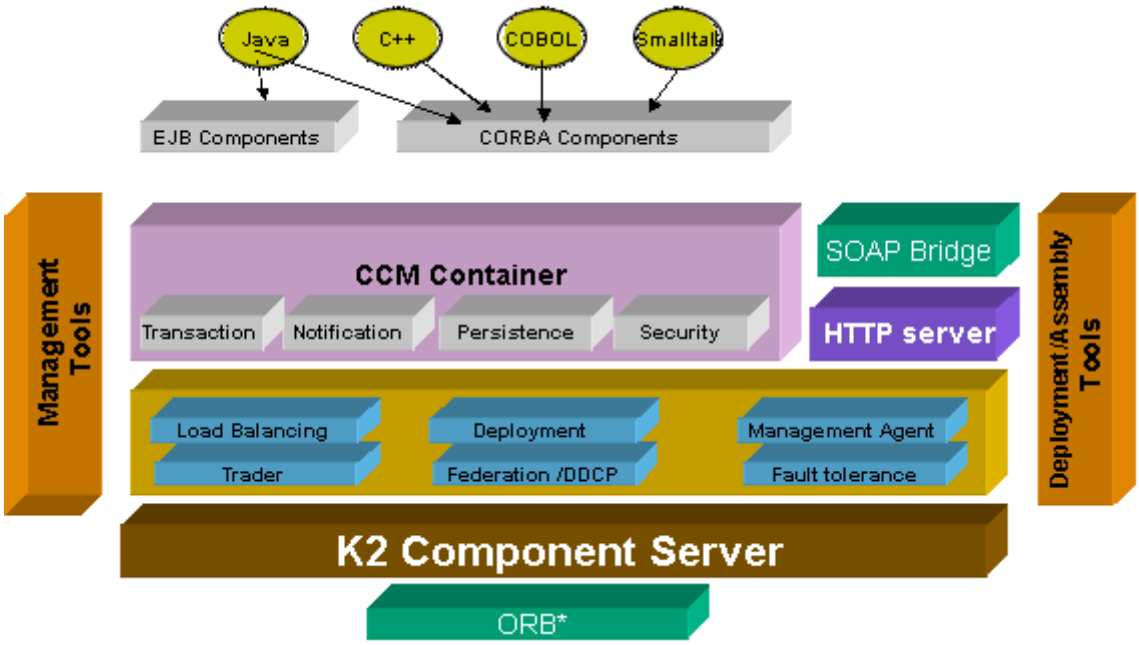


Figure 12. L'architecture de la plateforme iCMG [ICM]

Les fautes couvertes par le mécanisme de tolérance aux fautes K2 sont principalement des fautes du processus serveur. Elles sont provoquées par une erreur de programmation, qui provoque la terminaison des processus du système d'exploitation ou des fautes matérielles au niveau du serveur, qui entraînent inopinément l'arrêt de la machine et la perte des processus courants.

- Les fautes du processus serveur : la détection de ces fautes est assurée par le K2daemon, qui utilise les informations du système d'exploitation pour détecter un arrêt de processus. Le mécanisme de recouvrement des fautes utilise une fonctionnalité de base de CORBA pour réorienter un client vers un composant redémarré après la faute.
- Les fautes matérielles au niveau du serveur : le K2daemon permet de détecter une faute matérielle au niveau du serveur en utilisant un protocole fiable de multicast (RMP). Ce protocole emploie des messages "*Ping*" périodiques comme indication interne de fiabilité. Le client est informé de l'échec de la session par une exception appropriée. La détection de faute et le recouvrement lui-même sont implémentés d'une manière distribuée et décentralisée pour augmenter leur fiabilité. Un nœud rétabli est détecté par le RMP et dynamiquement lié au système.

Le serveur de composants K2 permet la détection des types de fautes ci-dessus et implémente le recouvrement des fautes sans recours à la réplication des composants mais en utilisant un processus démon et un protocole fiable de multicast. Le programmeur doit utiliser la fonctionnalité offerte dans le cas où il souhaite implémenter une application complètement récupérable. Cela signifie que le recouvrement d'une faute doit être considéré à la conception de l'application et au moment de l'exécution. Les mécanismes de tolérance aux fautes ne sont donc pas transparents pour l'utilisateur de ce système.

Cette approche par l'intégration semble relativement non portable parce qu'elle est intégrée dans un serveur de composants spécifique.

2.2.2.4. Approche de niveau applicatif

2.2.2.4.1. Approches de type librairies

Les librairies de tolérance aux fautes utilisent les services du système pour fournir des briques de base que l'application peut utiliser pour implémenter ses propres mécanismes de tolérance aux fautes. Par exemple, les librairies de tolérance aux fautes du projet CLEOPATRE (Composants Logiciels sur Etagères Ouverts Pour les Applications Temps-Réel Embarquées) [CLE][SIL06], qui a été développé à l'Institut de Recherche en Informatique de Nantes, fournissent à l'utilisateur des mécanismes de tolérance aux fautes à reprise par recouvrement pour des processus Linux.

Ce projet apporte des solutions au développement d'applications temps réel embarquées par la mise à disposition, après vérification auprès du monde industriel et universitaire, d'une bibliothèque de composants logiciels libres et ouverts s'appuyant sur Linux. Le modèle "source ouvert et libre" doit pouvoir s'appliquer désormais à un noyau temps réel embarcable. Il s'agit donc d'offrir, sous forme de COTS à code source ouvert, des fonctionnalités de niveau noyau innovantes en matière de tolérance aux fautes, d'ordonnancement et de communication, assorties d'une gamme d'utilitaires (méthodologie d'utilisation, algorithmes de contrôle-commande, algorithmes de vision temps réel, *etc.*).

Dans ce projet, les composants sont contenus dans des modules logiciels capables d'appeler les services d'autres composants pour servir les applications et/ou d'autres composants. Ils sont assemblés dynamiquement par le système Linux et programmés en langage C. Autrement dit, un composant est un module Linux fournissant des services.

CLEOPATRE propose une bibliothèque de composants destinés à la gestion des fautes temporelles en utilisant deux mécanismes, respectivement qualifiés de "mécanisme à échéance basé sur une redondance logicielle" et "méthode de calcul imprécis basée sur une algorithmique itérative". Ces mécanismes peuvent en outre être utilisés en vue d'une reconfiguration dynamique en cas de panne matérielle dans un système réparti. L'implantation de ces techniques soulève des problèmes liés à l'arrêt prématuré des tâches, à la gestion des points de reprise, à l'évaluation de la durée d'exécution d'une tâche, *etc.*

Le *mécanisme à échéance* assure un fonctionnement minimum de l'application. Pour cela, chaque tâche existe en deux versions : une version principale qui n'a aucune contrainte particulière et une version de secours dont l'utilisateur doit fournir la durée maximale d'exécution. Le mécanisme à échéance commence par prévoir une séquence statique des versions de secours selon un algorithme d'ordonnancement. Il s'agit d'une représentation du fonctionnement minimal garanti. Si une version primaire arrive à son terme, la version de secours associée devient inutile et est annulée. Si une version primaire ne peut pas se terminer avant le début de sa version de secours, elle est abandonnée pour laisser le processeur à sa version de secours.

La *méthode du calcul imprécis* se base sur l'hypothèse qu'une tâche est décomposée en deux sous-tâches respectivement qualifiées de partie obligatoire et partie optionnelle. Le calcul imprécis est donc d'une grande utilité pour l'implantation d'algorithmes itératifs où le rôle de la partie optionnelle consiste à accroître la précision du résultat fourni par la partie obligatoire. La partie obligatoire représente la durée minimale d'exécution nécessaire pour obtenir un résultat acceptable. Un temps de traitement additionnel contribuant à affiner ce résultat est représenté par la partie optionnelle. La partie obligatoire doit s'exécuter dans le respect de son échéance. Et, comme pour la méthode du mécanisme à échéance, un certain

pourcentage pour les parties optionnelles doit être assuré, de manière à maintenir la qualité du suivi du procédé.

L'utilisation de telles librairies au sein d'applications à composants n'offre ainsi que peu d'avantages. Non seulement l'utilisateur doit les adapter pour qu'elles gèrent correctement les références entre composants et autres particularités de son système, mais il doit aussi implémenter les mécanismes de tolérance aux fautes au sein de son application.

2.2.2.4.2. Services pour la tolérance aux fautes

Une caractéristique de base de l'architecture CORBA est le développement des mécanismes non-fonctionnels ou récurrents sous la forme de services. Ces services implémentent chacun une interface claire, définie en IDL et ainsi peuvent être facilement utilisés par les développeurs. Des projets qui emploient cette approche en vue de réaliser la tolérance aux fautes dans les composants CORBA sont étudiés dans les paragraphes qui suivent.

a. Le modèle AFT-CCM

Le modèle AFT-CCM (Tolérance aux fautes adaptative pour le modèle de composant CORBA) [FRA03][FAV03], qui est développé à l'université de Santa Catarina au Brésil, propose un service pour la gestion de la réplication des composants CORBA. Le fonctionnement de ce service est étroitement lié à la QoS définie par l'utilisateur. La configuration est gérée en utilisant un ensemble de composants logiciels non fonctionnels qui sont responsables d'implémenter des techniques de tolérance aux fautes, de définir et de contrôler le comportement des services de réplication. Ce modèle permet à l'utilisateur de spécifier les exigences de QoS, le nombre de répliques, la technique de réplication, *etc*.

La structure du modèle de tolérance aux fautes AFT-CCM qui comporte des composants non-fonctionnels est illustrée dans la Figure 13. Ces composants, qui sont disposés dans 4 hôtes d'un système distribué, configurent et dirigent les composants de l'application, la réplication des composants, et la technique de réplication afin de réaliser les niveaux désirés de QoS.

Dans l'exemple de la Figure 13, un composant est identifié comme serveur primaire et est placé sur l'hôte 2. Sa réplication est placée sur l'hôte 3 (le serveur de secours). Chaque technique de réplication a un protocole pour maintenir la cohérence entre les différentes répliques. Dans ce modèle, ce protocole est implémenté par le composant *coordinateur de réplication* (voir hôte 2 et hôte 3 sur la Figure 13). De cette façon, quand on utilise par exemple la technique de réplication passive, les états des répliques sont synchronisés par le coordinateur de réplication en utilisant des mécanismes de transfert d'état. La technique de réplication est facile à changer pendant l'exécution en activant à chaque fois le coordinateur adéquat.

Le client accède aux services du composant via un *connecteur*, qui cache au client les changements de configuration de l'application. Par exemple, le remplacement de la réplique primaire par la réplique secondaire n'affecte par le comportement du client. Si la réplique primaire échoue, le connecteur réoriente les appels vers la réplique de secours.

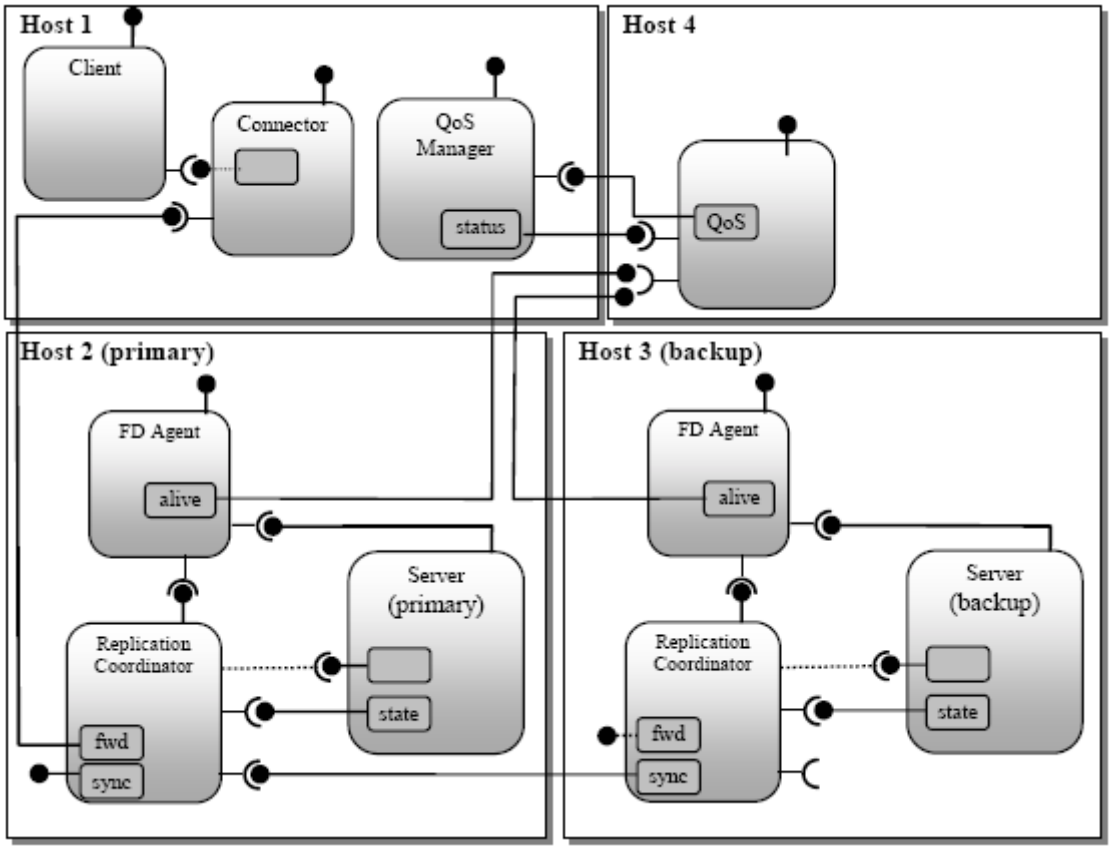


Figure 13. Vue globale du modèle AFT-CCM [FRA03]

Un manager de tolérance aux fautes adaptative (*Adaptive Fault-tolerance Management - AFT manager*) est crée pour chaque composant avec des besoins (exigences) de tolérance aux fautes. Le manager AFT définit la configuration, en fonction de la QoS exigée de l'application et de la fréquence des fautes. Une reconfiguration de l'application est envisagée quand des données rassemblées par le manager AFT indiquent que la configuration courante n'est pas appropriée pour assurer la QoS exigée par l'application. Une telle situation peut être due à un excédent de ressources (répliques) ou à l'utilisation d'une technique de réplication non adaptée.

Les fautes sont surveillées par les agents de détection de faute (agent FD, Figure 13). Chaque composant réplique et coordinateur de réplication est lié à un agent FD qui est responsable d'envoyer des données de surveillance au manager AFT. En employant les agents FD, deux niveaux de fautes sont détectés : des fautes de l'hôte - au cas où l'agent FD cesse de répondre à AFT et des fautes du composant - quand un composant cesse de répondre aux appels de l'agent FD. Ce modèle suppose que la communication entre les composants est fiable.

Le manager de la QoS permet à l'utilisateur de spécifier les différents niveaux de QoS en lien avec la fiabilité de l'application. Les demandes spécifiées par l'utilisateur sont transmises au manager AFT qui les interprète et définit une configuration appropriée pour l'application, par exemple le nombre de répliques et la technique de réplication qui sera utilisée. Le manager AFT essaie de maintenir le niveau voulu de QoS avec les ressources actuellement disponibles. Il change la configuration du système quand il détecte que les demandes ne sont pas satisfaites. En outre, des demandes peuvent être modifiées à tout moment par le manager

de QoS. Tout changement des conditions est transmis au manager AFT qui devra peut-être reconfigurer l'application afin d'imposer les nouvelles exigences de QoS. Le manager de QoS informe également l'utilisateur de la configuration courante, par exemple le nombre de répliques, leur localisation, la technique de réplication, et la fréquence des fautes dans l'application.

Le modèle AFT-CCM implémente la tolérance aux fautes adaptative de manière complètement transparente pour l'application. Le modèle AFT-CCM utilise un connecteur générique qui est indépendant de l'interface du composant répliqué, pour éviter qu'elle soit modifiée dans différentes applications.

b. Le projet ARCAD

Dans [MAR02a][MAR02b], une approche pour la réplication de composants CORBA, qui est intégrée dans le projet ARCAD de l'INRIA Rhône-Alpes, est présentée. Cette approche utilise des objets d'interception qui sont responsables de capturer les invocations faites au composant afin de déclencher des actions nécessaires pour la gestion de la réplication.

Cette approche utilise un protocole d'uniformité pour maintenir l'uniformité entre les répliques. Autrement dit, le protocole d'uniformité définit les relations d'uniformité entre les copies et fournit les traitements pour entretenir la validité de ces relations.

Dans ce projet, la gestion de la réplication des composants est basée sur :

- Les objets d'interception pour capturer les invocations de composants et exécuter les traitements d'uniformité. Les objets d'interception permettent d'intégrer la gestion d'uniformité sans modifier le code des composants. Ils interceptent les invocations et déclenchent les actions d'uniformité.
- Un objet d'interception est un objet qui implémente les mêmes interfaces que le composant correspondant, mais son code contient l'implémentation du protocole d'uniformité. Le code fonctionnel d'un composant est géré dans un objet séparé et est référencé par l'objet d'interception pour propager les invocations. L'implémentation des objets d'interception dépend du protocole d'uniformité spécifié choisi par l'application.
- Les liens d'uniformité pour interconnecter les répliques : le protocole d'uniformité définit les relations d'uniformité entre les copies et fournit les traitements pour entretenir ces relations valides. Ces traitements requièrent le lien d'uniformité pour propager les actions d'uniformité (Figure 14).
- Les fonctions accédant aux composants pour extraire les données internes : la plupart des protocoles d'uniformité accèdent aux données internes du composant. Dans le prototype de cette approche, le protocole d'uniformité préserve l'encapsulation principale du composant mais prend avantage de la gestion de l'état du composant pour laisser l'implémentation accéder aux primitives d'état du composant. Le protocole d'uniformité peut alors utiliser ces primitives et supprimer l'implémentation détaillée du composant.

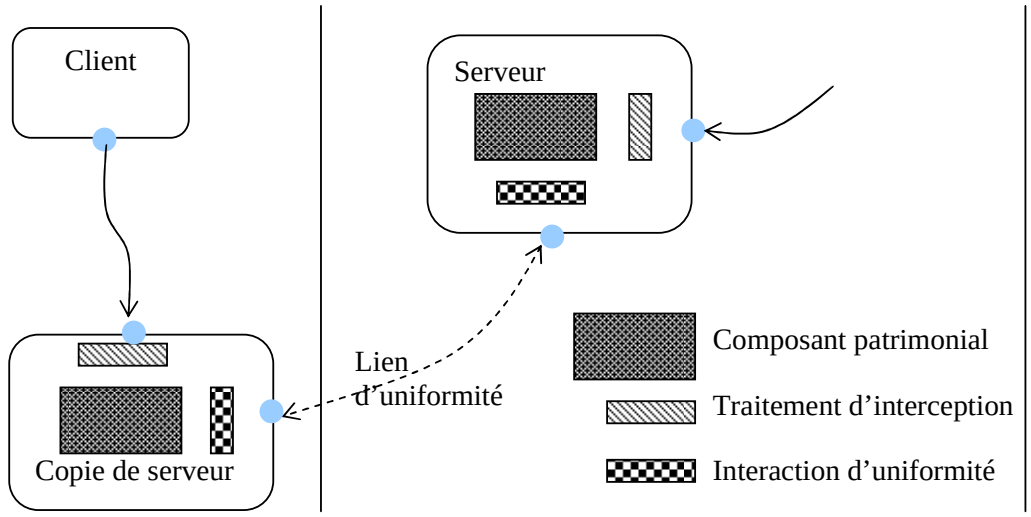


Figure 14. Un schéma simple de réplication dans le projet ARCAD [MAR02b]

Dans la Figure 15 ci-dessous, le composant à répliquer et le protocole d'uniformité sont représentés par leurs définitions IDL et leurs implémentations. La définition du protocole d'uniformité déclare les interfaces des liens d'uniformité impliqués dans la coordination des copies du composant répliqué. Le protocole implémente ces interfaces ainsi que les interfaces fonctionnelles du composant afin d'intercepter les invocations correspondantes. Le composant réplicable implémente les interfaces et compose les implémentations du composant initial et du protocole d'uniformité choisi.

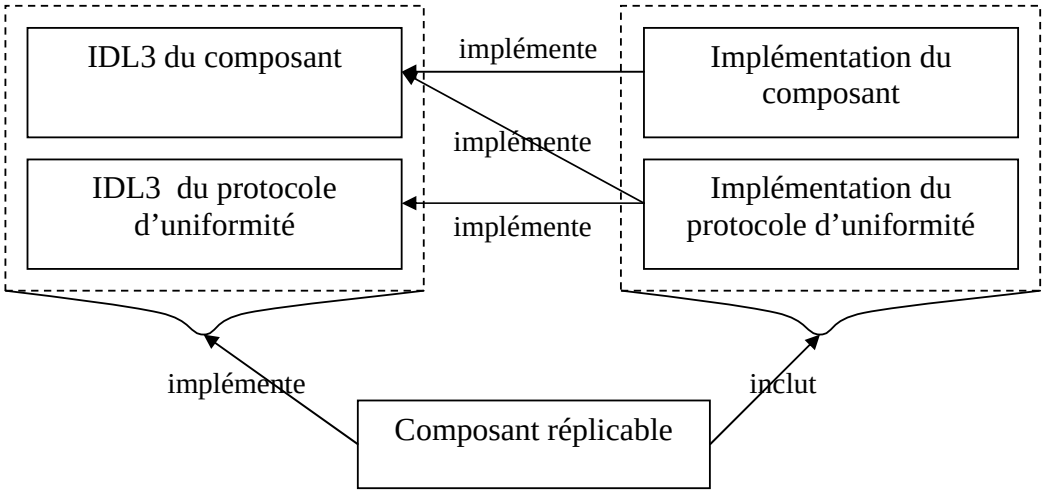


Figure 15. Composant réplicable [MAR02a]

Ces objets d'interception ont la même interface que le composant répliqué. Ceci implique que pour chaque nouveau composant à répliquer, un nouvel objet d'interception avec la même interface est ajouté dans l'application, en plus de la ou des répliques du composant. Toutefois, ceci se fait de manière transparente par rapport à l'utilisateur.

c. Le projet COACH

L'objectif du projet COACH (Component based Open source Architecture for distributed telecom applications) [COA] du consortium ObjectWeb est d'établir un canevas de composant qui peut rapidement transformer les composants du niveau modèle, architecture ou

conception, au niveau d'exécution et les déployer efficacement sur des plateformes matérielles distribuées. Ceci permet aux développeurs de construire des applications en assemblant des composants et en utilisant des composants existants, ce qui réduit les coûts de développement et le temps de mise sur le marché. Le projet COACH s'intéresse principalement aux applications de télécommunications, mais les résultats sont également applicables aux applications à grande échelle et critiques, telles que la gestion du trafic, la défense, ou les applications financières.

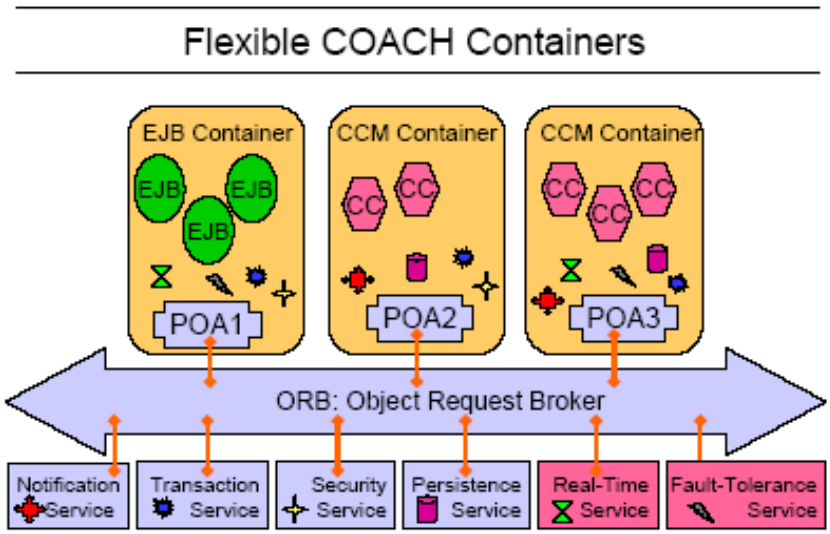


Figure 16. L'architecture du projet COACH [COA]

Le projet COACH propose une nouvelle technologie de conteneur d'exécution flexible permettant la conception, le développement, et le déploiement des conteneurs standards CCM (Figure 16). Il assure la gestion de manière transparente des services du système tels que l'activation, la communication, la notification, la sécurité des transactions, et la persistance. Il fournit également des conteneurs spécifiques aux nouvelles applications, par exemple, la gestion de la qualité de service, le temps réel, et les services de tolérance aux fautes. Ceci permet de développer des applications réparties fiables et adaptables appropriées au domaine des télécommunications.

Cette dernière approche de niveau service illustrée à travers les projets AFT-CCM, ARCAD et COACH nous semble bien plus transparente que les précédentes. Elle s'intègre facilement à l'application, principalement par l'utilisation de services de tolérance aux fautes fournis.

2.2.2.4.3. Approche de niveau langage

Cette approche est illustrée principalement par l'utilisation de la *programmation par aspects* [FIL04]. Un programme orienté aspects est constitué de deux parties : un programme de base, qui définit les préoccupations fonctionnelles de l'application, et un ou plusieurs programmes séparés, écrits dans des langages éventuellement spécialisés, qui définissent les préoccupations non-fonctionnelles à associer (par exemple la tolérance aux fautes). La coordination entre les deux parties se fait par un procédé d'insertion appelé tissage (*weaving*), qui peut être statique ou dynamique. Le principe est de spécifier les endroits où le tissage doit être effectué dans le code fonctionnel. Ces endroits sont appelés des points de tissage (*join point*).

La programmation par aspects a été utilisée pour réaliser la tolérance aux fautes dans les grappes d'applications Internet J2EE [BOU05]. Ce travail est basé plus précisément sur le langage AspectJ [ASP]. Le système développé permet de détecter et de masquer les fautes par des tentatives de re-exécution de la requête non satisfaite sur d'autres serveurs pairs.

Ce système, appelé JSR (Java Server Recovery) détecte et traite les fautes au niveau de chaque composant logiciel qui est responsable de traiter une requête sur un serveur J2EE. Tout d'abord, il détermine les points du composant où des traitements relatifs à la tolérance aux fautes doivent être effectués et intègre ensuite à ces points le traitement de la détection et du masquage de fautes. Toutefois, cette approche manque de transparence vis-à-vis du développeur.

2.2.3. Comparaison des différentes approches

Dans cette section, nous cherchons à comparer les différentes approches de tolérance aux fautes analysées précédemment selon les critères suivants : transparence, séparation, composabilité, visibilité, réutilisabilité et portabilité. Ces critères sont choisis car ils permettent le renforcement de l'approche de développement par composants. Les points suivants définissent ces critères et les évaluent pour chacun des systèmes que nous avons présentés. Enfin, le Tableau 2 présente une synthèse de ces résultats.

- **Transparence** : la transparence est la capacité d'une entité à être "invisible". Dans notre cas, il s'agit de cacher à l'utilisateur les mécanismes non-fonctionnels du système dont les mécanismes de tolérance aux fautes.
  - Les approches à base de librairies du projet CLEOPATRE, par intégration du projet K2, et le système JSR utilisant la programmation par aspects, n'offrent pas un niveau de transparence satisfaisant. En effet, l'utilisateur participe activement à l'implémentation des mécanismes de tolérance aux fautes.
  - Les approches de niveau système du projet DECOS et par service utilisée par le projet AFT-CCM, le projet ARCAD et le projet COACH sont transparentes. Dans les approches service, l'implication de l'utilisateur se limite à faire appel aux services de tolérance aux fautes quand il le juge nécessaire.
- **Séparation et composabilité** : la séparation correspond au degré d'indépendance existant entre l'implémentation des mécanismes de tolérance aux fautes et celle de l'application. Séparation et transparence sont des notions intimement liées. En effet, si l'approche n'est pas transparente, elle requiert une utilisation explicite des mécanismes dans l'application, ce qui nuit à l'indépendance et donc à la séparation entre ces entités. La composabilité correspond à la possibilité de combiner plusieurs mécanismes de nature différente. Evaluer la composabilité d'une approche n'a donc de sens que lorsque celle-ci permet la séparation. En effet, avec une approche qui ne permet pas la séparation, l'utilisateur implémente ses propres mécanismes au sein de l'application, leur composition est donc implicite.
  - Lorsque l'utilisateur utilise des librairies de CLEOPATRE, les mécanismes de tolérance aux fautes sont fortement couplés à son application.
  - Avec le mécanisme de tolérance aux fautes par intégration du serveur de composants K2, ce couplage est moins fort mais reste néanmoins peu satisfaisant.

- En revanche, dans le système DECOS, le modèle AFT-CCM, le projet ARCAD et le projet COACH, l'implémentation des mécanismes de tolérance aux fautes est bien séparée de celle de l'application. Il semble toutefois que la composition des mécanismes ne soit pas prise en compte dans ces différents systèmes.

- **Visibilité** : La visibilité concerne la façon avec laquelle les mécanismes sont visibles et configurables. Si plusieurs types de réplication sont disponibles, la visibilité permet à l'utilisateur de sélectionner celui qui est le plus adapté à son système ou à son application. Ce critère n'est pas incompatible avec la transparence.

  En effet, il est important que l'utilisation des mécanismes soit transparente au programmeur de l'application (simplicité, efficacité, réutilisation du code applicatif), mais aussi que les mécanismes soient aisément accessibles du point de vue de leur configuration pour une application donnée.

  - Lorsqu'il utilise des librairies du projet CLEOPATRE, l'utilisateur implémente lui-même ses propres mécanismes et, par conséquent, les contrôle parfaitement. Cette approche fournit donc une visibilité à outrance. Cet aspect peut avoir des effets non prévisibles sur le bon fonctionnement des mécanismes de tolérance aux fautes.

- **Réutilisabilité** : La réutilisabilité correspond à la possibilité donnée au concepteur du système de dériver de nouveaux mécanismes à partir des précédents et de les intégrer facilement au système.

  - Lorsque les mécanismes sont implantés statiquement dans le système, la librairie de CLEOPATRE ou le serveur de composants K2, ils sont plus difficiles à réutiliser. C'est encore plus flagrant lorsque l'utilisateur participe à leur implémentation (mauvaise séparation).

  - Lorsqu'en revanche ils sont dynamiquement présents dans le système, comme c'est le cas pour les techniques à base de services, leur réutilisation et modification sont facilitées.

- **Portabilité** : l'approche est portable si elle ne repose pas sur un matériel, un système d'exploitation, un intergiciel, un support d'exécution ou un langage particuliers ou spécifiques. L'intérêt d'utiliser une application portable est évident : on pourra la réutiliser dans plusieurs environnements sans avoir à modifier ni l'approche, ni même l'implémentation des mécanismes ou de l'application.

  - L'approche DECOS et le serveur composant K2 figent les mécanismes dans une implémentation basée sur des systèmes d'exploitation et intergiciels particuliers, ils sont donc difficilement portables.

  - Les autres approches sont plus ouvertes à la portabilité car elles sont moins liées à une implémentation particulière.

Tableau 2. Propriétés des différentes approches de la tolérance aux fautes dans les applications à base de composants logiciels.

Approche	Transparence	Séparation	Visibilité	Réutilisabilité	Portabilité
Système	+	+	+	-	-
Intégration	-	-	+	-	-
Librairies	-	-	+	-	+
Services	+	+	+	+	+
Langage	-	+	+	+	+

L'analyse de ces différentes approches de tolérance aux fautes fait ressortir que l'approche service est celle qui satisfait aux principaux critères définis précédemment. Ces critères étant intimement liés aux performances du développement à base de composants logiciels, nous optons donc pour l'approche service dans nos travaux, et ce dans le but de satisfaire le plus possible aux exigences de la programmation à base de composants logiciels.

Notre objectif se précise ainsi et consiste à proposer, sous forme d'un service, une approche de test inter-composant et de diagnostic en ligne pour les applications à base de composants logiciels. Pour être complet, l'état de l'art doit alors investiguer les solutions de diagnostic en ligne et de test des composants logiciels. Si pour le second aspect des travaux ont été publiés et sont résumés ci-dessous, pour l'aspect relatif au diagnostic en ligne de composants logiciels à partir de tests inter-composants, aucune étude, à notre connaissance n'a été menée. Pour cet aspect, nous nous limiterons donc à présenter les principes de base tels qu'énoncés pour les systèmes matériels. Nous verrons dans les chapitres 3 et 4 leur adaptation aux composants logiciels.

2.2.4. La technique du diagnostic en ligne : cas des systèmes matériels

Le diagnostic en ligne à partir de tests inter-composants, initialement proposé pour des systèmes matériels par Preparata *et al.* [PRE67], permet de déterminer l'état du système vis-à-vis des fautes pouvant toucher ses composants. L'idée fondamentale est d'utiliser des tests inter-composants. Un composant est alors utilisé pour tester et déterminer l'état (fautif ou correct) d'un autre composant. Le processus de diagnostic consiste donc à analyser les résultats des tests pour en déduire l'état de tous les composants du système. Or, il est possible qu'un composant testeur soit lui-même défaillant et transmette par conséquent des résultats de test non fiables. L'enjeu principal du processus de diagnostic consiste alors à analyser l'ensemble des résultats de tests produits tout en éliminant ceux provenant de composants testeurs défaillants. Le fait que ceux-ci ne sont pas connus *a priori* représente la difficulté majeure de ce processus.

Ce problème a donné lieu à de nombreux travaux, essentiellement théoriques, dont quelques états de l'art ont été publiés [BAR93a][BEN97][LEE94]. Nous nous limitons donc à

présenter ci-dessous les principales caractéristiques des algorithmes de diagnostic proposés pour les systèmes matériels.

Le processus de diagnostic peut être *centralisé* ou *distribué*. Dans le diagnostic centralisé, un composant central supposé fiable contrôle l'exécution des tests inter-composants et calcule l'état du système à partir des résultats des tests. Le diagnostic distribué permet à tout composant du système de déterminer l'état du système complet à partir des informations de tests produites localement et de celles transmises par d'autres composants.

Selon la manière dont les tests sont alloués[3], les algorithmes de diagnostic peuvent être *statiques* ou *adaptatifs*. Dans les algorithmes statiques, les relations de test sont établies avant le processus de diagnostic. Au contraire, elles sont dynamiques dans la stratégie adaptative et dépendent des résultats de tests précédents. L'avantage principal des algorithmes adaptatifs est l'utilisation d'un nombre de tests réduit qui entraîne normalement une réduction du nombre de messages échangés et du temps d'exécution du diagnostic.

Nous présentons ci-dessous le principe de fonctionnement du diagnostic centralisé et du diagnostic distribué à travers deux algorithmes représentatifs. Il est important de noter ici que ces deux algorithmes se basent sur deux hypothèses fondamentales :

- Les fautes qui surviennent au niveau des composants sont permanentes.
- Un test exécuté par un composant fiable produit toujours un résultat correct et indique avec certitude l'état du composant testé, alors qu'un test produit par un composant défaillant n'est pas fiable et peut fournir des informations erronées sur l'état du composant testé.

2.2.4.1. Algorithme de diagnostic centralisé

Le principe du diagnostic centralisé proposé par Preparata *et al.* [PRE67] est simple. Cet algorithme fait l'hypothèse de la présence simultanée d'un maximum de t composants défaillants durant une session de diagnostic. Afin d'assurer la détection et la localisation des t composants défaillants, il est nécessaire et suffisant que chaque composant soit testé par au moins t autres composants différents, sachant que $N \geq 2t+1$, où N est le nombre total de composants dans le système.

Chaque composant testeur transmet directement ses résultats de test à un composant central, supposé fiable. Ce composant central analyse alors l'ensemble des résultats de tests et détermine l'état des composants du système. La satisfaction de la condition ci-dessus sur le nombre d'éléments fautifs garantit que le diagnostic établi est correct (un consensus est possible car les composants corrects sont majoritaires).

2.2.4.2. Algorithme de diagnostic distribué

L'algorithme distribué [BIA91] permet à tout composant de déterminer l'état du système à partir d'informations de test produites localement et de celles transmises par d'autres composants.

[3] Les relations de test entre les composants forment un graphe orienté, appelé *graphe de test*.

L'idée principale de cet algorithme est qu'un composant n'accepte que les informations en provenance des composants qu'il teste et qu'il trouve corrects. Globalement, un composant opère de manière périodique selon le schéma suivant :

- le composant C_i teste le composant C_j comme étant correct,
- le composant C_i reçoit des résultats de tests transmis par le composant C_j,
- le composant C_i teste une nouvelle fois le composant C_j,
- le composant C_i suppose que les informations de tests transmises par C_j sont valides si C_j est toujours correct.

(Les N composants du système sont numérotés C_0, C_1, C_2, …, C_{N-1}).

Cet algorithme de diagnostic distribué peut diagnostiquer jusqu'à N-1 composants fautifs simultanés, N étant le nombre de composants dans le système. Pour cela, l'hypothèse supposée d'un réseau d'interconnexion complet est faite et une stratégie de diagnostic adaptative est utilisée. Cet algorithme permet une amélioration considérable du nombre de tests par rapport au diagnostic centralisé comme montré dans [BIA91].

En bref, nous pouvons constater que 2 types d'hypothèses sont nécessaires au fonctionnement correct d'un algorithme de diagnostic :

- Une hypothèse sur la nature des fautes considérées. Ici, il s'agit de fautes permanentes qui se manifestent toujours tant qu'aucune action corrective n'est effectuée.
- Une hypothèse sur la connectivité du système qui est liée de près au nombre de composants pouvant être simultanément défaillants.

La nature du diagnostic (centralisé ou distribué) et la stratégie de test utilisée (statique ou adaptative) sont également des éléments importants.

Comme nous pouvons le voir, l'approche de diagnostic en ligne initialement développée pour les composants matériels et que nous projetons de développer pour les applications à base de composants logiciels, se fonde sur la notion essentielle de test inter-composant. Il nous semble donc important d'étudier les approches de test existantes pour les composants logiciels. Ceci constitue la dernière partie de cet état de l'art.

2.3. Le test des composants logiciels

2.3.1. Définitions de base

Le test consiste à exécuter un programme avec des entrées valuées, appelées *cas de test*, et à vérifier la conformité des sorties par rapport au comportement attendu [BEI90]. Sa mise en œuvre nécessite la détermination de niveaux de test, la sélection de cas de test et la vérification de l'exactitude des résultats observés (ou le problème de l'oracle).

2.3.1.1. Niveau de test

Les niveaux de test sont déterminés selon les notions de test unitaire, test d'intégration et test système [BEI90].

Test Unitaire : une unité est la plus petite partie de programme/logiciel pour laquelle on peut entreprendre le test. Typiquement, c'est une fonction ou une procédure écrite par un seul

programmeur. L'unité est vérifiée par rapport à la conception détaillée. Le test unitaire est fait généralement par le programmeur, indépendamment du reste du logiciel, ceci afin de s'assurer qu'il répond aux spécifications fonctionnelles et qu'il fonctionne correctement en toutes circonstances.

Test d'intégration : l'intégration est un processus par lequel des composants sont agrégés pour former un composant plus complexe. L'objectif du test d'intégration est de tester l'intégration et la communication entre les composants.

Test système : c'est la phase ultime du test, lorsque tous les composants du programme/logiciel ont été intégrés. Le test système est généralement conduit avec l'intégration du matériel, les expérimentations dans une plate-forme simulée ou sur site, afin de pouvoir vérifier les comportements et les interactions du système vis-à-vis de l'environnement opérationnel. Ce niveau de test est différent avec le niveau de test d'intégration, car il concerne tout le système, et non seulement les interactions entre les composants (et avec des systèmes externes). En plus de tester les comportements, le test système peut inclure le test de performance, d'utilisation de ressources, de sécurité, *etc*.

2.3.1.2. Sélection de cas de test

Sauf cas particulier, le test exhaustif sur toutes les entrées possibles n'est pas praticable. Il est donc nécessaire de sélectionner, d'une manière pertinente, un sous-ensemble du domaine d'entrée. Cette décision est effectuée à l'aide de critères de test, qui peuvent être liés à un modèle de structure du programme ou à un modèle des fonctions que le programme doit réaliser. Ces deux cas conduisent respectivement au test structurel et au test fonctionnel.

2.3.1.2.1. Test structurel

Le test structurel sélectionne les entrées du programme à partir de la structure du programme. De nombreux critères sont définis dans [BEI90]. Le modèle utilisé est le *graphe de contrôle*, éventuellement enrichi par des informations sur le flot des données dans ce graphe.

Le graphe de contrôle : il est construit à partir du code source de l'application. Il fournit une vue compacte de la structure de l'application. Les nœuds du graphe sont des blocs d'instructions qui sont exécutés toujours dans le même ordre. Les arcs entre les nœuds correspondent aux branchements conditionnels ou inconditionnels dans le programme. Une exécution peut alors être vue comme un parcours complet entre un (des) nœud(s) d'entrée et un (des) nœud(s) de sortie, le chemin suivi étant déterminé par les valeurs des entrées de test.

Le critère de sélection le plus sévère demande au moins une activation de chaque chemin exécutable entre un (des) nœud(s) d'entrée et un (des) nœud(s) de sortie. Un chemin est exécutable s'il existe des entrées qui l'activent.

D'une manière générale, le test structurel demande d'avoir le code. Dans le cadre d'applications à base de composants, ce n'est pas toujours évident.

2.3.1.2.2. Test fonctionnel

Il repose sur un modèle qui décrit le comportement attendu du programme. Le test fonctionnel comprend donc un ensemble de méthodes diverses, selon le formalisme utilisé pour modéliser le comportement du programme. Les formalismes les plus classiques sont les classes d'équivalence, les tables de décision, les spécifications algébriques, les langages

ensemblistes, les machines à états finis et les systèmes de transition étiquetés [BEI90]. Ces formalismes sont succinctement décrits ci-dessous.

Classes d'équivalence : dans cette méthode, le domaine d'entrée du programme est partitionné en un nombre fini de sous-domaines, en distinguant des plages de valeurs valides et invalides pour chacune des variables d'entrée. Le critère consiste à sélectionner un élément par classe. Cette méthode peut efficacement être complétée par un test utilisant des valeurs qui se situent aux frontières des différents sous-domaines.

Tables de décision : une table de décision est utilisée afin d'identifier les combinaisons d'entrées qui influent sur la logique du programme. Elle comprend deux parties: une liste de conditions ou prédicats sur les entrées, et une liste d'actions à entreprendre (sorties). Chaque colonne de la table définit une règle, qui lie une combinaison de valeurs de vérité des conditions à une liste d'actions attendues. Le critère de test associé à la table consiste à activer au moins une fois chaque règle.

Spécifications algébriques : une spécification algébrique est définie par la donnée d'un ensemble de sortes d'éléments, de profils d'opérations et d'axiomes. La donnée des sortes et des profils d'opérations forme la signature de la spécification. Les axiomes imposent un comportement attendu des opérations. Tester un axiome consiste à instancier ses variables, et à vérifier que le programme satisfait la formule obtenue. Le choix des instances d'axiomes s'effectue à l'aide d'hypothèses de sélection.

Langages ensemblistes. Les langages ensemblistes, tels que Z, B, *etc.*, permettrent de modéliser l'espace d'état d'un logiciel, et les opérations faisant évoluer cet état. Les méthodes de test associées comportent deux aspects : la sélection de test pour chaque opération analysée individuellement (par exemple, les critères de couverture du prédicat avant/après de l'opération) et des séquences d'appels aux opérations.

Machines à états finis : les machines à états finis (appelées aussi automates finis) sont utilisées pour modéliser des comportements séquentiels. Une machine à états finis est un graphe, dont les nœuds représentent les états du système, et les arcs représentent les transitions entre les états. Les transitions sont étiquetées avec l'entrée qui les provoque et les sorties qui sont générées lors du franchissement de la transition.

Systèmes de transition étiquetés : les systèmes de transitions étiquetés (*Labelled Transitions Systems – LTS*) sont un formalisme pour modéliser des systèmes communicants. En pratique, on ne spécifie pas directement selon ce formalisme: la spécification est donnée dans un langage de haut niveau (par exemple SDL) dont la sémantique est définie par un LTS. Les synthèses de test sont effectuées par composition du LTS avec des objectifs de test, qui décrivent des séquences d'événements à tester.

2.3.1.3. Problème de l'oracle

Le test pose un problème de mise en œuvre qu'il ne faut pas négliger : le problème de l'oracle, ou comment décider de l'exactitude des résultats observés, fournis par le programme en réponse aux entrées de test.

Les solutions les plus satisfaisantes sont basées sur l'existence d'une spécification formelle du programme sous test. La spécification est utilisée pour déterminer les résultats attendus, soit lors de la sélection des entrées de test, soit *a posteriori*.

Le test dos-à-dos peut constituer une autre solution. Les sorties sont comparées à celles fournies par une autre version du programme, développée indépendamment à partir de la

même spécification. Une discordance indique la présence d'une faute dans une des deux versions ou dans les deux. C'est la spécification qui joue alors le rôle de référence pour identifier (manuellement) la (les) version(s) incorrecte(s).

D'autres exemples d'oracles peuvent être : un tableau d'exemples, les connaissances du programmeur sur le fonctionnement du logiciel, des versions correctes précédentes, le contrôle de cohérence entre différentes données, le contrôle d'appartenance à une plage de valeurs, *etc.* En cas de test déterministe, il peut être plus facile, puisque le testeur choisit lui-même les entrées de test : il lui suffit alors de sélectionner des entrées pour lesquelles il connaît la réponse correcte attendue. Cependant, cette méthode conduit naturellement à restreindre le test à des entrées simples.

2.3.2. Les approches de test des composants logiciels

Les composants logiciels posent des problèmes particuliers de test et de validation car ils s'exécutent dans des environnements qui ne sont pas connus *a priori* et qui peuvent évoluer dynamiquement. En effet, les composants logiciels sont développés indépendamment des applications qui les utilisent. Ils sont employés principalement dans des applications où la qualité de service et l'adaptation dynamique sont souvent requises. Ces contraintes et l'analogie avec les technologies de composants matériels conduisent la majorité des auteurs à proposer des approches de *test intégré* (*Built-in test*) [WAN00][HOR02][ATK02] ou *autotest* (*Self-test*) [TRA99][BEY03]. Dans ces approches, le composant contient lui-même des éléments de code non directement fonctionnel (assertions, pilotes de test, *etc*.) qui vont lui permettre de se tester lui-même ou de valider ses fonctions, le plus souvent à travers une interface spécifique dédiée à cet usage.

La première approche appelée *test intégré* par l'auteur est proposée dans [WAN00]. Un composant peut fonctionner selon cette approche en deux modes, *mode normal* et *mode d'entretien*. En mode normal, les possibilités de test intégré sont transparentes à l'utilisateur du composant et le composant est identique à un composant sans test intégré. Cependant, en mode d'entretien, l'utilisateur du composant peut tester le composant avec l'aide du test intégré. L'utilisateur du composant peut alors appeler des méthodes du composant, qui exécutent le test, évaluent de façon autonome ses résultats, et fournissent un rapport de test. Les cas de test et leurs implémentations sont fournis par le développeur du composant. Ils peuvent être contenus dans le composant ou générés à la demande. Avec ces cas de test, le composant peut se tester lui-même ou l'utilisateur peut exécuter les cas de test via l'interface de test.

L'inconvénient principal de cette approche concerne le stockage des cas de test ou d'une description de leur génération dans le composant. Ceci peut rendre le composant coûteux, notamment par ses demandes accrues de ressources. Par ailleurs, un tel test intégré a été envisagé par les auteurs uniquement lors du déploiement des composants. Le coût induit n'est donc pas amorti par une utilisation plus fréquente comme envisagé dans le diagnostic en ligne.

Les approches de test intégré sont également proposées dans le projet Component+ [HOR02][ATK02]. Les auteurs définissent une architecture consistant en trois types de composants : les composants BIT (Built-in test), les testeurs, et les gestionnaires. Les composants BIT sont les composants dotés de test intégré. Ces composants implémentent certaines interfaces obligatoires. Les testeurs sont les composants qui accèdent aux possibilités de test intégré des composants BIT par les interfaces correspondantes et qui contiennent les cas de test sous une certaine forme. Enfin, les gestionnaires sont des

composants qui ne contribuent pas au test, mais sont nécessaires, par exemple, pour assurer les mécanismes de recouvrement en cas de défaillances.

Dans ces approches de test intégré, les composants sont comme des boîtes noires et ils ont donc besoin de tests fonctionnels. De plus, les composants sont en général considérés comme des machines à états. Ils nécessitent donc des tests de transition d'état.

Les approches d'auto-test sont proposées dans [JEZ01][TRA99][DEV01][BAU01]. Dans ces travaux, on assume qu'un composant est implémenté en utilisant des langages orientés-objets, Java en particulier. L'autotest est implémenté par des méthodes additionnelles. Chaque méthode du composant pouvant être testée par les possibilités de test intégré possède une méthode de test comme contrepartie qui l'appelle avec des arguments prédéfinis. Le test est implémenté en utilisant les concepts d'*invariants*, de *pré* et *post conditions*. Invariants, pré et post conditions sont déterminés en fonction des spécifications du composant et sont embarqués par le fournisseur du composant dans le code source de celui-ci. La fonctionnalité nécessaire pour les valider et les tracer est implémentée par un framework, qui exige que le composant, ou plus précisément la classe principale du composant, implémente une certaine interface spécifique.

L'approche d'autotest est également proposée par Beydeda [BEY03][BEY04] sous l'appellation STECC (Self-Testing COTS Component). Dans cette approche, le développeur enrichit le composant avec la fonctionnalité spécifique du test. Ainsi, le composant peut se tester lui-même sans besoin d'exporter son code source. Cette approche utilise les tests basés sur le programme, c'est-à-dire des tests structurels. Ainsi, chaque composant encapsule un graphe de flot de contrôle qui modélise les informations du code source. Les cas de test sont générés et exécutés par le framework STECC. Le framework STECC implémente plusieurs algorithmes pour déterminer les chemins à traverser selon les critères spécifiques et pour générer les cas de test correspondants.

Les travaux sur le test des composants logiciels présentés ci-dessus ont pour principal objectif de tester un composant dans son nouvel environnement d'exécution lors du déploiement d'une application à base de composants. Dans le cas de nos travaux, le test de composants doit servir le processus de diagnostic en ligne. Notre intérêt se porte donc principalement sur une exécution en ligne, c'est-à-dire pendant l'exécution de l'application, du test des composants. En plus, les applications à base de composant ne disposent pas toujours le code des composants. Donc, nous avons basé sur le principe de l'approche de test intégré du projet Component+ pour la mise en œuvre de tests inter-composants en ligne. C'est-à-dire le test fonctionnel à partir de machine à états du composant.

Notre contribution consiste principalement à les étendre pour un contexte en ligne, c'est-à-dire en tenant compte des contraintes de concurrence du test et des fonctionnalités des composants, de partage de ressources entre le test et les fonctions des composants, ce qui rend nécessaire la mise en place de mécanismes qui perturbent le moins possible les fonctions des composants.

Comme indiqué précédemment, les approches de test intégré utilisent principalement une représentation sous forme de machine à états des composants. Le principe de la machine à états finis est étudié ci-dessous.

2.3.3. Machine à états finis

Comme nous l'avons présenté ci-dessus, les machines à états finis sont utilisées pour modéliser des comportements séquentiels. Une machine à états finis est un graphe, dont les

nœuds représentent les états du système, et les arcs représentent les transitions entre les états. Les transitions sont étiquetées avec l'entrée qui les provoque et les sorties qui sont générées lors du franchissement de la transition.

Les critères de sélection de cas de test pour les machines à états sont des critères de couverture de la structure du graphe [OFF03][AMM08] : passer par tous les états, par toutes les transitions [HUA75] ou par toutes les paires de transitions [PIM79]. Des versions plus sophistiquées du test de toutes les transitions ont été définies. Elles visent à l'exhaustivité vis-à-vis d'un modèle de faute simple, induisant des erreurs de transfert (la transition amène à un autre état que celui spécifié) ou de sortie (la transition ne génère pas l'événement attendu). Les travaux les plus connus sont : la méthode W [CHO78], la séquence de distinction et les séquences UIO (*Unique Input Ouput*) [SAB85][URA92]. Ces méthodes ont également été étudiées pour des machines indéterministes et des machines étendues avec des données [PHA90][CAV96].

Comme nous venons de le présenter, les machines à état sont utilisées pour modéliser des comportements séquentiels. Afin de couvrir des caractéritiques des systèmes complexes (comme la scalabilité, la simultanéité, *etc.*), *les statecharts* [HAR87] ont été développés. Des méthodes de génération des cas de test pour ce formalisme sont proposées dans [HON00], [OFF03]. En principe, il s'agit des critères de couverture des états et des transitions.

Dans la suite de nos travaux, nous utilisons les éléments de statecharts pour modéliser nos exemples. La Figure 17 montre un exemple simple de statechart UML.

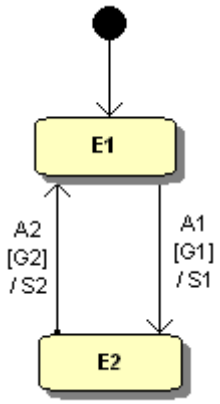


Figure 17. Un exemple de Statechart UML

Les deux transitions représentent les progressions des deux états E1 et E2. Les étiquettes sur les transitions sont de la forme *événements [garde]/sortie*. Par exemple, la transition de E1 en E2 est provoquée par l'événement A1 avec la garde G1 et donne la sortie S1.

Nous présentons ensuite en détail les méthodes de génération des cas de test que nous utilisons dans le cadre de nos travaux.

2.3.4. Génération des cas de test

Le test à base d'état se concentre sur la vérification du fonctionnement correct du composant. La conception des cas de test est basée sur les états et les transitions entre ces états. Binder [BIN00] présente une étude approfondie de la génération des cas de test à base d'états. Les principales stratégies de conception des cas de test sont décrites ci-dessous :

- "*Piecewise coverage*" : se concentre sur l'activation d'éléments distincts, par exemple la couverture de tous les états, de tous les événements, ou de toutes les actions. Ces techniques ne sont pas directement liées à la structure de la machine à états sous-jacente qui implémente le comportement. Il est possible de couvrir tous les états mais de manquer certains événements ou actions, ou produire toutes les actions sans visiter tous les états ou accepter tous les événements. Ceci entraîne parfois l'échec de la détection de fautes.
- "*Transition coverage*" : la couverture complète des transitions est assurée par une suite de tests si chaque transition spécifiée dans le modèle de l'état est exercée au moins une fois. En conséquence, ce critère couvre tous les états, tous les événements et toutes les actions.
- "*Round-trip path coverage*" : la couverture de chemin aller-retour est définie par la couverture d'au moins chaque séquence définie de certaines transitions qui commencent et se terminent dans le même état. Le plus court chemin aller-retour est une transition qui boucle sur le même état. Une suite de tests qui atteint la couverture complète de chemin aller-retour va révéler toutes les paires événement/action incorrectes ou manquantes.

Puisque les tests intégrés sont inclus dans un composant pendant l'exécution, leur taille et leur vitesse d'exécution sont des paramètres importants. Si un test n'est pas assez détaillé, il ne peut pas fournir le degré exigé de confiance dans l'exactitude du composant testé. En outre, si un test est trop détaillé, il peut augmenter la taille ou ralentir la vitesse d'exécution du test. Le test optimal représente un équilibre entre ces deux conditions, et dépend fortement du contexte dans lequel le test est appliqué. Par exemple, les tests en ligne dans un contexte embarqué doivent considérer des limitations de mémoire.

Dans notre travail sur le diagnostic en ligne de composant logiciel à partir de tests inter-composant, nous utilisons la deuxième stratégie de génération des cas de test, c'est-à-dire la couverture de toutes les transitions du modèle d'états. Nous ajoutons tous les cas de test dans le composant testé. Nous devons alors considérer la nature du système, les ressources disponibles pour le test en ligne ainsi que le degré de fiabilité requis pour le composant, lors du choix des cas de tests pour un composant donné.

Dans l'approche initiale de test intégré comme proposée par Wang *et al.* [WAN00], des cas de test complets sont mis à l'intérieur des composants et donc sont automatiquement réutilisés avec le composant. Bien qu'elle semble attrayante à première vue, cette stratégie n'est pas assez flexible pour convenir au cas général. Parce que le but de ce test intégré est de tester le composant dans un nouvel environnement, un composant a besoin de différents types de tests pour différents environnements et il n'est ni faisable ni flexible de les avoir tous intégrés de manière permanente. Pour résoudre ce problème, dans le cadre du projet Component+ [GRO02], les cas de test sont séparés de leurs composants et mis dans les composants testeurs. Une autre approche de test intégré a été proposée par Martins *et al.* [MAR01]. Elle consiste à mettre un nombre minimal de tests, comme des assertions, ainsi que des spécifications de test à l'intérieur des composants. Cependant, un logiciel spécifique doit être utilisé afin de transformer les spécifications de test en tests effectifs.

Dans notre travail, nous optons pour l'intégration des cas de tests à l'intérieur des composants testés. En effet, comme décrit plus haut, le but de nos tests est de servir le processus de diagnostic, qui nécessite des tests inter-composants. Ainsi, l'intégration des cas

de tests dans les composants testeurs serait coûteuse, car un composant peut tester plusieurs autres composants et un composant peut être testé par plusieurs autres composants.

2.4. Conclusion

Ce chapitre nous a permis d'étudier les principales briques de base nécessaires à nos travaux : les modèles de composants logiciels, la tolérance aux fautes pour les applications à base de composants logiciels et le test des composants logiciels. Les domaines du test des composants logiciels et de la tolérance aux fautes des applications à base de composants ont évolué séparément. Or, il a été démontré dans le cas des composants matériels que des solutions efficaces de tolérance aux fautes pouvaient être obtenues en utilisant le principe du test inter-composant en ligne. D'un point de vue conceptuel, les applications à base de composants logiciels sont similaires à des systèmes matériels construits par l'assemblage de composants de base. Ceci nous interpelle quant à l'exploration du concept de test inter-composant logiciel, en vue du diagnostic en ligne des applications logicielles. Les différents choix retenus et le développement de nos travaux sont détaillés dans le chapitre 3 pour ce qui concerne le test inter-composant et dans le chapitre 4 pour ce qui est relatif au diagnostic.

Chapitre 3

TESTS INTER-COMPOSANTS EN LIGNE

Le diagnostic en ligne est basé sur l'existence de tests mutuels entre les composants d'une application. La première contribution de nos travaux consiste principalement à étendre l'approche de test intégré pour un contexte en ligne, c'est-à-dire en tenant compte des contraintes de concurrence du test et des fonctionnalités des composants, et de partage des ressources entre le test et les fonctions des composants. Ce chapitre présente la solution proposée de test intégré en ligne de composant logiciels. La mise en œuvre de cette solution est illustrée à travers deux cas d'étude. Ces travaux ont donné lieu à deux publications Quatita'07 [BUI07a] ET ERCS'07[BUI07b].

3.1. Hypothèses de travail

3.1.1. Hypothèses sur les fautes considérées

Comme nous l'avons vu dans le chapitre précédent, le diagnostic en ligne nécessite des hypothèses sur la nature des fautes considérées et la connectivité du système sous test.

Dans le cas des applications logicielles, tout composant peut *a priori* être affecté au test de tout autre composant de l'application. Il est alors possible de considérer qu'une application logicielle est un système complètement connecté.

Concernant les fautes, peu de travaux ont, à notre connaissance, étudié les modèles de fautes adaptés au test en ligne de composants logiciels. Or, certains auteurs [GEF98][ARL00] s'accordent pour dire que la validation d'un composant logiciel avant son intégration dans une application ne suffit pas forcément à en assurer un fonctionnement correct. En particulier, un composant peut être mal employé, ou subir par propagation des erreurs dues à des fautes dans le matériel sous-jacent ou encore comporter lui-même des défauts résiduels de conception ou de réalisation [MEY79][MIL99].

Comme mentionné dans le chapitre 2, nous utilisons dans nos travaux des tests fonctionnels. Nous nous limitons donc à considérer la manifestation des fautes éventuelles au niveau du comportement des composants sous test et faisons l'hypothèse que ces fautes sont *permanentes*. Par conséquent, dès qu'elles se produisent dans un composant, elles se manifestent toujours jusqu'à leur correction.

3.1.2. Hypothèse sur les composants

Après l'étude des approches de test inter-composants existantes (section 2.3.2), l'approche de test intégré du projet Component+ [HOR02][ATK02] nous a semblé être une bonne base de départ pour le test en-ligne des composants logiciels. En effet, cette approche, développée pour le déploiement des composants, se fonde sur le test fonctionnel à partir de modèles d'états du composant. L'hypothèse est donc faite que :

- soit les modèles d'états sont livrés avec les composants par leurs concepteurs,
- soit les modèles d'états dont construits à partir des spécifications des composants.

Dans cette thèse, un composant qui intègre une interface de test est appelé *composant testable*.

Pour que notre proposition soit indépendante le plus possible des modèles de composants, les seuls types d'interfaces considérés nécessaires pour notre travail sont les interfaces requises et les interfaces fournies. Ces interfaces sont généralement disponibles dans la plupart des modèles de composants.

La contribution de nos travaux consiste principalement à étendre l'approche de test intégré pour un contexte en ligne, c'est-à-dire fournir des mécanismes pour gérer les interférences entre les tests et les fonctionnalités normales du composant pendant l'exécution du système. Dans les parties suivantes, nous présentons le principe de l'approche de test intégré du projet Component+, la mise en œuvre dans le contexte en ligne, et deux exemples de cas d'études.

3.2. Interface de test intégré

Comme indiqué dans le chapitre 2, l'idée principale du test intégré [HOR02][ATK02] est d'intégrer une interface de test dans le composant qui fournit des fonctionnalités de test. Cette interface de test constitue alors le point d'entrée pour tout composant souhaitant tester un autre composant.

Comme présenté dans le Chapitre 2, un composant comprend généralement des interfaces fournies et des interfaces requises lui permettant de communiquer avec son environnement. Chaque interface fournie est un ensemble d'opérations que le composant fournit, tandis que chaque interface requise est un ensemble d'opérations nécessaires au composant pour effectuer ses opérations. De la même manière, nous considérons que le test est juste un autre service que le composant fournit à son environnement. Ainsi, des fonctions de test sont fournies par une interface dédiée qui est *l'interface de test intégré* (Figure 18).

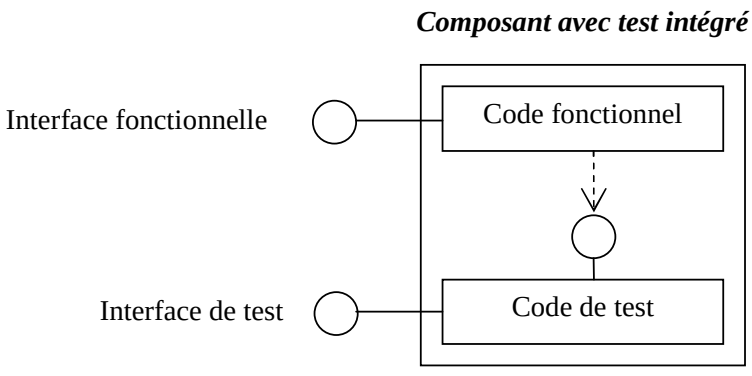


Figure 18. Composant avec test intégré

Un composant peut en général être considéré comme une machine à états. Chaque fois qu'un client appelle une opération d'un composant, l'état du composant change. En outre, quand un composant exécute une fonction donnée, son état change aussi. En général, un sous-ensemble des états d'un composant est significatif pour un intégrateur. Ainsi, pour le processus de test, certains états ont un intérêt primordial et doivent être définis. Il est par exemple important de connaître l'état correspondant à une exécution correcte d'une opération. De tels états sont définis en analysant le comportement du composant à travers ses spécifications.

La machine à états d'un composant logiciel fournit un support pour le test de transition d'état. Par conséquent, avant l'exécution d'un test, le composant à tester doit être amené à l'état initial requis pour ce test. Après l'exécution d'un test, le testeur doit vérifier que la sortie (si produite) est celle attendue et que le composant testé se trouve bien dans l'état final attendu.

Par définition, cependant, les états internes d'un composant sont invisibles aux entités externes. Par conséquent, le test ne peut pas effectuer ou vérifier les états internes exceptés à travers l'interface fonctionnelle normale du composant. Une séquence spécifique des invocations d'opérations à travers l'interface fonctionnelle normale semble donc nécessaire pour amener le composant dans un état requis pour l'exécution d'un test. Cependant, puisque les tests sont réalisés pour vérifier que l'interface fonctionnelle se comporte comme prévu, il est imprudent d'utiliser l'interface fonctionnelle pour affecter et vérifier les états internes d'un composant, et vérifier les résultats de test. En d'autres termes, nous ne devrions pas utiliser interface fonctionnelle pour réaliser un test sachant que c'est réellement le sujet de ce test. Ce problème est résolu en introduisant des opérations spéciales dans l'interface de test pour affecter et obtenir l'état interne d'un composant (Figure 19).

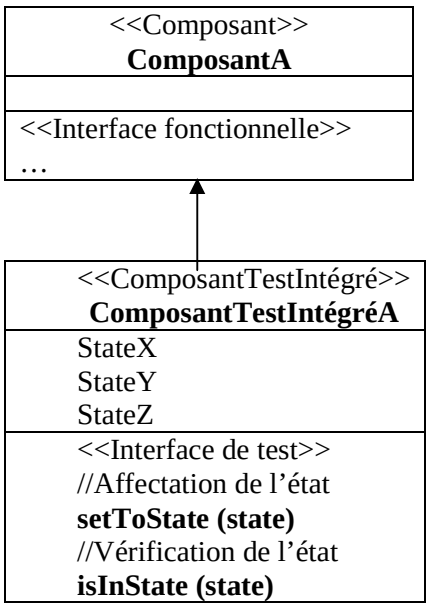


Figure 19. Description d'un composant avec test intégré et interface de test

Ces opérations sont `setToState(state)` et `isInState(state)`. L'opération `isInState(state)` vérifie si le composant réside dans un état logique passé en paramètre. L'opération `setToState(state)` affecte les attributs internes du composant pour représenter un état distinct logique.

L'interface de test est exécutée comme une extension autonome du composant, pour que l'implémentation du programme de test soit encapsulée et clairement séparée de la fonction normale. L'interface de test est l'une des interfaces du composant, le composant testeur peut utiliser cette interface comme les autres interfaces fonctionnelles.

3.3. Contexte en ligne

Dans le cas du diagnostic en ligne, les processus fonctionnels et les processus de test s'exécutent en parallèle. Un problème important consiste à gérer les interférences entre les tests et les fonctionnalités normales du composant. En effet l'exécution des tests peut interférer avec les fonctionnalités normales du composant. Que doit faire le composant si pendant le processus de test, il reçoit une demande de service fonctionnel d'autres composants et réciproquement ? Que doit faire le composant si le test manipule et change des données fonctionnelles du composant ? Concernant les données fonctionnelles du composant sous test, des mécanismes de retour arrière, garantissant que les données modifiées pendant le test retrouvent leur état d'origine après le test doivent être fournis. En particulier, si les opérations de test modifient des données persistantes de l'application, ces données doivent être restaurées aux valeurs précédentes dès la fin du test.

Concernant la gestion des interférences entre le test et les fonctionnalités du composant, une idée de base, inspirée du test des composants matériels, consiste à détecter les périodes d'oisiveté d'un composant. Ainsi, un composant oisif serait candidat à un test inter-composant, que ce soit dans le rôle du composant testeur ou dans celui du composant testé. Cependant, en pratique il n'est pas évident de détecter les composants oisifs. En effet, une telle détection serait liée étroitement au système d'exploitation sous-jacent, ce qui réduirait la portabilité du test inter-composant. D'autres approches sont mentionnées dans la littérature. Dans [SUL06], quelques approches sont présentées, mais elles n'ont pas fait l'objet d'une implémentation concrète à notre connaissance. Ces approches sont :

- **Blocage du composant** : le composant est bloqué pendant le processus de test. Au cours de l'exécution d'un test, les demandes de test ou de l'application sont retardées jusqu'à la fin du processus de test. Les données persistantes du composant doivent être restaurées à l'état original après la fin du processus de test.
- **Abandon du processus de test** : le composant abandonne le processus de test. Quand un autre composant demande la fonctionnalité normale du composant testé, le processus de test est abandonné. Une fois abandonné, les données du composant doivent être restaurées.
- **Clonage du composant** : le composant est cloné par l'infrastructure d'exécution avant le démarrage du test. Le processus de test est alors exécuté sur le clone, et la fonction normale est exécutée sur le composant original. Avec cette solution, le fonctionnement normal de l'application n'est pas perturbé. Cependant, cette option peut être très coûteuse en terme de consommation de ressources.
- **Session de test** : l'interface de test du composant fournit des méthodes qui permettent des sessions de test. Ces méthodes garantissent que les données de test et les données réelles ne sont pas mélangées durant le processus de test. Cela pourrait aussi être fait par le clonage du composant (conduit par le composant lui-même) ou par des opérations spécifiques. Avec cette solution, le fonctionnement

normal de l'application n'est pas perturbé mais une charge supplémentaire pour le développeur de composants ou de l'application est inévitable.

Il est important de noter qu'aucune solution n'est optimale et que le choix d'une solution plutôt qu'une autre dépend du contexte considéré. L'analyse du principe de ces différentes stratégies nous amène toutefois à considérer les cas suivants :

- Quand les applications considérées sont temps réel ou critiques, les demandes de test sont abandonnées au profit des appels fonctionnels. Il est toutefois nécessaire de garantir que tout composant est testé au moins une fois dans une période donnée, sinon l'apparition de fautes peut passer inaperçue et nuire au fonctionnement de l'application.
- Quand les applications ne sont pas critiques, il est envisageable d'interrompre momentanément un composant en vue de le tester. Pour ce type d'applications, nous optons pour la démarche suivante :
  - Quand les fonctions du composant sous test sont de type lecture seule, l'opération est bloquée jusqu'à la fin du test.
  - Quand les opérations sous test modifient des données, on créé des sessions de test. Ainsi le test n'agit pas sur les données réelles du composant.

3.4. La mise en œuvre des tests en ligne

Un point délicat pour la mise en œuvre d'un test inter-composant en ligne comme décrit précédemment consiste en l'obtention du modèle d'états du composant.

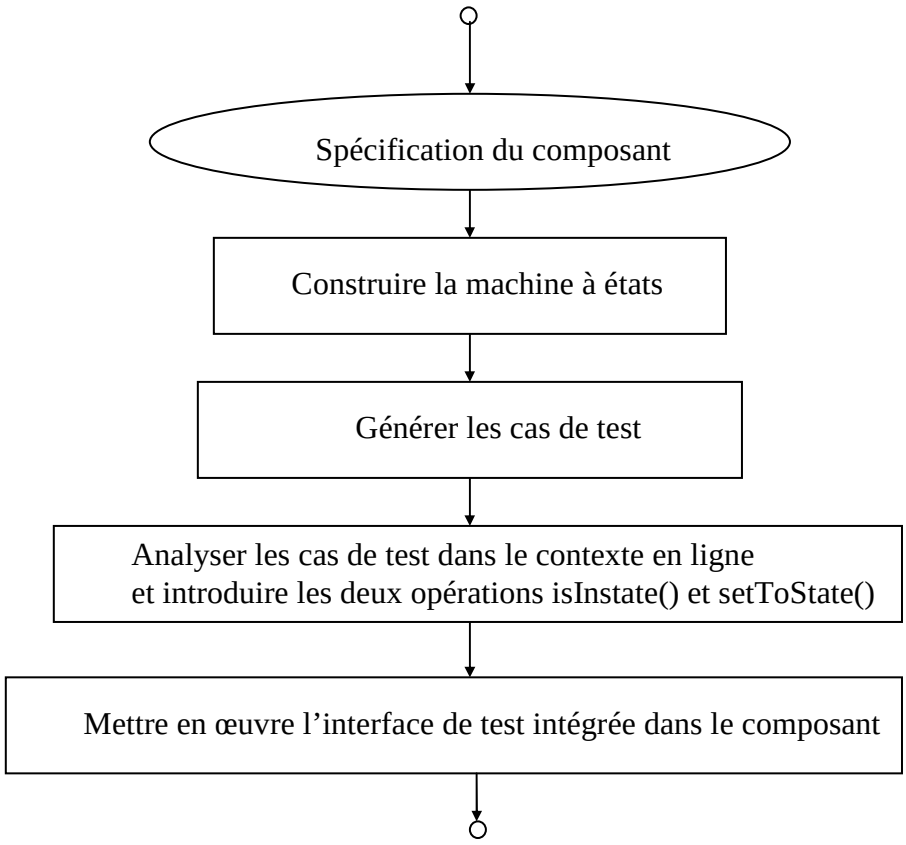


Figure 20. La mise en œuvre des tests en ligne

En général, le modèle d'états d'un composant est construit à partir du modèle de comportement (spécification) du composant ou bien, il est fourni par le concepteur du composant. Cette première étape ne constitue donc pas un point de blocage à la mise en œuvre du test en ligne (Figure 20). L'étape suivante est de générer des cas de test avec l'aide du modèle d'états selon le critère de couverture de toutes les transitions du modèle d'états (ce choix est justifié dans la section 2.3.4 du Chapitre 2). En analysant les opérations testées dans le contexte en ligne, on met en œuvre les deux opérations dédiées au test `isInState()` et `setToState()` et les cas de test. Les deux opérations précédentes ainsi que les cas de test sont placés dans l'interface de test du composant testé. L'intégration des cas de tests dans les composants testeurs serait coûteuse, car un composant peut tester plusieurs autres composants et un composant peut être testé par plusieurs autres composants. (la motivation de l'intégration des cas de test dans le composant testé est présentée dans la section 2.3.4 du Chapitre 2). Cette phase peut être automatisée comme le montrent les travaux de [OFF99][OFF03] et [SEI08] sur les modèles d'états UML.

Par conséquent, outre les deux fonctions `setToState(State)` et `isInState(State)`, l'interface de test exporte aussi une opération `testComponent(int[] testcases)` qui comprend tous les cas de test pour que les autres composants puissent tester ce composant. Le tableau `testCases` passé en paramètre contient les cas de test que le composant testeur peut utiliser pour tester ce composant. L'interface de test se présente donc comme suit :

```
<<interface>>
TestInterface
isInState (State state)
setToState(State state)
testComponent(int[] testcase)
{
   testcase1
   testcase2
…
}
```

Dans la suite de ce chapitre, nous montrons la mise en œuvre des tests en ligne telle que proposée ci-dessus, à travers deux exemples décrits avec le modèle de composant CCM.

3.5. Exemples de mise en œuvre des tests en ligne

Dans cette partie, nous présentons la mise en œuvre des interfaces de test intégré dans les composants dans le contexte en ligne pour deux cas d'étude assez simples : un exemple de système de carte bancaire et un exemple de système de climatisation.

3.5.1. Système de carte bancaire

3.5.1.1. Description du système de carte bancaire

Le système de carte bancaire comprend cinq composants : le composant client, le composant de validation du code Pin, le composant de dépôt, le composant de retrait et le composant de consultation.

- Composant *CodeServer* : la mission principale de ce composant est de vérifier le code fourni par l'utilisateur. La carte sera bloquée après trois essais fournissant un

code pin incorrect et elle sera débloquée par le banquier en entrant un code de sécurité.

- Composant *ConsultServer* : se comporte comme une base de données. Il contient toutes les informations sur le client.
- Composant *WithdrawServer* : traite la commande de retrait d'espèces du client. La limite du retrait est de 300 euros, et le retrait doit être inférieur au montant disponible dans le compte.
- Composant *DepositServer* : traite la commande de dépôt d'espèces. Il n'y a pas de limite dans ce cas.
- Composant *Client* : le composant client comprend les interfaces graphiques pour permettre l'utilisation du système. Il fait entrer le code et choisir sa demande à l'utilisateur.

Nous avons implémenté ce cas d'étude en utilisant le modèle de composants CORBA (CCM). Les connexions entre les composants de ce système sont illustrées dans la Figure 21. La description détaillée de ces composants est donnée en annexe B.

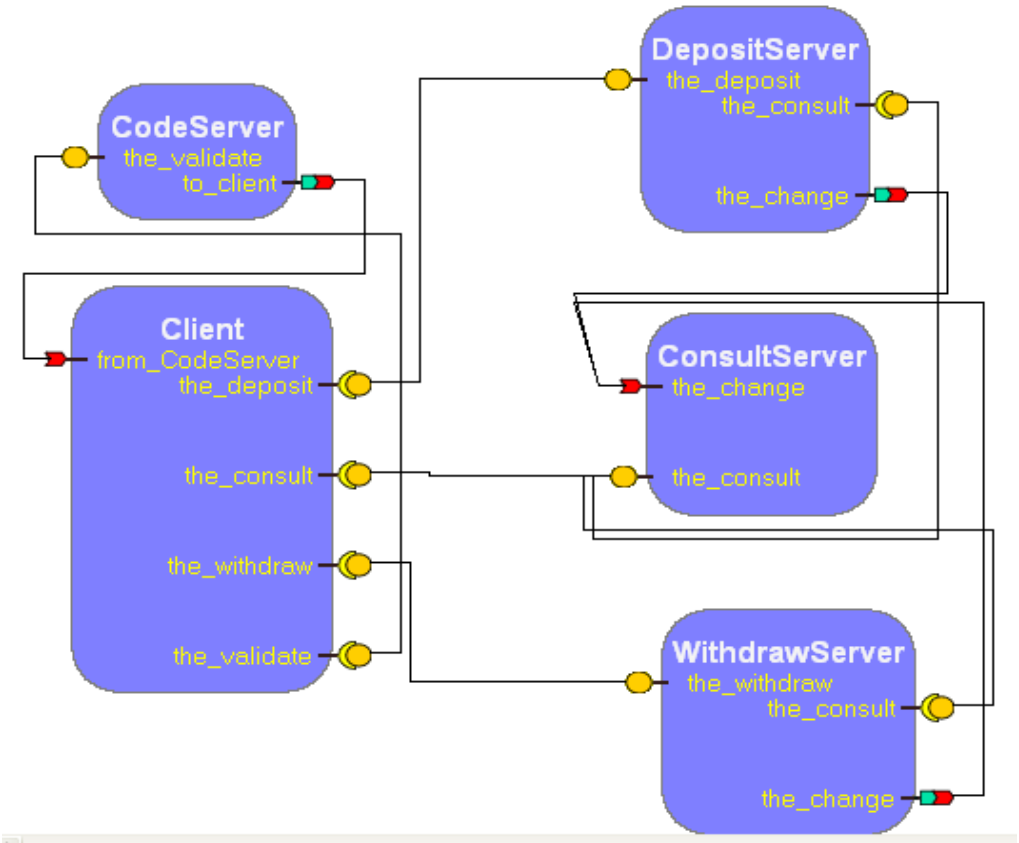


Figure 21. Les connexions entre les composants de l'exemple du système de carte bancaire

Les composants CodeServer, DepositServer, WithdrawServer, ConsultServer fournissent les services de vérification de code pin, dépôt d'argent, de retrait d'argent, et de consultation de compte, respectivement. Les composants DepositServer et WithdrawServer peuvent utiliser le service de consultation pour connaître le solde actuel du client. Le client est une interface graphique qui utilise les services de tous les autres composants.

3.5.1.2. Interfaces de test intégré dans les composants

Parmi les 5 composants du système de carte bancaire, le composant client est une interface graphique, les 4 autres composants fournissent des services. Nous procédons donc au test de ces 4 composants uniquement.

3.5.1.2.1. Interface de test du composant validation de code

La carte sera verrouillée après un nombre maximal d'essais fournissant un faux code pin. Typiquement, la carte sera verrouillée après 3 essais non réussis. Ce compte est toujours remis à zéro (état *cleared*) si le code correct a été fourni. Ceci est représenté dans le modèle d'état de la Figure 22. L'état "*cleared*" indique l'essai réussi, et l'état "*locked*" indique que la carte bancaire sera retenue par le banquier.

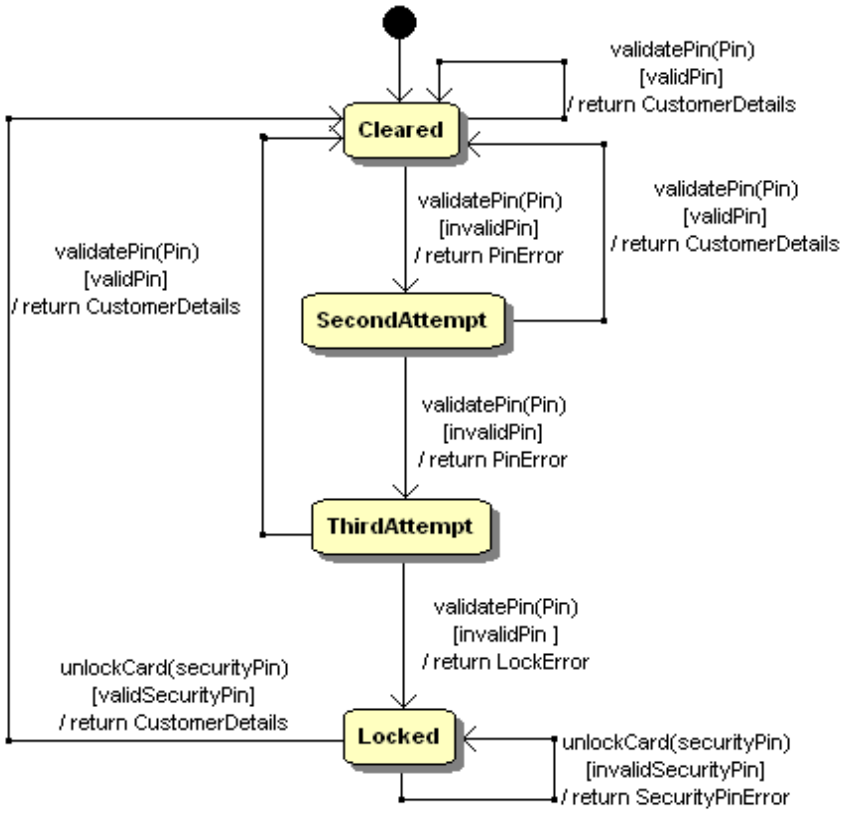


Figure 22. Modèle d'états du composant de validation de code

Les cas de test du composant de validation de code pin qui correspondent à toutes les transactions possibles du modèle d'états du composant sont décrits dans le Tableau 3.

Tableau 3. Les cas de tests du composant de validation de code pin

N° Cas de test	Entrées	Etat initial	Méthode à tester	Etat final	Sorties
1	Code pin correct	Cleared	validatePin(Pin)	Cleared	Information du client
2	Code pin faux	Cleared	validatePin(Pin)	SecondAttempt	Code pin erreur
3	Code pin correct	SecondAttempt	validatePin(Pin)	Cleared	Information de client
4	Code pin faux	SecondAttempt	validatePin(Pin)	ThirdAttempt	Code pin erreur
5	Code pin correct	ThirdAttempt	validatePin(Pin)	Cleared	Information de client
6	Code pin faux	ThirdAttempt	validatePin(Pin)	Locked	Bloque erreur
7	Code de sécurité correct	Locked	unlockCard (securityPin)	Cleared	Information de client
8	Code de sécurité faux	Locked	unlockCard (securityPin)	Locked	Code de sécurité erreur

Le composant de validation de code passe par quatre états différents qui sont liés au compteur du nombre d'essais non réussis pour fournir le code pin correct de la carte. Ainsi, l'implémentation des opérations d'affectation et de vérification d'état est simple. La réalisation ces deux opérations `isInState(State)` et `setToState(State)` est représentée sur la Figure 23.

setToState

```
case: State in
Cleared:
unsuccessfulAttempts = 0
SecondAttempt:
unsuccessfulAttempts = 1
ThirdAttempt:
unsuccessfulAttempts = 2
Locked:
unsuccessfulAttempts = 3
default: continue
```

isInState

```
case: State in
Cleared: return
(unsuccessfulAttempts = = 0)
SecondAttempt: return
(unsuccessfulAttempts = = 1)
ThirdAttempt: return
(unsuccessfulAttempts = = 2)
Locked: return
(unsuccessfulAttempts = = 3)
default: return false
```

Figure 23. Implémentation de deux opérations isInState et setToState

3.5.1.2.2. Interface de test du composant ConsultServer

Ce composant fournit une méthode qui retourne le solde actuel du client :

```
long consultAccount(accountNumber);
```

Ce composant n'a qu'un état et le test de ce composant est très simple. Il n'a que deux cas de test : un test avec un bon numéro de compte et un test avec un mauvais numéro de compte.

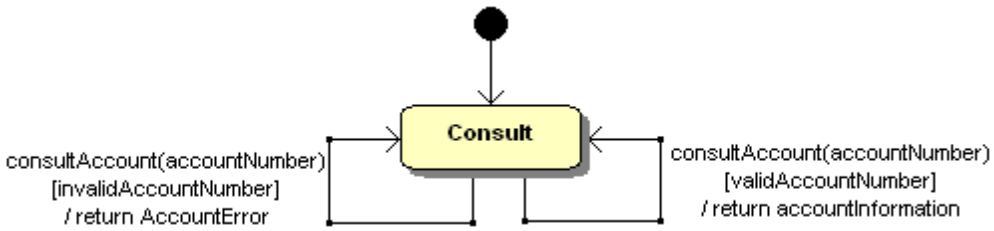


Figure 24. Modèle d'états du composant ConsultServer

Le modèle d'états de ce composant est illustré sur la Figure 24 et les cas de test sont résumés dans le Tableau 4.

Tableau 4. Les cas de test du composant ConsultServer

N° Cas de test	Entrées	Méthode à tester	Sorties
1	Numéro de compte correct	consultAccount(accountNumber)	AccountInfomation
2	Numéro de compte faux	consultAccount(accountNumber)	AccountError

3.5.1.2.3. Interface de test du composant DepositServer

Ce composant traite la commande de dépôt d'argent. Il fournit une méthode pour ajouter un montant sur le compte du client :

```
boolean depositMoney(accountNumber, amount) ;
```

Tout comme le composant ConsultServer, le composant DepositServer n'a qu'un seul état et le test de ce composant est très simple. Il y a deux cas de test : le succès du dépôt ou l'échec du dépôt.

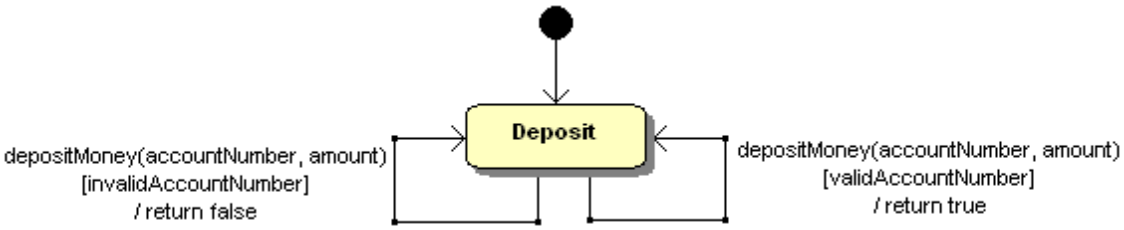


Figure 25. Modèle d'états du composant DepositServer

Le modèle d'états de ce composant est illustré par la Figure 25 et les cas de test sont résumés dans le Tableau 5.

Tableau 5. Les cas de test du composant DepositServer

N° Cas de test	Entrées	Méthode à tester	Sorties
1	Numéro de compte correct	depositMoney(accountNumber, amount)	True
2	Numéro de compte faux	depositMoney(accountNumber, amount)	False

3.5.1.2.4. Interface de test du composant WithdrawServer

Ce composant traite la commande de retrait d'argent. Il fournit une méthode pour retirer l'argent du compte du client :

```
boolean withdrawMoney(accountNumber, amount);
```

Cette méthode vérifie si le montant à retirer est inférieur à 300 euros et à la somme disponible dans le compte. Le composant a un seul état et quatre cas de test :

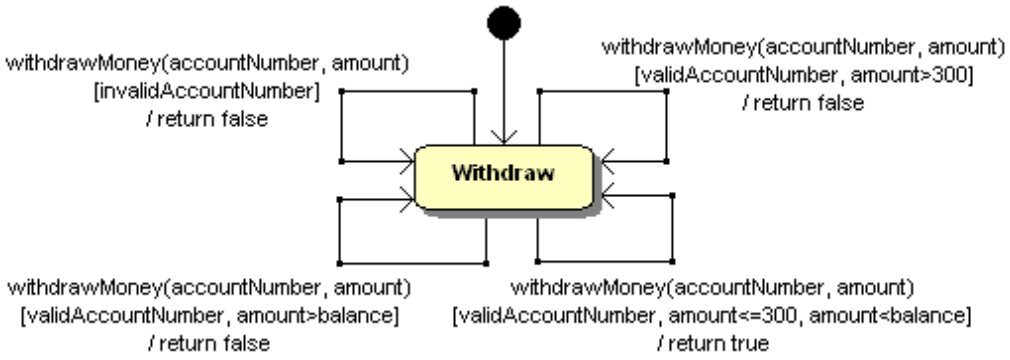


Figure 26. Modèle d'états du composant WithdrawServer

Le modèle d'états de ce composant est illustré par la Figure 26 et les cas de test sont résumés dans le Tableau 6.

Tableau 6. Les cas de test du composant WithdrawServer

N° Cas de test	Entrées	Méthode à tester	Sorties
1	Numéro de compte faux	withdrawMoney(accountNumber, amount)	False
2	Numéro de compte correct Montant <= 300 et Montant < Solde	withdrawMoney(accountNumber, amount)	True
3	Numéro de compte correct Montant > 300	withdrawMoney(accountNumber, amount)	False
4	Numéro de compte correct Montant > Solde	withdrawMoney(accountNumber, amount)	False

3.5.1.3. Contexte en ligne

Dans le cas du système de carte bancaire, nous pensons que la solution de blocage du composant et la solution de session de test sont probablement les plus adaptées comme décrit ci-après.

- Pour les cas de test avec la méthode `validate(codePin)` du composant CodeServer, on a opté pour la solution de blocage du composant. Ainsi, pendant le processus de test, si la commande de validation du code du client est demandée, elle doit attendre jusqu'à la fin du processus de test. Comme le temps de test est normalement faible, le fonctionnement normal du système est donc peu perturbé. Avant le test, nous sauvegardons l'état actuel du composant, et à la fin du test, nous restaurons le composant à l'état original. Par exemple, avant le test, le composant est dans l'état "*cleared*", si nous testons le cas de test 6 (Tableau 3), l'état final de ce cas de test est "*locked*", il faut donc sauvegarder et restaurer le composant à l'état original avant et après chaque cas de test respectivement (Figure 27).

```
//Cas de test 6: tester le troisième essai de fournir un faux code pin
// état actuel du composant est 'cleared', c'est-à-dire unsuccessfulAttempts = = 0
TestCase6()
{
// sauvegarder l'état actuel
unsuccessfulAttemptsSave = unsuccessfulAttempts
// Amener le composant avec test intégré à un état spécifique avant le test
setToState ("ThirdAttempt");
// Exécution de l'opération à tester "validatePin(pin)"
validatePin(pin)
//Vérifier le résultat de cette opération et l'état final du composant
 if (validPin == false)
      if (isInstate ("locked"))
      testResult="OK"
      else
      testResult="FALSE"
else
testResult="FALSE"
// Restaurer à l'état initial
unsuccessfulAttempts = unsuccessfulAttemptsSave
return testResult
}
```

Figure 27. Exemple d'un cas de test tenant compte de la restauration des données du composant

- Pour les autres cas de test des composants DepositServer, WithdrawServer, on a opté pour la solution de session de test. Il est plus simple de créer un compte pour le test, les opérations de retrait et de dépôt sont testées sur le compte de test. Il n'y a donc pas d'interférence avec les données fonctionnelles de ces composants.
- Pour les cas de test du composant ConsultServer, il n'y a aucun impact sur le fonctionnement du composant, car ils retournent les informations de compte du client.

3.5.2. Système de climatisation

3.5.2.1. Description du système de climatisation

Le système de climatisation consiste en :

- Des thermomètres : rapportent la température à distance à divers endroits.
- Des thermostats : fonctionnent comme des thermomètres mais permettent en plus de sélectionner une température nominale.
- Une station de contrôle : permet à un opérateur de surveiller, de gérer les thermomètres et les thermostats, et de changer les températures à distance.
- Moteur de température : contrôle et dirige la valve d'air chaud et la valve d'air froid pour ajuster la température fournie à la salle.

La réalisation de ce système de climatisation utilise 7 composants :

Thermomètre :

- Composant **Thermometer** : qui détient l'identifiant unique du thermomètre, son modèle et sa localisation. La température de l'environnement est mesurée par un capteur et ce composant la fournit au composant administrateur par l'interface *GetEnviTempThermometer*.

Thermostat :

- Composant **Sensor** : qui fournit la température de l'environnement à l'utilisateur ou l'administrateur par l'interface *GetEnviTempThermostat*.
- Composant **User** : qui fournit l'interface graphique d'interaction entre l'utilisateur et le thermostat. Il permet à l'utilisateur de spécifier la température désirée, d'arrêter le thermostat, de connaître la température actuelle et l'état du thermostat, *etc.*
- Composant **Thermostat** : qui met la salle à la température demandée par l'utilisateur au moyen de l'interface *SetUserTemp*. La température demandée doit être supérieure et inférieure à la valeur minimale et maximale respectivement. Si la température demandée est supérieure à la température environnante, l'air chaud sera diffusé dans la salle. Si la température demandée est inférieure à la température environnante, l'air froid sera diffusé dans la salle. En outre, ce composant permet à l'administrateur d'établir les valeurs minimale et maximale spécifiques à ce thermostat via l'interface *SetMaxMinTemp*. Enfin, ce composant fournit l'interface *UserStop* à l'utilisateur pour lui permettre d'arrêter le thermostat.

Station centrale de contrôle et moteur de température :

- Composant **Administrator** : fournit l'interface graphique d'interaction entre l'administrateur et le système. Il permet à l'administrateur de surveiller, de contrôler le système de climatisation, de modifier et de connaître la température dans les salles à distance, d'arrêter les thermostats et les thermomètres, de consulter la localisation et le type des stations, d'ajouter, de modifier, de supprimer des stations, *etc.*
- Composant **Gestion** : se présente sous forme d'une base de données, qui stocke toutes les informations des stations (le nom de la station, le type de la station qui est thermomètre ou thermostat, la localisation de la station). Ce composant fournit les interfaces *Consult, Add, Modify, Delete* qui permettent à l'administrateur de consulter, d'ajouter, de modifier, et de supprimer des stations.
- Composant **EngineTemp** : qui contrôle et dirige la valve d'air chaud et la valve d'air froid pour ajuster la température fournie à la salle.

Nous avons implémenté ce cas d'étude en utilisant le modèle de composants CORBA. Les connexions entre les composants du système de climatisation sont illustrées dans la Figure 28. La description détaillée de ces composants est donnée en annexe C.

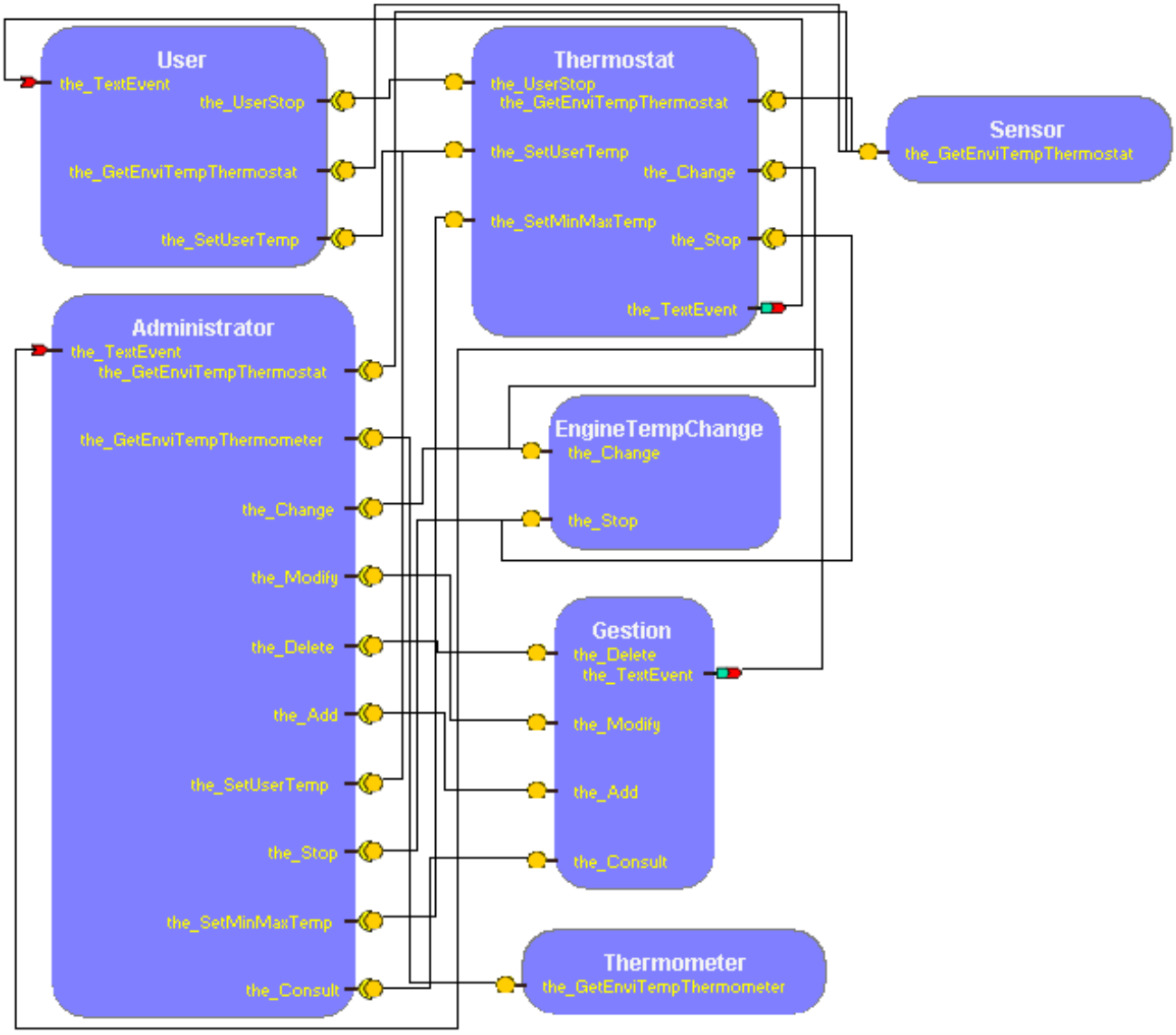


Figure 28. Connexions entre les composants du système de climatisation

3.5.2.2. Interfaces de test

Parmi les composants de ce système, cinq composants fournissent des services, les deux autres composants (User et Administrator) sont des interfaces graphiques qui utilisent les services des autres composants. Nous avons donc intégré les interfaces de test dans les 5 composants qui sont thermostat, thermomètre, capteur, gestion, et moteur de température pour les rendre testables.

3.5.2.2.1. Interface de test du composant Thermostat

La mission du thermostat est de garder la température dans la salle à la valeur spécifiée par l'utilisateur. L'attribut qui détermine ce comportement est la différence entre la température désirée de l'utilisateur et la température ambiante, un attribut interne de ce composant. Ce composant a 4 états : "*Heat*", "*Cool*", "*Inactive*" et "*Stop*" comme indiqué dans la Figure 29.

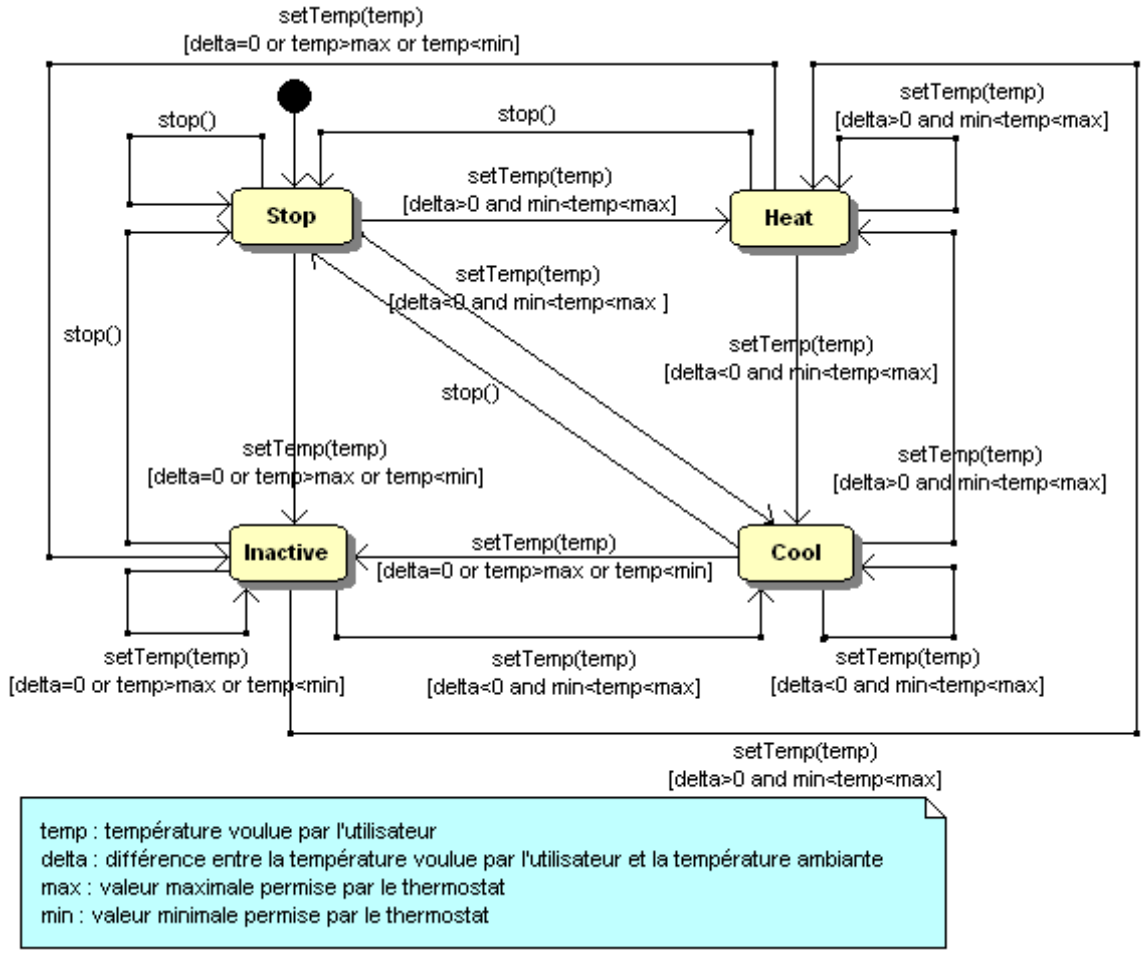


Figure 29. Modèle d'états du composant Thermostat

- L'état "*Heat*" indique que le thermostat diffuse de l'air chaud et que la température désirée par l'utilisateur est supérieure à la température ambiante.
- Quand la température voulue par l'utilisateur est inférieure à la température ambiante, le composant est dans l'état "*Cool*". Ceci indique que le thermostat diffuse de l'air froid.
- L'état "*Inactive*" indique que la machine est en service mais elle ne diffuse pas d'air.
- Et l'état "*Stop*" indique que le thermostat est éteint.

Les cas de test de ce composant sont décrits dans le Tableau 7.

Tableau 7. Les cas de test du composant Thermostat

N° Cas de test	Entrées	Etat initial	Méthode à tester	Etat final
1		Stop	stop()	Stop
2		Heat	stop()	Stop
3		Cool	stop()	Stop
4		Inactive	stop()	Stop
5	delta>0 and min<temp<max	Stop	setTemp(temp)	Heat
6	delta=0 or min>temp or temp>max	Stop	setTemp(temp)	Inactive
7	delta<0 and min<temp<max	Stop	setTemp(temp)	Cool
8	delta>0 and min<temp<max	Heat	setTemp(temp)	Heat
9	delta=0 or min>temp or temp>max	Heat	setTemp(temp)	Inactive
10	delta<0 and min<temp<max	Heat	setTemp(temp)	Cool
11	delta>0 and min<temp<max	Cool	setTemp(temp)	Heat
12	delta=0 or min>temp or temp>max	Cool	setTemp(temp)	Inactive
13	delta<0 and min<temp<max	Cool	setTemp(temp)	Cool
14	delta>0 and min<temp<max	Inactive	setTemp(temp)	Heat
15	delta=0 or min>temp or temp>max	Inactive	setTemp(temp)	Inactive
16	delta<0 and min<temp<max	Inactive	setTemp(temp)	Cool

La réalisation des opérations `isInState(State)` et `setToState(State)` du composant Thermostat est montrée sur la Figure 30.

setToState

```
Case : State in
Stop :
isStop = true
Inactive :
isStop = false and
(reqTemp = enviTemp or
reqTemp < min or reqTemp > max)
Heat :
isStop = false and
reqTemp > enviTemp and
min< reqTemp < max
Cool :
isStop = false and
reqTemp < enviTemp and
min< reqTemp < max
default : continue
```

isInState

```
Case : State in
Stop : return
isStop = = true
Inactive : return
isStop = = false and
((reqTemp-enviTemp) = = 0 or
reqTemp < min or reqTemp >max)
Heat : return
isStop = = false and
(reqTemp-enviTemp) > 0 and
min<reqTemp<max
Cool : return
isStop = = false and
(reqTemp-enviTemp) < 0 and
min < reqTemp <max
default : return false
```

Figure 30. Implémentation des opérations isInState et setToState du composant Thermostat

3.5.2.2.2. Interface de test du composant Gestion

Le composant Gestion est comme une base de données qui stocke et gère toutes les informations des thermostats et thermomètres sous la forme d'un tableau. Ce tableau permet de consulter, ajouter, modifier, et supprimer les informations (nom, type, localisation) des thermostats et des thermomètres. Le modèle d'états du composant Gestion est décrit dans la Figure 31.

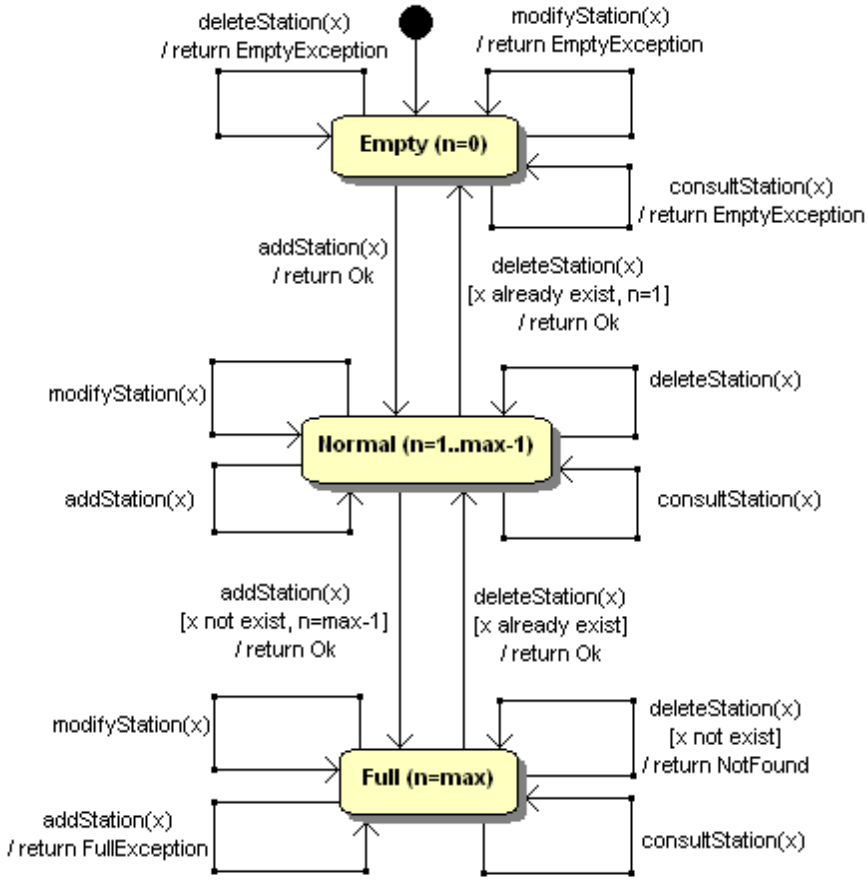


Figure 31. Modèle d'états du composant Gestion

Ce composant a trois états :

- L'état "*Empty*" indique que le système de climatisation n'a aucune station, ni thermomètre, ni thermostat. Dans cet état, l'administrateur ne peut qu'ajouter des stations, il ne peut pas modifier ou supprimer des stations.
- Le composant est en état "*Normal*" si le système de climatisation a au moins une station et au plus (max -1) stations où max est le nombre maximum de station dans le système.
- L'état "*Full*" indique que le système a un nombre de stations maximal (max). On ne peut pas ajouter de station.

A partir du modèle d'états de ce composant, nous déduisons les cas de test décrits dans le Tableau 8 :

Tableau 8. Les cas de test du composant Gestion

N° Cas de test	Entrées	Etat initial	Méthode à tester	Etat final	Sorties
1	x	Empty	deleteStation(x)	Empty	EmptyException
2	x	Empty	modifyStation(x)	Empty	EmptyException
3	x	Empty	consultStation(x)	Empty	EmptyException
4	x	Empty	addStation(x)	Normal	Ok
5	x n'existe pas	Normal	deleteStation(x)	Normal	NotFound
6	x existe, n=1	Normal	deleteStation (x)	Empty	Ok
7	x existe, n>1	Normal	deleteStation (x)	Normal	Ok
8	x n'existe pas	Normal	consultStation(x)	Normal	NotFound
9	x existe	Normal	consultStation(x)	Normal	Détail de x
10	x n'existe pas	Normal	modifyStation(x)	Normal	NotFound
11	x existe	Normal	modifyStation(x)	Normal	Ok
12	x n'existe pas n<(max-1)	Normal	addStation(x)	Normal	Ok
13	x n'existe pas n=max-1	Normal	addStation(x)	Full	Ok
14	x existe	Normal	addStation(x)	Normal	Déja existe
15	x n'existe pas	Full	modifyStation(x)	Full	NotFound
16	x existe	Full	modifyStation(x)	Full	Ok
17	x	Full	addStation(x)	Full	FullException
18	x n'existe pas	Full	consultStation(x)	Full	NotFound
19	x existe	Full	consultStation(x)	Full	Détail de x
20	x n'existe pas	Full	deleteStation(x)	Full	NotFound
21	x existe	Full	deleteStation(x)	Normal	Ok

La réalisation des opérations `isInState(State)` et `setToState(State)` du composant Gestion est présentée dans la Figure 32.

setToState

```
Case : State in
Empty :
numberElement = 0
Normal :
numberElement =  t
// t est un nombre >0 et <max
( 0 <  t < max )
for (i = 0; i < t; i++)
{database[i] = i;}
Full:
numberElement = max
for (i = 0; i < max; i++)
{database[i] = i;}
default : continue
```

isInState

```
Case : State in
Stop : return
numberElement = =0
Normal : return
numberElement = = t
( 0 <  t < max )
Full : return
numberElement = =max
default : return false
```

Figure 32. Implémentation des opérations isInState et setToState du composant Gestion

3.5.2.2.3. Interface de test du composant thermomètre et du composant senseur

Ces composants fournissent une fonction unique qui donne la température de la salle. Pour tester ces composants, on doit simplement vérifier si la température est normale ou non (trop haute ou trop basse).

3.5.2.2.4. Interface de test du composant moteur de température

Ce composant contrôle et dirige la valve d'air chaud et la valve d'air froid pour ajuster la température fournie à la salle. Il fournit deux méthodes pour diriger les valves : `stop(name)` et `change(name, temperature)`. Le modèle d'états de ce composant est décrit dans la Figure 33.

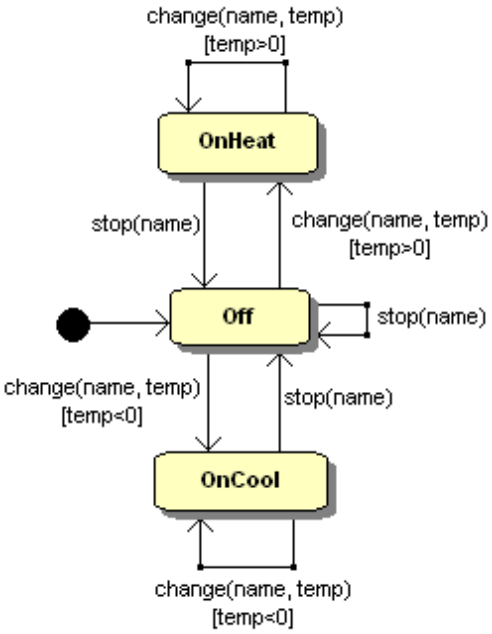


Figure 33. Modèle d'états du composant EngineTemp

Ce composant peut prendre trois états :

- L'état *"Off"* indique que la valve d'air chaud et la valve d'air froid d'une station sont arrêtées.
- Le composant est en état *"OnHeat"* si la valve d'air chaud est ouverte.
- Le composant est en état *"OnCool"* si la valve d'air froid est ouverte.

A partir du modèle d'état de ce composant, on a les cas de test décrits dans le Tableau 9.

Tableau 9. Les cas de test du composant EngineTemp

N° Cas de test	Entrées	Etat initial	Méthode à tester	Etat final
1		Off	stop(name)	Off
2	temps>0	Off	change(name, temp)	OnHeat
3	temps<0	Off	change(name, temp)	OnCool
4	temps>0	OnHeat	change(name, temp)	OnHeat
5		OnHeat	stop(name)	Off
6	temps<0	OnCool	change(name, temp)	OnCool
7		OnCool	stop(name)	Off

La réalisation des opérations `isInState(State)` et `setToState(State)` du composant EngineTemp est présentée sur la Figure 34.

setToState

```
Case : State in
Off :
hotValve=false
coldValve=false
OnHeat:
hotValve=true
OnCool:
coldValve=true
```

isInState

```
Case : State in
Off : return
hotValve == false
coldValve == false
OnHeat: return
hotValve == true
OnCool: return
coldValve==true
```

Figure 34. Implémentation des operations isInState et setToState du composant EngineTemp

3.5.2.3. Contexte en ligne

- Sachant que les tests des composants EngineTemp et Thermostat sont courts, nous optons pour la solution de blocage des composants pendant le processus de test, Le composant est restauré à l'état original après l'exécution du test.
- Le test des composants Thermomètre et Senseur n'influencent pas leur fonctionnalité nominale.
- Pour tester le composant Gestion, nous optons pour la solution de session de test. Nous avons créé des données dédiées, par exemple, nous créons un thermomètre pour le test de la suppression. Il n'y a donc pas d'interférence avec le fonctionnement du composant.

3.6. Conclusion

Dans ce chapitre, nous avons présenté le principe du test inter-composant, un élément important pour construire le service de diagnostic en ligne. L'idée principale est d'intégrer une interface de test dans les composants qui rend le composant testable.

L'intégration de cette interface de test nécessite deux phases préliminaires qui consistent respectivement à générer le modèle d'états associé au composant puis à générer les cas de test à partir du modèle d'états. Dans nos travaux, ces deux phases ont été réalisées manuellement. Une perspective intéressante de ce travail consiste alors à les automatiser afin de simplifier la prise en compte du test inter-composant.

Dans le chapitre suivant, nous présentons l'architecture et l'implémentation du service de diagnostic DISCO qui se fonde sur l'utilisation des tests inter-composants.

Chapitre 4

SERVICE DE DIAGNOSTIC DISCO

Dans le chapitre 2, nous avons présenté différentes approches de tolérance aux fautes pour les applications à base de composants logiciels et avons positionné le diagnostic vis-à-vis de ces approches. L'analyse de ces approches nous a permis de conclure que le niveau service est le mieux adapté pour le diagnostic tel que nous l'envisageons, c'est-à-dire en se basant sur des tests inter-composants. En effet, l'approche service s'intègre naturellement dans le processus de développement à base de composants logiciels et permet de garantir les apports d'un tel processus dans la construction efficace d'applications informatiques. Dans ce chapitre, nous présentons le service de diagnostic DISCO que nous avons développé. Nous détaillons son architecture, l'implémentation de ses composants, ainsi que la procédure de diagnostic. Ces travaux ont été publiés et présentés dans les conférences internationales IEEE SIES'08 [BUI08], Qualita'09 [BUI09a] et ACM EFTS'07 [BUI07c].

4.1. Éléments de base du diagnostic

Tout d'abord, nous rappelons quelques notions de base du diagnostic de niveau système :

- Une relation de test entre un composant Ci et un composant Cj indique que le composant Cj est testé par le composant Ci.
- Les relations de test entre les composants forment un graphe orienté, appelé *graphe de test*.
- Un groupe de diagnostic est défini comme étant un groupe de composants qui emploient le même algorithme de diagnostic. Par exemple, dans la Figure 35, les composants du groupe 1 et du groupe 2 exécutent l'algorithme de diagnostic 1 et 2 respectivement.

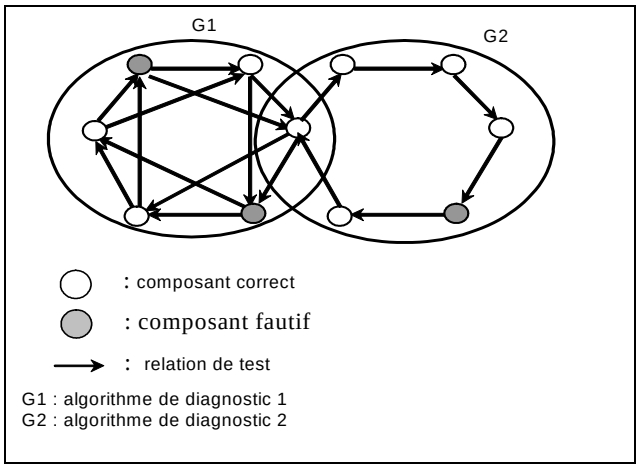


Figure 35. Groupes de diagnostic

- Le choix du partitionnement en groupes pour le diagnostic peut se faire selon divers critères, en particulier :
  - La localisation géographique : pour les systèmes distribués à base de composants logiciels, les composants peuvent être répartis sur différents sites. Les groupes de diagnostic sont alors déterminés par la localisation géographique des composants.
  - Le niveau de fiabilité des composants : notre volonté de diviser le système en groupes de diagnostic découle aussi du niveau de fiabilité des composants. A l'aide des informations statistiques sur les modes de défaillance des composants, nous pouvons par exemple grouper les composants souvent défaillants dans un groupe et utiliser un algorithme de diagnostic approprié de manière plus efficace, alors qu'un algorithme de diagnostic plus léger peut être utilisé pour un groupe moins défaillant.
- L'état d'un groupe correspond à l'état de faute[4] de ses composants (corrects ou défaillants).

4.2. Description du service de diagnostic DISCO

4.2.1. Procédure de diagnostic

La procédure de diagnostic est montrée dans la Figure 36.

Avant le déploiement de l'application, tous les composants sont analysés pour générer les modèles d'états et les cas de test. Une interface de test qui comprend tous ces cas de test est ajoutée dans chaque composant. Une fois les composants déployés, un algorithme de partitionnement est utilisé pour diviser l'application en groupes de diagnostic. En effet, pour pouvoir être extensible et s'adapter à différents types d'applications à base de composants, le recours au partitionnement de l'application en groupes de diagnostic nous a semblé être une stratégie prometteuse pour augmenter le degré de flexibilité du service DISCO.

La stratégie de test est ensuite appliquée dans chaque groupe ou de manière globale au niveau de toute l'application pour produire le graphe de test. Après le processus de diagnostic,

[4] de l'anglais : fault state.

si la topologie de l'application est modifiée, un nouveau graphe de test est obtenu pour le groupe concerné. Ceci rend le service de diagnostic flexible, et lui permet de s'adapter à l'évolution dynamique des applications.

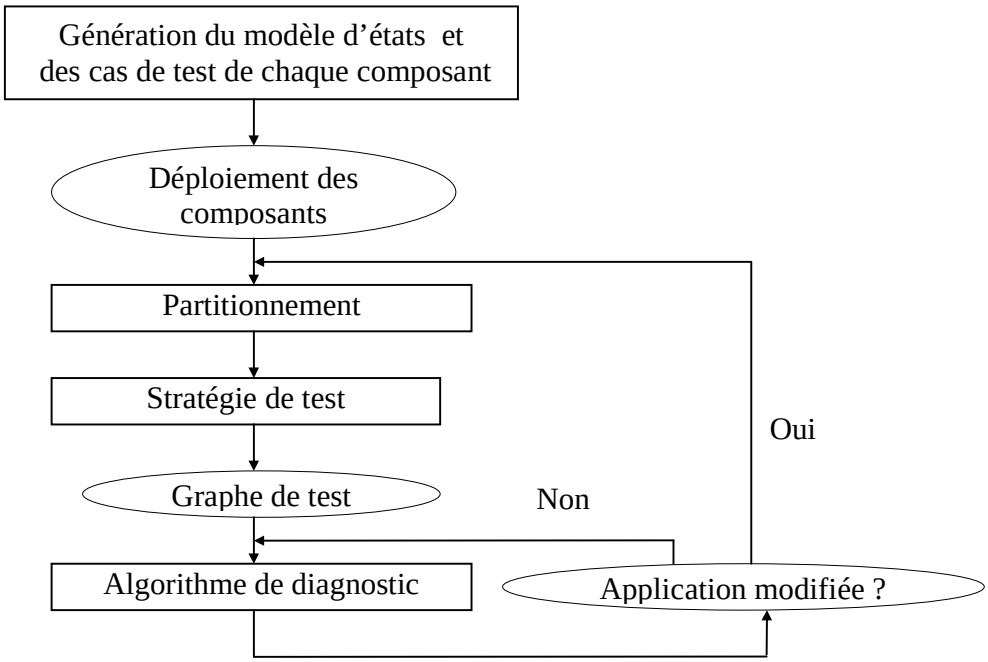


Figure 36. Procédure de diagnostic

La réalisation des différentes étapes de la procédure de diagnostic est assurée par les composants du service DISCO. Dans les parties suivantes, nous présentons l'architecture du service de diagnostic DISCO. La description de l'implémentation des composants de ce service utilise un modèle de composants générique, qui ne nécessite que des interfaces requises et des interfaces fournies.

4.2.2. Architecture globale du service de diagnostic DISCO

Notre objectif est de proposer un service de diagnostic qui permet de détecter et localiser des composants logiciels défaillants afin d'en permettre la correction ou le remplacement et préparer une éventuelle reconfiguration de l'application à base de composant. Ce service doit considérer les ressources disponibles et s'adapter afin de ne pas perturber le fonctionnement normal de l'application. De plus, il doit fournir différents algorithmes de diagnostic pour que l'utilisateur ait le choix de l'algorithme le mieux adapté à son application.

Pour satisfaire ces différentes contraintes, l'architecture du service DISCO comporte un certain nombre de composants, chargés chacun d'une mission précise. Cette architecture est présentée sur la Figure 37 et ses différents composants sont décrits ci-dessous.

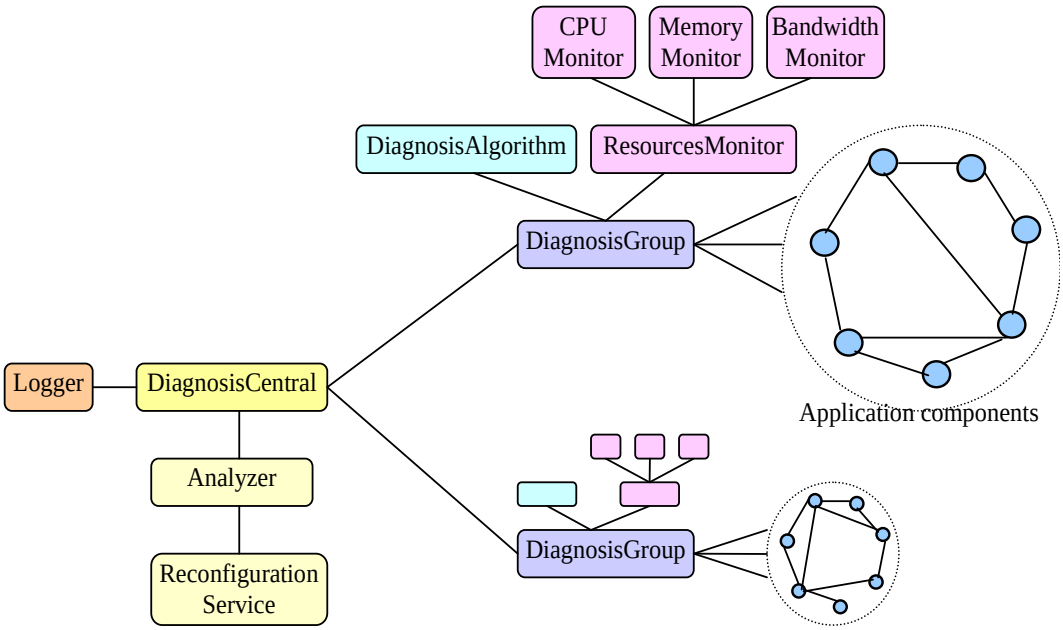


Figure 37. Architecture du service de diagnostic DISCO

Description des composants :

- *DiagnosisCentral* : est le composant de contrôle. Il divise l'application en groupes de diagnostic. Il a aussi pour rôle de gérer les groupes de diagnostic et de collecter l'état de faute des composants de tous les groupes. Pour cela, il enregistre localement les informations nécessaires sur tous les composants de l'application (nom, localisation, cas de test, *etc.*).
- *Logger* : a la responsabilité de répertorier les erreurs, pour l'analyse statistique, le rapport d'état de faute de l'application, *etc.* Ce composant fournit donc des informations utiles pour le partitionnement en groupes et le choix d'un algorithme de diagnostic approprié. Par exemple, nous pouvons grouper les composants souvent défaillants dans un groupe et utiliser un algorithme de diagnostic approprié de manière plus efficace, alors qu'un algorithme de diagnostic plus léger peut être utilisé pour un groupe moins défaillant. Ces informations statistiques peuvent aussi être utilisées pour déterminer la bonne période d'exécution du diagnostic.
- *DiagnosisAlgorithm* : fournit différents algorithmes de diagnostic, qui peuvent être choisis par l'utilisateur selon des critères précis.
- *ResourcesMonitor* : observe l'utilisation des ressources du système (CPU, mémoire, bande passante, *etc.*) afin d'informer le processus de diagnostic des ressources disponibles dans le système.
- *DiagnosisGroup* : gère et ordonnance les tests inter-composants dans un groupe selon l'algorithme de diagnostic choisi et en tenant compte des ressources disponibles dans le système.
- *ApplicationComponents* : sont les composants de l'application fournissant des fonctionnalités de test.

- *Analyzer* : reçoit les informations transmises par le composant *DiagnosisCentral* sur les composants défaillants. Ces informations, comme la journalisation des erreurs, ou encore les cas de tests, seront analysées pour déterminer le type de fautes produites par les composants défaillants et déclencher le processus de reconfiguration.
- Le service de diagnostic fonctionne en étroite liaison avec un service de reconfiguration pour assurer la continuité de service de l'application. Le service de reconfiguration *ReconfigurationService* fournit différentes stratégies de reconfiguration, par exemple, la réparation en ligne, le remplacement de composants ou l'arrêt du système. Ce service sort du cadre de notre étude.

Nous décrivons ci-dessous, les détails de réalisation de ces différents éléments.

4.3. Implémentation

Dans cette partie, nous détaillons l'implémentation des différents éléments du service DISCO.

4.3.1. Architecture orientée composants

Dans l'architecture du service DISCO, chaque élément est considéré comme un composant. Les composants et leurs interactions dans l'architecture DISCO sont illustrés sur la Figure 38.

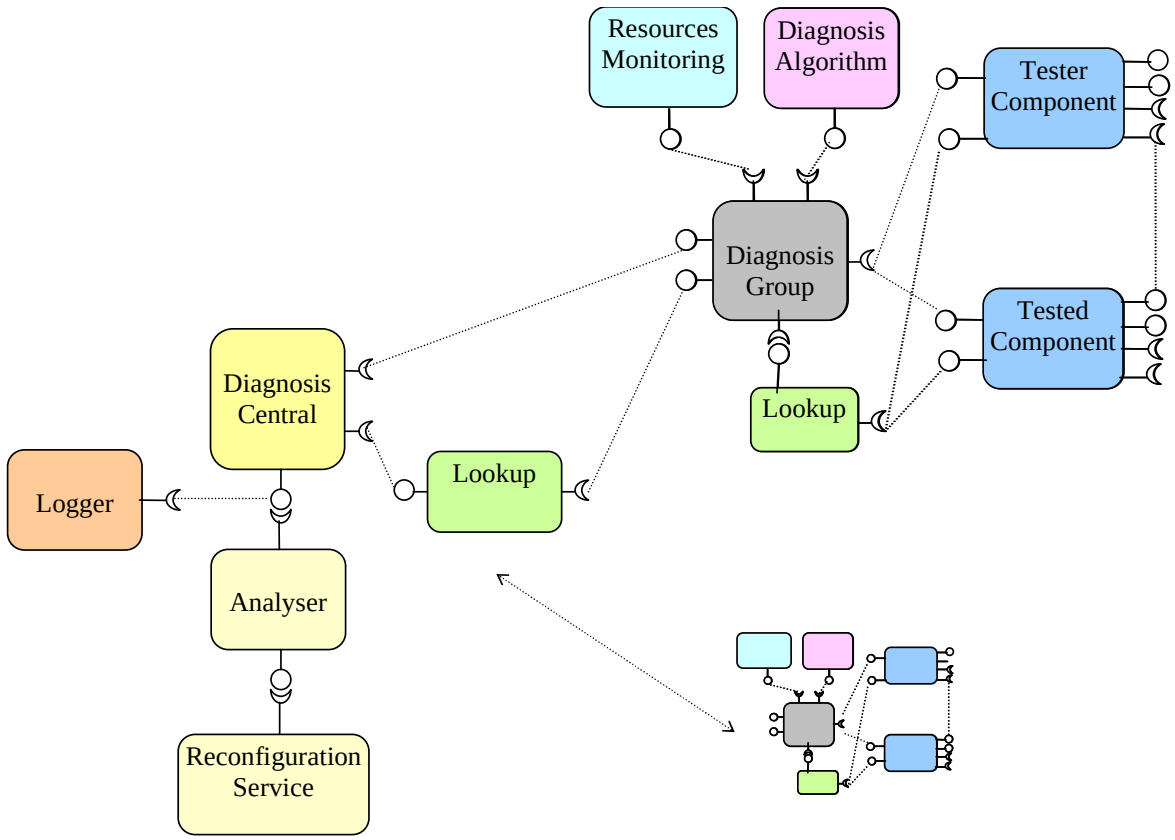


Figure 38. Composants du service DISCO

4.3.2. Composant DiagnosisCentral

Le composant DiagnosisCentral a pour rôle de contrôler et de gérer tous les groupes de diagnostic dans l'application. Ses fonctionnalités principales sont :

- Partitionnement de l'application en groupes de diagnostic. Quand le système comporte un nombre élevé de composants, le recours au partitionnement offre de meilleures performances par rapport aux approches sans partitionnement (qui couvrent le système entier) en réduisant le trafic des messages et le temps d'exécution, selon l'étude décrite dans [BAR93b].
- Gestion des groupes de diagnostic en demandant aux composants DiagnosisGroup correspondants de les diagnostiquer.
- Observation de l'état des composants dans un groupe ou dans toute l'application. Il fournit alors les résultats de diagnostic aux composants Logger et Analyzer.

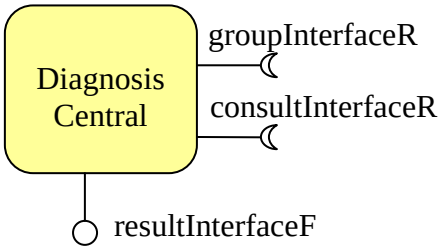


Figure 39. Interfaces du composant DiagnosisCentral

La Figure 39 présente les interfaces du composant DiagnosisCentral qui :

- Utilise l'interface *groupInterfaceR* pour transférer les paramètres de diagnostic à chaque groupe.
- Utilise l'interface *consultInterfaceR* pour consulter les informations des composants de l'application.
- Fournit les résultats de diagnostic au composant Logger et Analyzer via l'interface *resultInterfaceF*.

Ce composant fournit aussi une interface graphique qui permet à l'utilisateur de spécifier les paramètres du diagnostic, par exemple l'algorithme de diagnostic ; le mode d'exécution du diagnostic (une seule exécution ou plusieurs exécutions périodiques) ; les seuils de disponibilité des ressources permettant l'exécution des tests. Cette interface est montrée dans l'annexe D.

4.3.3. Composant DiagnosisAlgorithm

Le composant DiagnosisAlgorithm fournit différents algorithmes de diagnostic. Les algorithmes de diagnostic diffèrent principalement par la stratégie d'application des tests inter-composants et la procédure d'analyse des résultats des tests produits. Toutefois, ils reposent tous sur l'exécution de tests inter-composants et la transmission des résultats de tests à d'autres composants (diagnostic distribué) ou à un composant central (diagnostic centralisé). Ainsi, nous pouvons tous les représenter au moyen de triplets de la forme (composant testeur, composant testé, composants destinataires des informations de test).

Nous avons utilisé pour cela une table de la forme :

Composant testeur	Composant testé	Composants destinataires des informations de test

Dans le cas d'un diagnostic centralisé, tous les composants testeurs doivent envoyer les résultats de tests qu'ils obtiennent au composant central, supposé fiable pour qu'il puisse analyser ces résultats et déterminer l'état de faute des composants. Dans ce cas, le composant central est l'unique destinataire des résultats de test.

En revanche, dans le cas du diagnostic distribué, les résultats de tests obtenus par les composants testeurs sont envoyés à leurs propres composants testeurs.

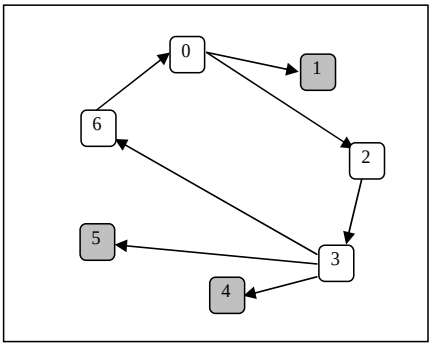


Figure 40. Un exemple d'un graphe de test

Considérons l'exemple de la Figure 40 où les composants 1, 4, 5 sont défaillants et les autres sont corrects. L'idée principale du diagnostic distribué est qu'un composant n'accepte que les informations en provenance de composants qu'il teste et qu'il trouve corrects (voir annexe A).

Tableau 10. L'extrait de la table des relations de test

Composant testeur	Composant testé	Composants destinataires
0	1	6
0	2	6
2	3	0
3	4	2
3	5	2
3	6	2
6	0	3

L'extrait des relations de test est présenté dans le Tableau 10. Ce tableau montre que, par exemple, le composant 0 teste le composant 2 et envoie le résultat du test au composant 6.

Pour les algorithmes de diagnostic statique, les relations de tests sont déterminées de manière statique avant le déploiement des composants. Ce tableau est alors établi une fois pour toutes avant le processus de diagnostic. Par contre, pour les algorithmes de diagnostic adaptatif, ce tableau est mis à jour de manière dynamique après chaque test inter-composant. En effet, dans ce cas, les relations de tests sont déterminées de manière dynamique en tenant compte des résultats des tests antérieurs. La mise à jour continue de ce tableau se fait alors selon l'algorithme suivant :

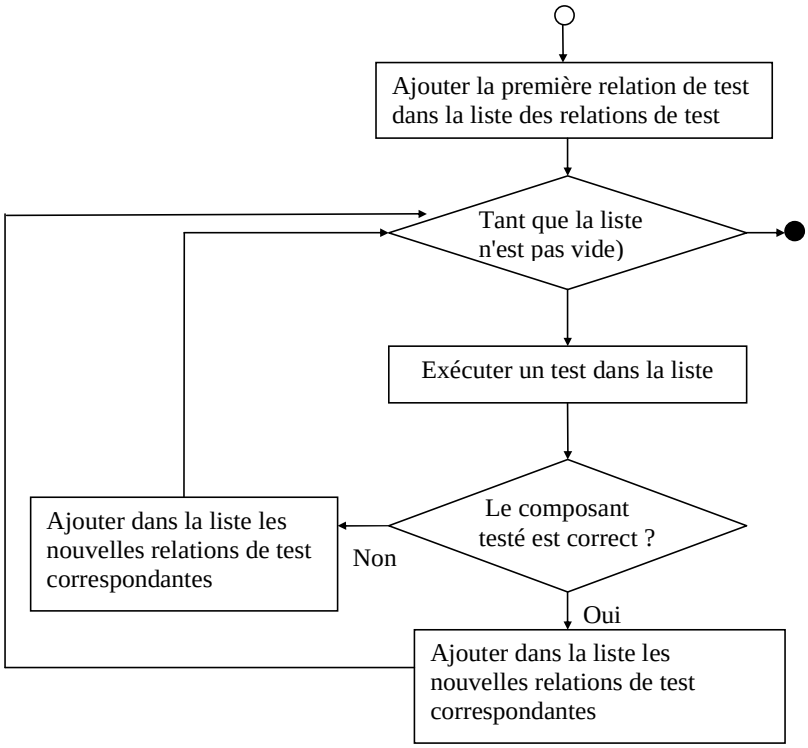


Figure 41. Algorithme de mise à jour du tableau des relations de test dans le cas d'un diagnostic distribué adaptatif

4.3.4. Composant DiagnosisGroup

Le rôle principal de ce composant est de diagnostiquer un groupe de composants selon l'algorithme de diagnostic choisi par l'utilisateur. Il fournit les fonctionnalités suivantes :

- Déterminer la stratégie de test pour son groupe à l'aide du composant DiagnosisAlgorithm.
- Analyser les informations du composant ResourcesMonitoring (voir 4.3.8) pour bénéficier des ressources disponibles des composants et du système pour les tests inter-composants.
- Envoyer les informations de test aux composants de son groupe.
- Récupérer l'état de son groupe.

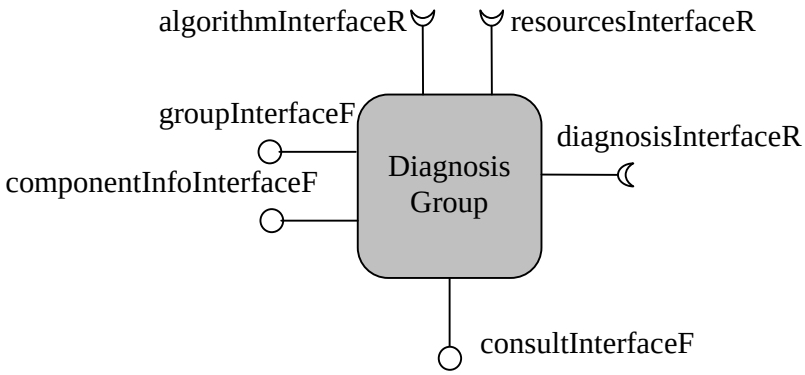


Figure 42. Interfaces du composant DiagnosisGroup

La Figure 42 présente les interfaces du composant DiagnosisGroup. On peut noter que le composant DiagnosisGroup :

- Implémente l'interface *groupInterfaceF* qui fournit le service pour diagnostiquer un groupe de composants selon des paramètres de diagnostic transmis par le composant DiagnosisCentral.
  - Cette interface comprend la méthode `Result[] diagnosisRequest( List[] parametres)` qui transfère les paramètres de diagnostic choisis par l'utilisateur pour ce groupe et retourne comme résultat l'état des composants dans ce groupe.
- Fournit les informations des composants dans son groupe de diagnostic via l'interface *componentInfoInterfaceF*.
- Utilise l'interface *algorithmInterfaceR* pour déterminer la stratégie de test ou des relations de test.
- Utilise l'interface *resourcesInterfaceR* pour observer les ressources du système et l'ordonnancement des tests. Si l'utilisateur souhaite vérifier les ressources disponibles du système avant d'exécuter un test, il contrôle périodiquement la disponibilité des ressources avant d'envoyer la demande de test au composant testeur.
- Utilise l'interface *diagnosisInterfaceR* pour envoyer les demandes de test aux composants de son groupe. La demande de test est définie à l'aide de la stratégie de test transmise par le composant DiagnosisAlgorithm, en particulier le tableau des relations de test (Tableau 10) présenté dans la section 4.3.3. La demande de test est extraite du tableau des relations de test en ajoutant les cas de test à utiliser qui peuvent être déterminés en considérant les ressources du système. Elle se présente sous la forme d'une structure comme suit :

Composant testeur	Composant testé	Cas de test	Composants destinataires des informations de test

4.3.5. Composant Lookup

Le composant Lookup est un service de découverte qui gère un catalogue contenant les informations sur les composants testables dans le système. Le catalogue est centralisé sur le site de contrôle (DiagnosisCentral) ou distribué dans chaque groupe (DiagnosisGroup). Le

composant Lookup maintient une base de données des informations des composants testables qui sont présentes dans le système ou dans un groupe. Ces informations concernent principalement le nom et la localisation du composant, ainsi que les cas de test fournis par le composant.

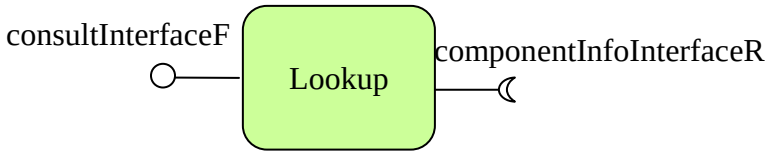


Figure 43. Interfaces du composant Lookup

Ce composant fournit les informations des composants via l'interface *consultInterfaceF* et utilise l'interface *componentInfoInterfaceR* pour rassembler les informations des composants (Figure 43). Il peut fournir des opérations pour modifier, mettre à jour, supprimer ou ajouter des informations sur les composants.

4.3.6. Composant d'application

Pour rendre les tests inter-composants possibles, nous ajoutons les interfaces supplémentaires aux composants de l'application (Figure 44).

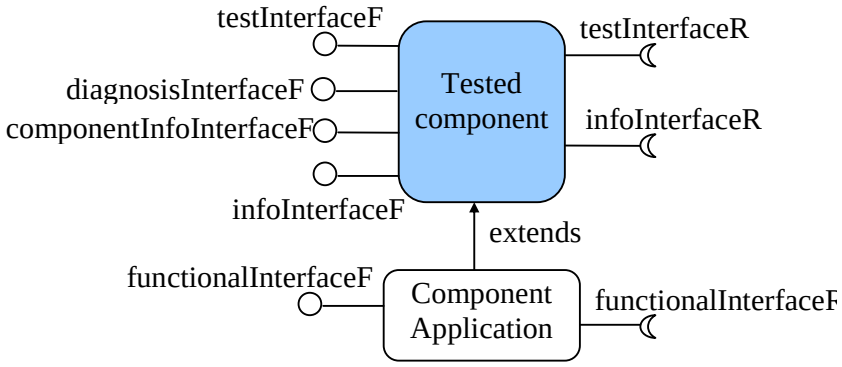


Figure 44. Interfaces d'un composant d'application

Ces interfaces fournissent les fonctions principales suivantes :

- Fournir les informations relatives au composant de l'application au composant Lookup correspondant : son nom, sa localisation, ses cas de test fournis, *etc.*
- Recevoir les demandes de test du composant DiagnosisGroup. Analyser chaque demande de test pour déterminer quel composant sera testé et avec quels cas de test.
- Exécuter la demande de test sur le composant indiqué.
- Envoyer les informations de test aux composants destinataires qui analysent les résultats de test pour déterminer l'état de faute des composants de l'application.
- Un composant d'application peut jouer le rôle du composant central dans le cas d'un diagnostic centralisé. Il analyse alors les résultats des tests inter-composants pour déterminer l'état de faute des composants de son groupe. Dans le cas d'un diagnostic distribué, ce sont tous les composants corrects d'un groupe qui déterminent de manière fiable l'état du groupe.

Un composant d'application peut être à la fois composant testeur et composant testé. Par souci de clarté, la description ci-dessous aborde successivement le rôle de composant testeur et le rôle de composant testé.

4.3.6.1. Du point de vue du composant testé

Les interfaces ajoutées au composant testé sont illustrées sur la Figure 45.

- Le composant testé fournit une interface *testInterfaceF* qui permet aux autres composants de le tester. L'interface de test est détaillée dans le chapitre 3. Cette interface se compose des méthodes suivantes :
  - `isInState(State)` pour vérifier si le composant réside dans un état donné.
  - `setToState(State)` pour affecter un état au composant.
  - `testComponent(int[] testcases)` : Cette méthode exécute les cas de test demandés en paramètres et retourne les résultats obtenus :

```
List[] testComponent(int[] testcases)
{
    for (i=0 ; i<testcases.length, i++)
    {
          if (testCases[i] = = 1)
          {    execute the testcase 1;
                testResults[0].add("TestCase 1");
                testResults[1].add(result);
          }
          if (testCases[i] = =2)
          {     execute the testcase 2;
                testResults[0].add("TestCase 2");
                testResults[1].add(result);
          }
          ….
     }
  return testResults;
  }
```

- Ce composant fournit aussi une autre interface *componentInfoInterfaceF* pour transmettre ses informations au composant Lookup.

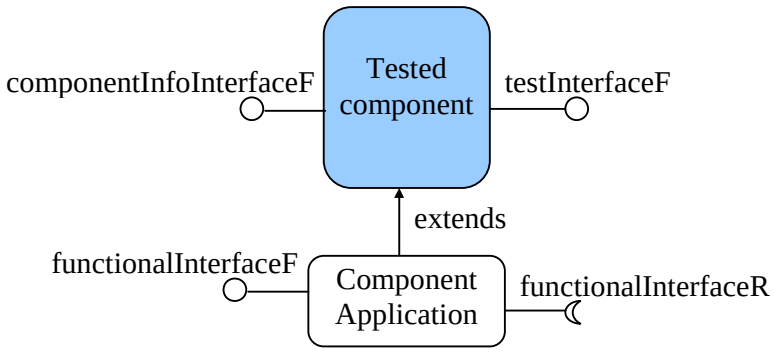


Figure 45. Interfaces du composant testé

4.3.6.2. Du point de vue du composant testeur

La Figure 46 montre les interfaces du composant testeur.

- Le composant testeur reçoit les demandes de test du composant DiagnosisGroup via l'interface *diagnosisInterfaceF*. Cette interface implémente la méthode `testRequest(componentTester, componentTested, testcases[])` qui transfère les demandes de test du composant DiagnosisGroup au composant testeur. Le composant testeur analyse cette demande de test pour déterminer le composant à tester et avec quels cas de test.
- Le composant testé fournit les fonctionnalités de test via l'interface *testInterfaceF*. Le composant testeur a besoin de cette interface pour tester ce composant avec les cas de test indiqués par le composant DiagnosisGroup.
- Dans les algorithmes de diagnostic distribué, un composant testeur récupère des informations transmises par des composants qu'il teste et qu'il trouve corrects. Ceci est possible grâce à l'interface *infoInterfaceR*.
- Après le test, le composant testeur envoie les résultats de test aux composants destinataires indiqués dans la demande de test (selon l'algorithme de diagnostic) via l'interface *infoInterfaceF*.

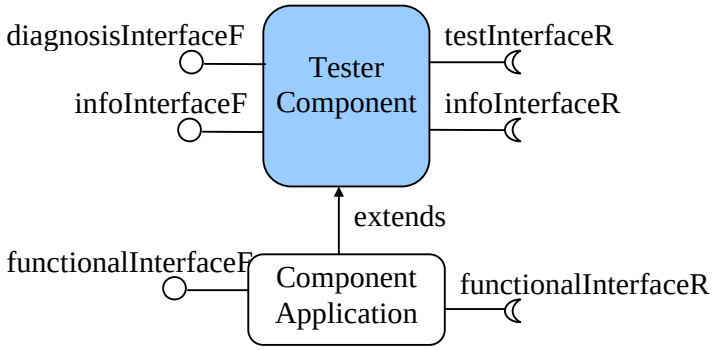


Figure 46. Interfaces du composant testeur

4.3.7. Composant Logger

Le composant collecteur a la responsabilité de répertorier les erreurs, pour l'analyse statistique, le rapport d'état de faute de l'application, *etc.* sous forme d'une base de données des états des composants de l'application. Ce composant est utile pour le partitionnement en groupes et le choix d'un algorithme de diagnostic approprié. Par exemple, nous pouvons grouper les composants souvent défaillants dans un groupe et utiliser un algorithme de diagnostic approprié de manière plus efficace, alors qu'un algorithme de diagnostic plus léger peut être utilisé pour un groupe moins défaillant. Ces informations statistiques peuvent aussi être utilisées pour déterminer la période appropriée pour le diagnostic ou la durée du diagnostic.

Les informations sur les erreurs produites peuvent être enregistrées sous la forme d'une base de données comme suit :

Composant défaillant	Date	Cas de test utilisés
Composant 1	12 :23 :00 12/10/2008	1
Composant 2	02 :23 :12 23/10/1008	3

4.3.8. Composant ResourcesMonitoring

L'objectif à travers l'utilisation des moniteurs de ressources est d'exploiter au mieux les ressources disponibles du système pour augmenter l'efficacité du service DISCO en perturbant le moins possible la fonctionnalité de l'application. Nous avons utilisé les moniteurs de ressources pour observer les ressources et déclencher les tests selon la limite des ressources disponibles définie par l'utilisateur. Ainsi, les tests sont exécutés quand la disponibilité des ressources du système dépasse un certain seuil. Les ressources ici peuvent être la mémoire, la bande passante ou le CPU. Dans le cadre de cette thèse, nous nous intéressons uniquement à la ressource mémoire. Cependant, le service DISCO peut facilement être enrichi pour la prise en compte d'autres types de ressources.

Pour exécuter les tests en fonction de la mémoire disponible dans le système, il est nécessaire de pouvoir mesurer la mémoire pendant l'exécution. Pour mesurer la mémoire disponible du système, une première approche consiste à utiliser le gestionnaire des tâches de Microsoft Windows (qui permet de mesurer la mémoire réservée pour la machine virtuelle Java (JVM) [WTM]. Malheureusement, cette approche est peu pratique, puisque l'utilisateur est obligé de noter "à la main" les mesures. Une approche sans doute plus intéressante vise à insérer des sondes dans le code source, à l'aide de la classe Runtime de Java [RUN]. Dans cette classe, il y a 2 méthodes intéressantes :

- `Runtime.totalMemory()` : retourne la taille totale (en octets) de la mémoire dont dispose la Machine Virtuelle Java.
- `Runtime.freeMemory()` : retourne la quantité de mémoire libre du système.

La mémoire utilisée par l'application est égale à (`Runtime.totalMemory()-Runtime.freeMemory()).`

Nous avons utilisé cette approche pour mesurer la mémoire disponible du système.

4.3.9. Les composants Analyzer et ReconfigurationService

Les deux composants Analyzer et ReconfigurationService ne sont pas pris en compte dans notre travail. La reconfiguration en ligne d'une application à base de composants logiciels est étudiée de manière approfondie dans [BAT00][KET04]. Parmi les approches possibles, nous pouvons citer la réparation en ligne ou le remplacement du composant défaillant, ou encore l'arrêt du système. La plupart des approches sont fondées sur le remplacement des composants défaillants. Le principe est de remplacer le composant défaillant par une nouvelle version ou un nouveau composant qui fournit le même service. Ensuite, l'état du composant à remplacer est transmis au composant remplaçant, en suivant les règles de correspondance. L'implémentation de ce mécanisme utilise aussi deux opérations équivalentes aux opérations `isInState(State)` et `setToState(State)`vues précédemment. La première vérifie si le composant réside dans un état. La deuxième affecte l'état du composant.

Dans notre approche, ces deux opérations sont ajoutées dans l'interface de test de chaque composant à tester. Par conséquent, elles peuvent servir pour la mise en œuvre le la reconfiguration du système.

4.4. Fonctionnement du service de diagnostic

Le fonctionnement du service de diagnostic se compose des étapes suivantes : le déclenchement du diagnostic (sous-section 4.4.1), l'exécution du diagnostic (sous-section 4.4.2) et la fin du diagnostic (sous-section 4.4.3).

4.4.1. Déclenchement du diagnostic

Le service de diagnostic peut être lancé à différents moments, en fonction d'une des conditions suivantes :

- Changement de la topologie de l'application : si la topologie change (des composants sont remplacés, enlevés ou ajoutés), le service de diagnostic est déclenché pour vérifier si l'application fonctionne correctement.
- Périodicité : le service de diagnostic est exécuté périodiquement selon une période prédéterminée.
- Demande de l'administrateur du système : l'administrateur peut exécuter le diagnostic pour connaître l'état de faute des composants de l'application à n'importe quel instant.
- Contraintes de ressources : quand le système a suffisamment de ressources disponibles, le service de diagnostic peut être lancé. L'administrateur du système peut fixer les limites des ressources disponibles pour le diagnostic.

Ces *scénarii* peuvent être choisis par l'utilisateur au moment de la configuration du système. Un ou plusieurs *scénarii* peuvent être combinés en même temps.

4.4.2. Exécution du diagnostic

Une fois déclenché, le service de diagnostic fonctionne comme illustré par la Figure 47. Tout d'abord, les paramètres spécifiés par l'utilisateur sont transmis par le composant DiagnosisCentral aux différents composants DiagnosisGroup (1). Chaque composant DiagnosticGroup charge l'algorithme de diagnostic choisi par l'utilisateur pour son groupe (2). Le moniteur de ressources observe les ressources et envoie au composant DiagnosticGroup les informations sur les ressources disponibles (3). Ensuite, le composant DiagnosticGroup ordonnance les tests inter-composants en fonction de la stratégie de test et des ressources disponibles du système (et éventuellement des résultats de test antérieurs si l'algorithme de diagnostic est adaptatif). La demande de test est envoyée au composant d'application (4). Ce composant testeur analyse cette demande de test pour savoir quel composant il testera avec quels cas de test.

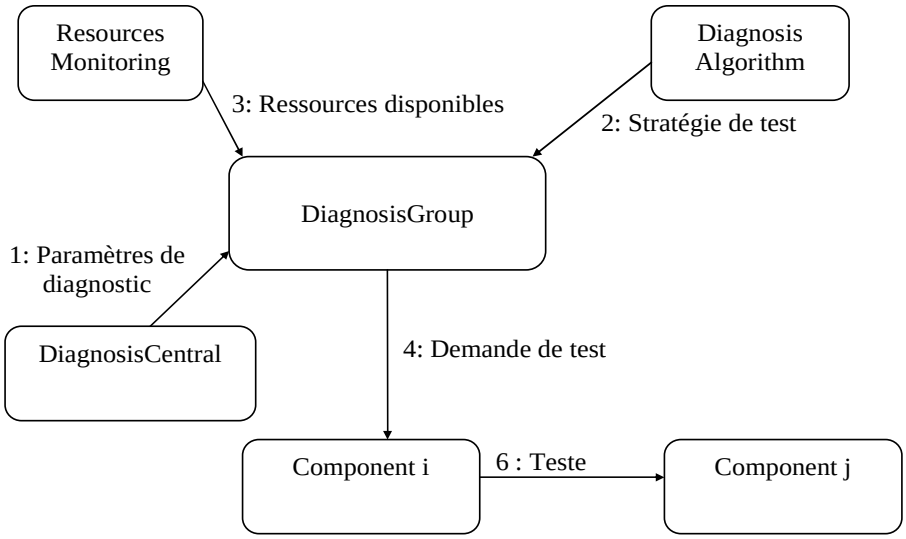


Figure 47. Processus de diagnostic

Les interactions entre les membres dans un groupe et le composant DiagnosticGroup sont illustrées dans la Figure 48. Après la réception de la demande de test (1), le composant testeur effectue les tests demandés (2) et transmet les résultats obtenus aux composants destinataires (4) qui sont définis par le composant DiagnosisAlgorithm selon une stratégie de test prédéterminée.

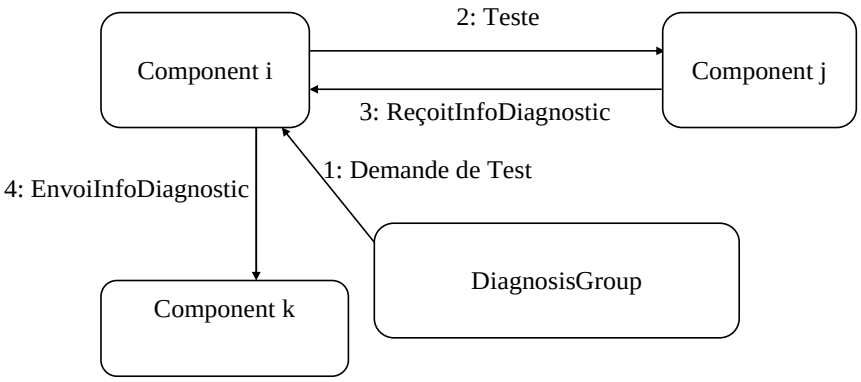


Figure 48. Interaction entre le composant DiagnosticGroup et les composants membres d'un groupe

Le composant qui est responsable du processus de diagnostic, c'est-à-dire, celui qui analyse les résultats des tests inter-composants, maintient un tableau de résultats. Après analyse de ce tableau, ce composant détermine l'état de faute des composants de son groupe.

4.4.3. Fin du diagnostic ou d'une période de diagnostic

La fin du diagnostic est décrite par la Figure 49. Les composants qui effectuent le diagnostic (en analysant les résultats des tests inter-composants) sont spécifiés et le composant DiagnosisGroup récupère l'état de son groupe (1). Il envoie ensuite ces résultats au composant DiagnosisCentral (2). Le composant DiagnosisCentral informe l'analyseur et le collecteur des composants fautifs.

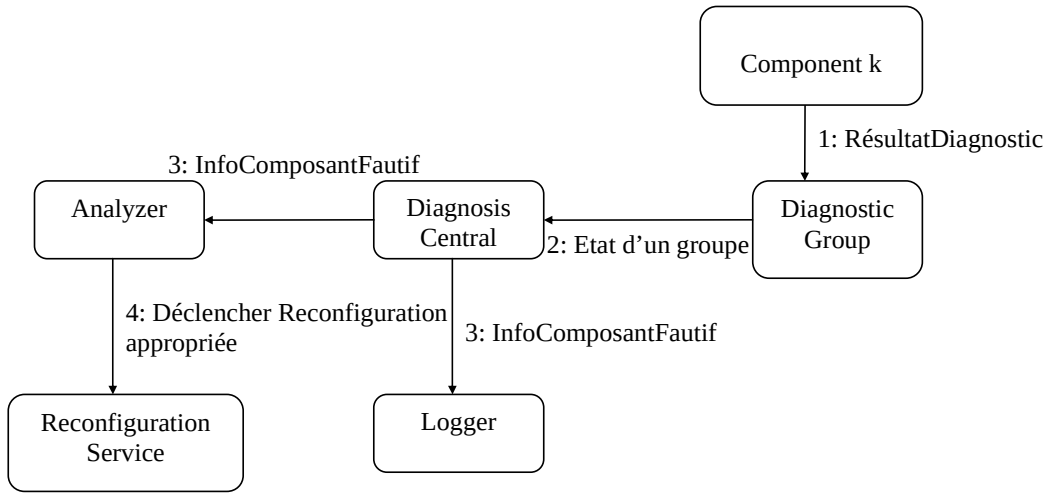


Figure 49. Fin du service de diagnostic

L'analyseur se sert de ces informations pour déclencher le service de reconfiguration (4) qui ne fait pas partie de l'objet de cette thèse.

4.5. Conclusion

Dans ce chapitre, nous avons présenté le service de diagnostic DISCO, son architecture, l'implémentation détaillée de ses composants, ainsi que la procédure de diagnostic. Nous avons développé une architecture générale pour ce service afin de l'appliquer à différents modèles de composants. Ce service est flexible car il tient en compte des évolutions dynamiques des applications à base de composants après chaque période de diagnostic.

Le service de diagnostic DISCO permet de détecter et localiser des composants défaillants afin de préparer une éventuelle reconfiguration de l'application. Il faut noter que la phase de reconfiguration n'est pas prise en compte dans notre travail mais qu'elle a fait l'objet d'autres travaux [BAT00][KET04]. Le service de diagnostic peut fournir différents algorithmes de diagnostic et considère des ressources disponibles afin de ne pas perturber le fonctionnement normal de l'application. Dans le cadre de cette thèse, nous nous sommes principalement intéressés à l'utilisation de la ressource mémoire.

Ainsi, une perspective directe de ce travail consiste en la prise en compte du seuil de disponibilité d'autres ressources, en particulier le CPU. La prise en compte de la ressource CPU nécessite *a priori* des interactions avec le système d'exploitation sous-jacent, ce qui peut minimiser la portabilité du service de diagnostic DISCO. Des réflexions supplémentaires sont donc nécessaires pour permettre une telle prise en compte tout en conservant les qualités d'indépendance et de portabilité du service DISCO.

Dans le chapitre suivant, nous présenterons une validation expérimentale du service de diagnostic DISCO sur une plate-forme d'expérimentation que nous avons implémentée. Le portage du service de diagnostic DISCO sur un cas d'étude industriel est également décrit.

Chapitre 5

EXPERIMENTATION ET CAS D'ETUDE

Dans ce chapitre, nous présentons une analyse expérimentale du service de diagnostic DISCO. Notre objectif est de mesurer les surcoûts du diagnostic et analyser les performances relatives du diagnostic centralisé et du diagnostic distribué. Nous présentons également un cas d'étude pour le service DISCO qui consiste en un système de gestion à grande échelle de données de capteurs hétérogènes. Ce travail a donné lieu à une publication dans IEEE Depend'09 [BUI09b].

5.1. Validation expérimentale

5.1.1. Plate-forme expérimentale

Nous avons utilisé la technologie de composants OSGi pour implémenter une plate-forme d'expérimentation afin de valider le fonctionnement du service de diagnostic DISCO et mesurer ses performances. Aujourd'hui, les composants logiciels s'exécutent dans des environnements complexes qui peuvent évoluer dynamiquement, de même que la structure de l'application peut changer. Il y a donc un besoin accru d'environnements d'exécution dynamique, qui permettent d'installer et désinstaller des composants à tout moment. La technologie de composant OSGi remplit ces critères. En outre, sa prise en main est relativement simple et intuitive.

Plusieurs implémentations de la technologie OSGi sont disponibles librement (e.g. Oscar [OSC], Knopflerfish [KNO], IBM Service Management Framework [SMF]). Nous avons utilisé l'implémentation Oscar développée au sein de l'équipe ADELE [ADE] du laboratoire LIG car cette implémentation nous fournissait tous les aspects nécessaires à notre développement.

La plate-forme développée permet de spécifier le nombre de composants à considérer dans l'application sachant que chaque composant peut fournir une ou plusieurs interfaces. Pour cette étude, nous avons considéré le cas de 21 composants interconnectés selon le schéma de la Figure 50. Le composant 21 représente une interface graphique et n'est pas pris en compte dans les expérimentations ci-dessous. Chaque interface fournie par ces composants exécute une fonction de calcul. Les fonctions de calcul considérées sont la moyenne de n nombres générés au hasard ou la factorielle d'un entier n.

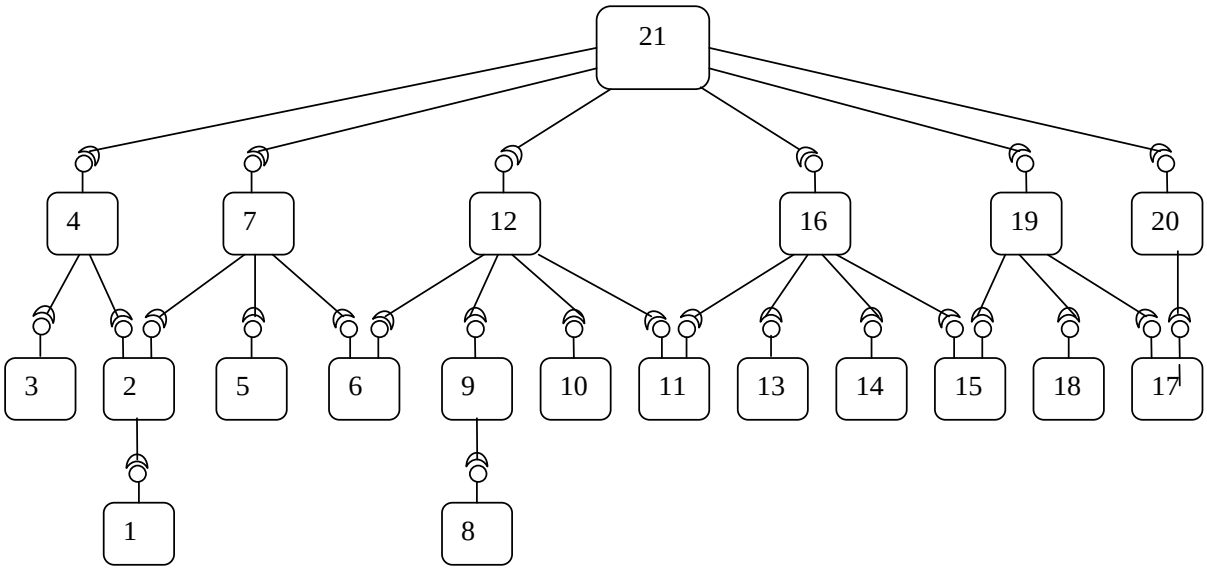


Figure 50. Connexions entre les composants

L'interface de test est intégrée dans chaque composant. Cette interface exécute une fonction, par exemple, le calcul d'une moyenne ou d'une factorielle pour simuler le test de ce composant.

Cette plate-forme nous permet de modifier de manière souple les paramètres de diagnostic pour analyser les performances du service DISCO.

5.1.2. Démarche expérimentale

L'objectif de ces expérimentations est de mesurer le surcoût dû à l'introduction du diagnostic et d'analyser les performances du service DISCO en utilisant successivement l'algorithme de diagnostic centralisé et l'algorithme de diagnostic distribué.

L'utilisation du service de diagnostic induit nécessairement un coût supplémentaire en mémoire et en temps processeur. Dans ces expérimentations, nous nous sommes intéressés à l'évaluation du coût en mémoire du service DISCO. Pour cela, nous avons mesuré la mémoire physique occupée par le service DISCO ainsi que la mémoire utilisée durant l'exécution du service DISCO. Par la suite, nous avons étudié l'évolution du temps d'exécution d'une application lorsque celle-ci fait appel au service DISCO. Par ailleurs nous avons considéré successivement les deux algorithmes de diagnostic présentés dans le chapitre 2 et dont l'implémentation est donnée en annexe A.

Toutes les expérimentations ont été menées sur un ordinateur Dell doté d'un processeur Intel 1.60 GHz et de 512 MB RAM. Les paramètres des expérimentations sont précisés à chaque fois. Notons ici la signification du paramètre *période de diagnostic* qui exprime le temps entre deux exécutions successives du diagnostic complet.

5.1.3. Les résultats obtenus

5.1.3.1. Surcoût en mémoire du système par le service DISCO

Pour estimer le surcoût en mémoire du diagnostic, nous avons mesuré la mémoire physique occupée par le service DISCO, c'est-à-dire la mémoire de stockage sur le disque dur du service DISCO, ainsi que la mémoire utilisée durant l'exécution du service DISCO.

5.1.3.1.1. Mémoire physique occupée par le service DISCO

Le Tableau 11 montre les résultats obtenus pour les mesures de la mémoire physique nécessaire pour le stockage du service de diagnostic.

Tableau 11. Surcoût du service de diagnostic

Critères	**Surcoût du service DISCO**	**Surcoût dans chaque composant Dépend des cas de test (ici, 4 cas de test)**
Nombre de lignes de code	~1000	~200
Taille des sources	43.1 kB	6.94 kB
Taille des classes compilées	20.9 kB	5.19 kB
Taille de bundle	12.2 kB	2.58 kB

Ces mesures nous permettent d'observer que le diagnostic n'est pas gourmand en mémoire.

5.1.3.1.2. Mémoire utilisée durant l'exécution du service DISCO

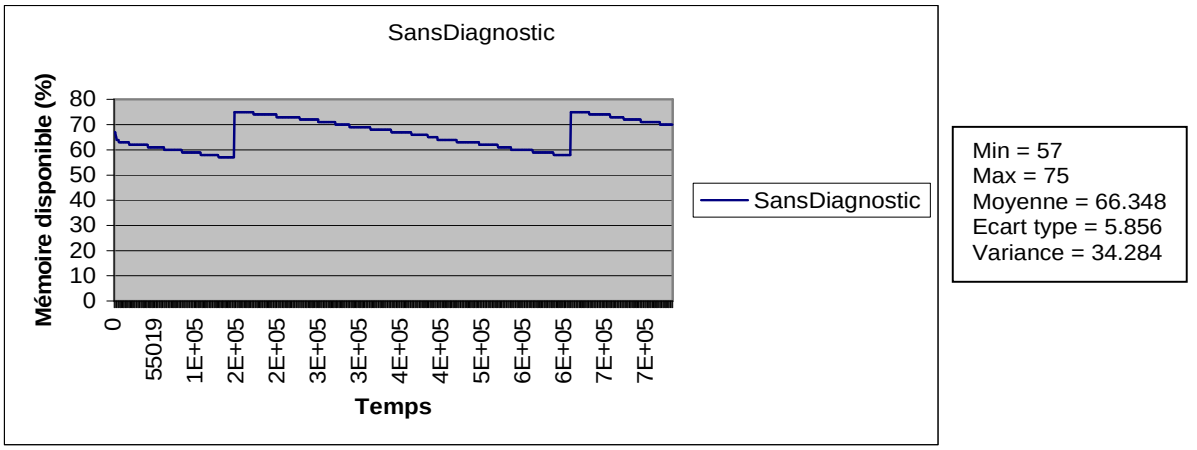
Pour estimer la mémoire utilisée durant l'exécution du service DISCO, nous avons mesuré la mémoire disponible de l'application sans diagnostic et avec diagnostic.

Pour cette expérimentation, les caractéristiques suivantes ont été considérées :

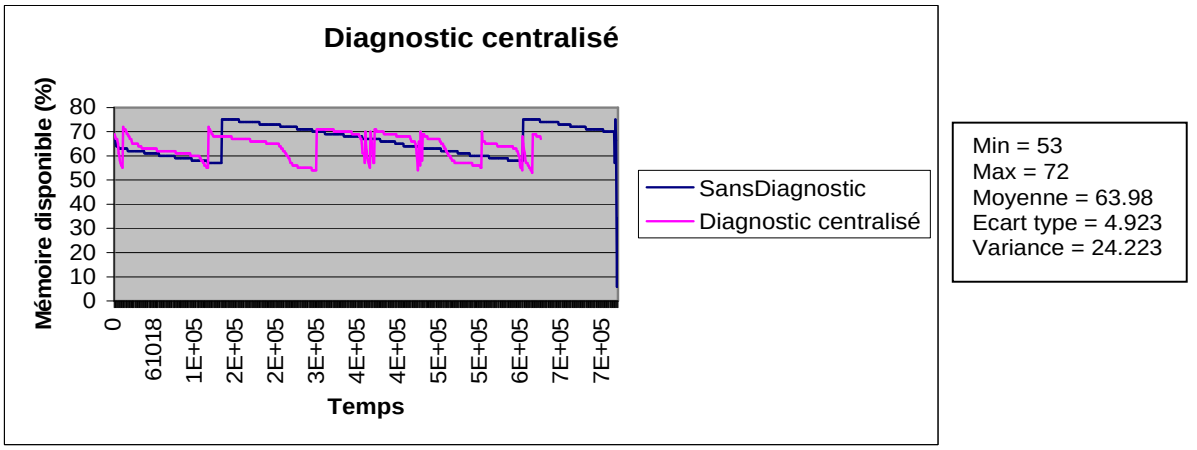
- Période de diagnostic : 100000 ms.
- Le temps d'exécution de chaque cas de test dans les composants correspond au temps de calcul de la moyenne de 10000 nombres générés de manière aléatoire.
- Il y a 4 cas de test dans le composant, et nous avons activé tous ces 4 cas de test.
- On considère 20 composants au total avec 1 composant fautif.

La Figure 51 montre les mesures de la mémoire disponible de l'application sans diagnostic (a), avec diagnostic centralisé (b) et avec diagnostic distribué (c). La mémoire disponible est calculée comme étant la différence entre la mémoire utilisée par l'application et la mémoire totale dédiée à la Machine Virtuelle Java (~5M).

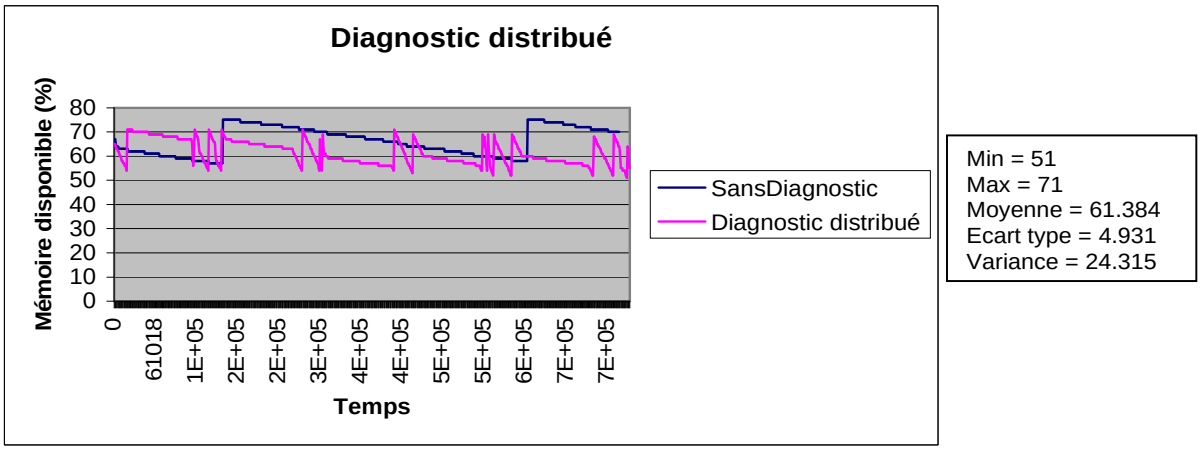
Les résultats obtenus, illustrés par la Figure 51, montrent que les surcoûts liés au service de diagnostic sont très faibles. En effet, la mémoire utilisée par le service est comprise entre 3 et 6 % de la mémoire totale.



(a) Application sans diagnostic



(b) Application avec diagnostic centralisé



(c) Application avec diagnostic distribué

Figure 51. Mesures de la mémoire disponible pendant l'exécution du service DISCO

5.1.3.2. Surcoût du diagnostic sur le temps d'exécution de l'application

Le temps d'exécution de l'application représente le temps nécessaire à l'exécution de tous les appels prévus entre composants. Pour estimer le surcoût du diagnostic sur le temps d'exécution de l'application, nous avons mesuré le temps d'exécution de l'application sans diagnostic et avec diagnostic. Les caractéristiques utilisées pour cette expérimentation sont :

- Période de diagnostic : 100000 ms.
- Le temps d'exécution de chaque cas de test dans les composants correspond au temps de calcul de la moyenne de 10000 nombres générés de manière aléatoire.
- Il y a 4 cas de test dans le composant, et nous avons activé tous ces 4 cas de test.
- On considère 20 composants au total avec 1 composant fautif.

Les mesures du temps d'exécution de l'application sans diagnostic et avec diagnostic centralisé et diagnostic distribué sont présentées dans le Tableau 12 :

Tableau 12. Le temps d'exécution de l'application sans diagnostic et avec diagnostic

	Sans diagnostic	**Avec diagnostic centralisé**	**Avec diagnostic distribué**
Temps d'exécution de l'application (ms)	21117.79	21121.85	21122.29

Les résultats présentés sont des moyennes obtenues pour 30 essais.

Nous remarquons que le surcoût de diagnostic en temps d'exécution est faible, la latence de l'application lorsque celle-ci fait appel au service DISCO est de 4 à 5 ms, c'est-à-dire de 0,019 à 0.021% de temps d'exécution de l'application, pour ce cas précis d'expérimentation.

5.1.3.3. Comparaison du diagnostic distribué et du diagnostic centralisé

5.1.3.3.1. Nombre de tests exécutés

La Figure 52 montre l'évolution du nombre de tests exécutés par le diagnostic centralisé et par le diagnostic distribué par rapport le nombre de composants fautifs dans le cas de 20 composants. Les résultats présentés sont des moyennes obtenues pour 20 essais.

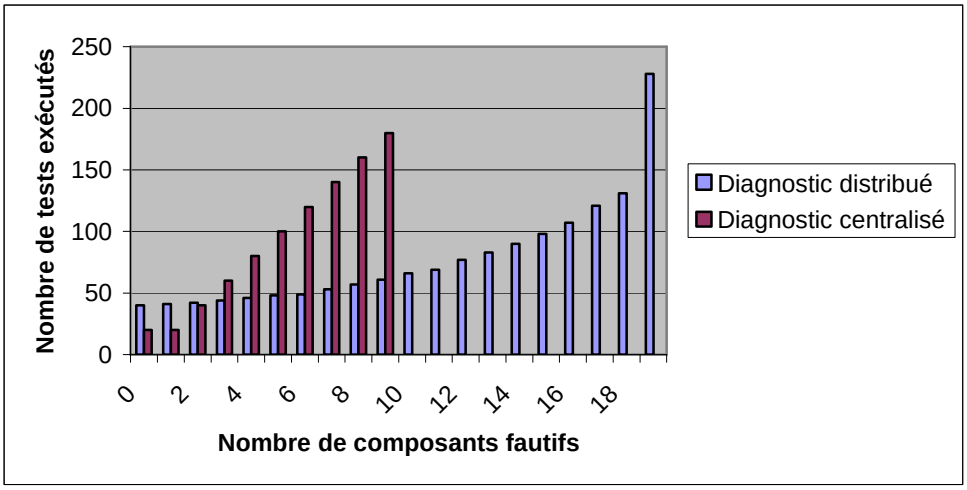


Figure 52. Nombre de tests exécutés par le diagnostic centralisé et le diagnostic distribué

Ce graphe nous permet de voir que le diagnostic distribué réduit le nombre de tests exécutés par rapport au diagnostic centralisé dans le cas où le système a plus de deux composants fautifs. Dans le cas où le système a un composant fautif, le nombre de tests exécutés par le diagnostic centralisé est plus intéressant. Ce résultat est conforté par les performances théoriques respectives du diagnostic centralisé et du diagnostic distribué. Le nombre maximal de tests de l'algorithme de diagnostic centralisé dépend du nombre

maximum de composants fautifs qui est égal au quotient de (n-1)/2 (noté |(n-1)/2|). Dans ce cas, le nombre de tests est égal à |(n-1)/2|*n. Pour l'algorithme de diagnostic distribué, le nombre de test maximal est égal à 1+2+…+(n-1)+n+(n-1), dans le cas où l'application a (n-1) composants défaillants.

5.1.3.3.2. Evolution du temps de diagnostic en fonction du nombre de composants

Le temps du diagnostic (ou latence du diagnostic) représente le temps nécessaire à la détection et la localisation des éléments défaillants dans le système. L'objectif ici est de déterminer l'évolution du temps de diagnostic par rapport au nombre de composants de l'application.

Les caractéristiques considérées pour cette expérimentation sont :

- 4 cas de test sont considérés pour chaque composant.
- 1 composant fautif dans l'application de 20 composants.
- Le temps d'exécution de chaque cas de test dans les composants est le temps de calcul de la moyenne de 10000 nombres générés au hasard.

La Figure 53 décrit l'évolution du temps d'exécution de l'algorithme de diagnostic centralisé et du diagnostic distribué en millisecondes par rapport au nombre de composants.

Les résultats présentés sont des moyennes obtenues pour 20 essais.

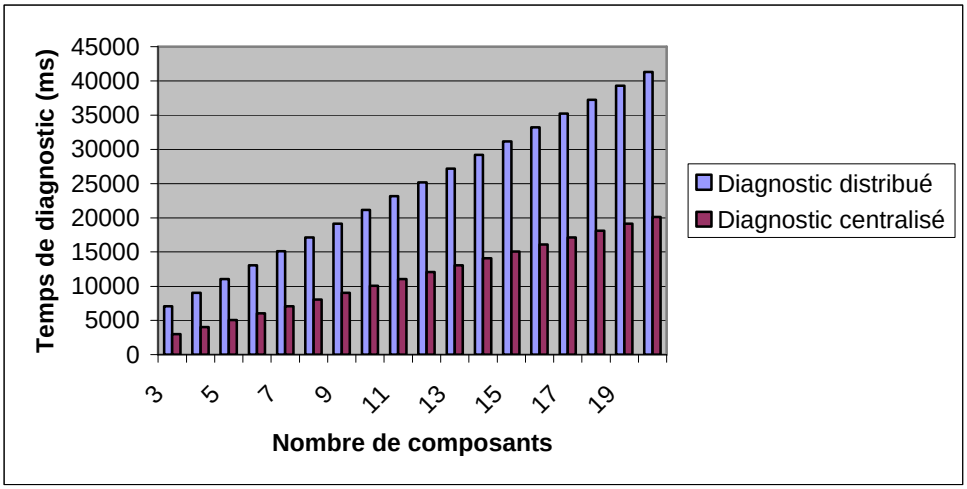


Figure 53. L'évolution du temps de diagnostic en fonction du nombre de composants

Le graphe de la Figure 53 nous permet de faire les observations suivantes :

- Le temps d'exécution de la méthode de diagnostic distribué est plus important que celui de la méthode de diagnostic centralisé car le système a un seul composant fautif.
- Quand l'application a un composant fautif, le temps de diagnostic distribué est presque deux fois plus grand que celui de la méthode de diagnostic centralisé.

5.1.3.3.3. Evolution du temps de diagnostic en fonction du nombre de composants fautifs

Les conditions d'expérimentation sont :

- Application de 20 composants.
- Le temps d'exécution de chaque cas de test dans les composants correspond au temps de calcul de la moyenne de 10000 nombres générés au hasard.
- 4 cas de test sont considérés pour chaque composant.

L'évolution du temps du diagnostic centralisé et du diagnostic distribué par rapport au nombre de composants fautifs est décrite dans le Figure 54.

Les résultats présentés sont des moyennes obtenues pour 20 essais.

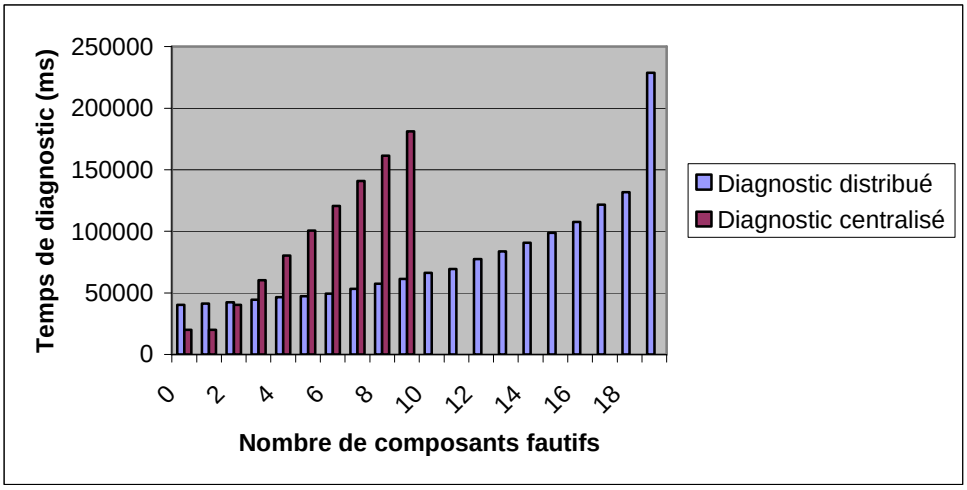


Figure 54. L'évolution du temps de diagnostic en fonction du nombre de composants fautifs

Le graphe de la Figure 54 nous permet de faire les observations suivantes :

- Quand le système a peu de composants fautifs (0, 1 ou 2 composants fautifs), le temps d'exécution de la méthode de diagnostic distribué est plus grand que celui de la méthode de diagnostic centralisé.
- Mais quand le système a plus de composants fautifs (plus de 2), le temps d'exécution du diagnostic distribué est plus petit que celui du diagnostic centralisé.
- Le temps d'exécution de la méthode de diagnostic centralisé est très sensible au nombre de fautes. Il présente une évolution quasi proportionnelle au nombre de composants fautifs.
- Le temps d'exécution de la méthode de diagnostic distribué est moins sensible au nombre de fautes.
- Le temps d'exécution de la méthode de diagnostic distribué est plus sensible au nombre de composants fautifs lorsque ce nombre est relativement élevé. En effet, le dernier cas correspond à un nombre de fautes égal au nombre de composants moins 1 (t=n-1). C'est le pire cas pour l'algorithme de diagnostic distribué.

5.1.3.3.4. Evolution du temps de diagnostic en fonction de la durée du test

La durée de test des composants est le temps principal intervenant dans le temps de diagnostic. Pour déterminer l'évolution du temps du diagnostic par rapport à la durée de test,

nous avons fait des expérimentations de changement de la durée de test sur le système. La même durée est considérée pour tous les tests.

Les paramètres considérés sont :

- Application de 20 composants.
- 1 composant fautif.
- 4 cas de test sont considérés pour chaque composant.

Les résultats obtenus sont décrits dans la Figure 55 qui présente l'évolution du temps d'exécution de l'algorithme de diagnostic centralisé et du diagnostic distribué par rapport à la durée de test en millisecondes.

Les résultats présentés sont des moyennes obtenues pour 20 essais.

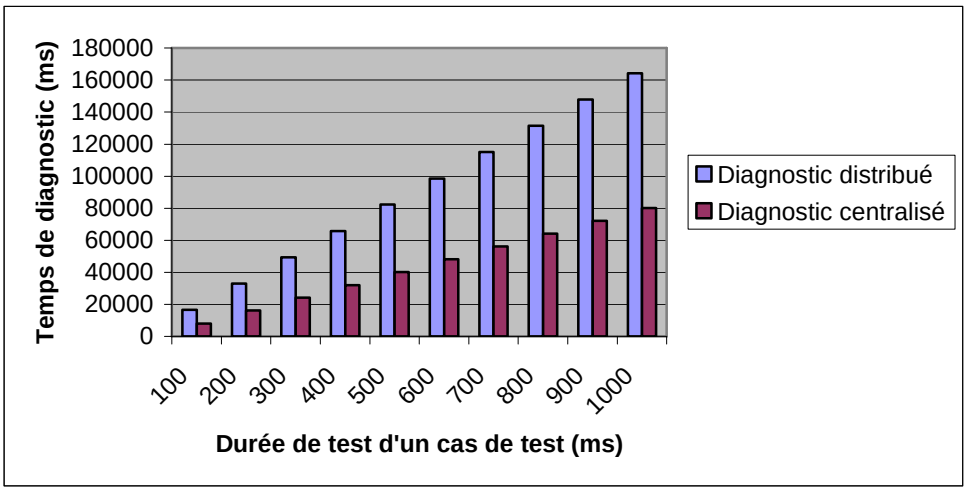


Figure 55. L'évolution du temps de diagnostic par rapport à la durée des tests

Le graphe de la Figure 55 nous permet de faire des observations suivantes :

- Dans le cas où l'application a un composant fautif, le temps d'exécution de la méthode de diagnostic distribué est presque deux fois plus grand que celui de la méthode de diagnostic centralisé.
- Le temps d'exécution de la méthode de diagnostic centralisé et la durée de test sont proportionnels.
- Le temps d'exécution de la méthode de diagnostic distribué et la durée de test sont aussi proportionnels.

5.2. Cas d'étude

Nous avons effectué un travail de portage du service de diagnostic DISCO sur une application développée dans le cadre d'une collaboration entre le laboratoire LIG et France Télécom. Ce travail nous a permis d'expérimenter concrètement le portage et la réutilisation du service DISCO sur un exemple d'application relativement complexe. Ce cas d'étude est un système de gestion à grande échelle de données de capteurs hétérogènes. Ce travail a donné lieu à une publication dans IEEE Depend'09 [BUI09b].

Le système de gestion de capteurs hétérogènes [GUR08] est un système qui permet de surveiller, à grande échelle et en temps réel, un environnement physique. La surveillance se

fait à l'aide de requêtes complexes évaluées sur des flux de données provenant de capteurs. Il permet de traiter également des aspects d'administration de parcs de capteurs hétérogènes.

5.2.1. Description du système

L'implémentation du système de gestion à grande échelle de données de capteurs hétérogènes (SStreaMWare) est faite avec le modèle de composants OSGi sur la plate-forme Oscar. La Figure 56 montre l'architecture de ce système qui comprend les composants suivants :

- Composant **S**ensor **Q**uery **S**ervice (SQS)
- Composant **G**ateway **Q**uery **S**ervice (GQS)
- Composant **P**roxy **Q**uery **S**ervice (PQS)
- Composant **L**ookup **S**ervice (LS)

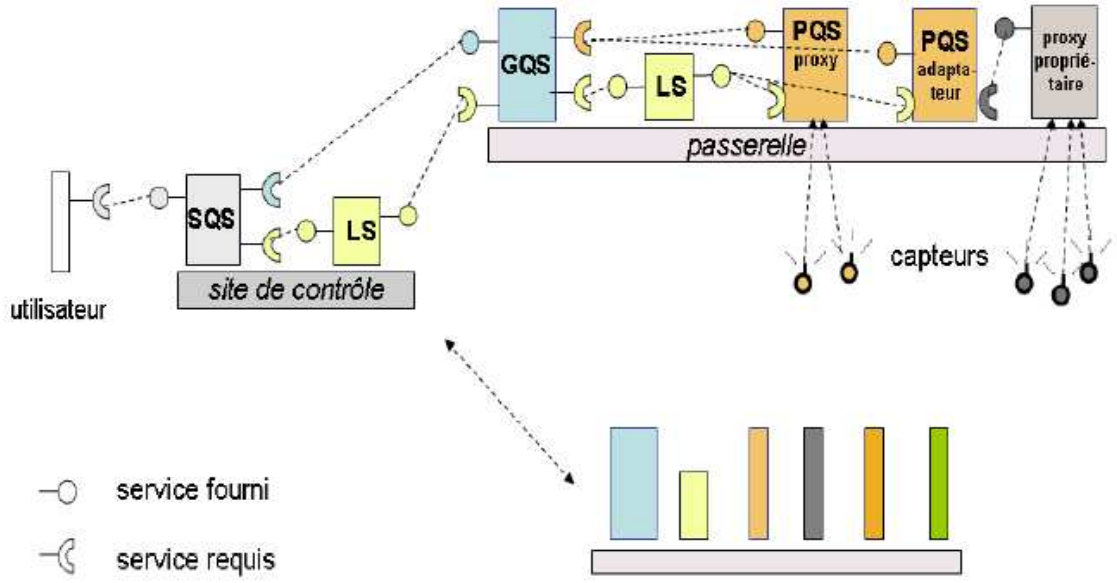


Figure 56. Architecture du système SStreaMWare [GUR08]

La description de ces composants est détaillée dans les sections suivantes.

5.2.1.1. Composant SQS

Un composant SQS fournit le service d'interrogation de tous les capteurs de l'environnement qu'il gère. Ces services sont déployés sur les sites de contrôle pour être proposés aux utilisateurs/applications. Ses responsabilités sont :

- Contribuer à l'évaluation des requêtes :
  - Recevoir les requêtes adressées à tous les capteurs dans le système.
  - Décomposer des requêtes et les envoyer aux GQSs concernées à l'aide des informations du composant Lookup Service.
  - Récupérer les résultats partiels provenant des GQSs et effectuer les traitements nécessaires d'évaluation des requêtes.

- Retourner les résultats sous forme de flux de données.

- Arrêter une requête continue.
- Fournir diverses informations sur les GQSs, PQSs et les capteurs présents dans le système.

Dans ce système, un composant SQS est implémenté comme un service distant accessible à partir d'une adresse IP et d'un numéro de port. La communication à distance utilise le protocole RMI (Remote Method Invocation) [RMI]. Le composant SQS implémente l'interface *SensorQueryService* qui étend l'interface *java.rmi.Remote*. Ce service est enregistré auprès d'un "RMI registry" qui permet aux clients de le découvrir et de l'invoquer.

5.2.1.2. Composant GQS

Ce composant fournit un service d'interrogation déployé sur chaque passerelle. Il est chargé d'évaluer les requêtes concernant la sous-région du parc des capteurs géré par la passerelle. Le composant SQS envoie les sous-requêtes aux composants GQSs concernés. Le composant GQS re-décompose les sous-requêtes afin de les envoyer aux composants PQSs concernés qui sont localisés grâce au composant LS exécuté sur la passerelle. Après l'évaluation de la requête par le GQS, les flux de résultats sont renvoyés au SQS.

Similairement au SQS, le composant GQS fournit diverses méthodes pour arrêter une requête, récupérer l'information sur les proxies présents sur la passerelle et l'information sur les capteurs, *etc.*

Un GQS implémente l'interface *GateWayQueryService* qui étend aussi l'interface *java.rmi.Remote* pour permettre au SQS de l'utiliser à distance.

5.2.1.3. Composant PQS

Le composant Proxy Query Service fournit un service d'interrogation offert par les proxies. Il joue le rôle d'interface entre les capteurs et le système. Il peut se comporter soit comme un proxy, soit comme un adaptateur dans le cas où le proxy est propriétaire. Il a la capacité d'évaluer des requêtes concernant les données de ses capteurs. Pour cela, il exporte une méthode d'interrogation qui reçoit en entrée une sous-requête et qui retourne les résultats sous forme de flux. Les PQSs cachent l'hétérogénéité des capteurs participant au système. Dès son entrée dans le système, un composant PQS est répertorié par le LS. Le GQS peut ainsi le découvrir et l'utiliser.

5.2.1.4. Composant LS

Le composant Lookup Service, noté LS, fournit un service de découverte. Ce service gère un catalogue contenant l'information sur les composants présents dans le système. Le catalogue peut être soit centralisé sur le site de contrôle, soit distribué dans le système. Le LS offre la possibilité de découvrir les services de manière dynamique. En d'autres termes, Le LS maintient une base de données des services qui sont présents dans le système, et fournit des moyens d'interroger et de mettre à jour cette base.

Ce système a utilisé deux services existants qui sont "Service Binder" [SER] et "Extended Service Binder" [BGL06]. Le "Service Binder" a été proposé pour l'environnement OSGi pour l'automatisation des liaisons de services. Il est chargé de résoudre, de manière automatique et durant l'exécution, les dépendances entre les services décrits dans un fichier de configuration.

Cependant, ce service est conçu pour la liaison de services appartenant à une seule passerelle, il ne prend pas en compte les services distribués sur plusieurs passerelles. Il y a une version étendue de ce service, à savoir "Extended Service Binder", développée au sein de France Telecom R&D. Avec ce service, la découverte et la liaison de services dépassent les frontières d'une seule passerelle pour permettre un fonctionnement dans un environnement distribué. ESB résout les dépendances entre les services, découvre les services nécessaires et les lie entre eux.

5.2.2. Diagnostic du système

Dans cette architecture de système de gestion de capteurs hétérogènes (Figure 56), nous proposons que chaque passerelle soit considérée comme un groupe de diagnostic qui comprend les composants suivants :

- Un GQS
- Les PQSs (TemperatureService, CameraService, GPSService, *etc.*)

Le rôle principal de GQS est de gérer les proxies dans la passerelle. En considérant les fonctions de ces composants, seul le diagnostic des composants PQSs est mis en place.

5.2.2.1. L'interface de test du composant PQS

Les composants PQSs fournissent un service qui s'appelle *ProxyQueryService*. Ce service se compose des fonctions principales suivantes :

- La méthode d'interrogation des capteurs : `evaluateQuery(QueryPlan qp, Receiver receiver)`.
  - qui prend en entrée un plan d'exécution d'une requête (conçu par le SQS) et la référence d'un récepteur (`Receiver`) et qui récupère les résultats en flux ;
  - qui retourne la référence de l'envoyeur `Sender` qui va envoyer les résultats en flux.
- La méthode `stopQuery(int QId)` qui arrête l'exécution de la requête spécifiée par l'identificateur donné en argument.
- La méthode `getSensorsInfo()` qui retourne l'information sur les capteurs dont le PQS a la charge.
- La méthode `updateSensor(Integer SId, String attr, Object nValue, int priority)` qui permet de modifier ou mettre à jour les informations des capteurs.

L'analyse de ces composants nous a permis de déduire le modèle d'états illustré sur la Figure 57. Deux états sont considérés : "*Active*" et "*Idle*". L'état "*Active*" correspond à l'état d'un composant ayant des requêtes en cours sur ses capteurs. L'état "*Idle*" correspond à l'état d'un composant n'ayant aucune requête en cours sur ses capteurs.

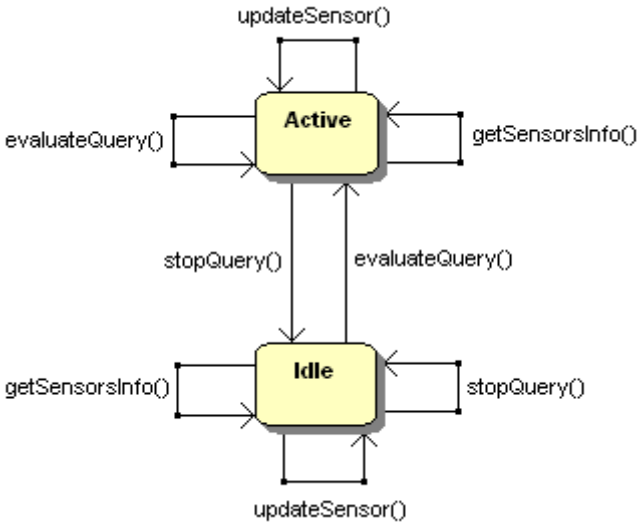


Figure 57. Modèle d'états des composants PQSs

A partir de ce modèle d'états, les cas de test qui correspondent à toutes les transactions possibles du modèle d'états du composant ont été générés et sont décrits dans le Tableau 13.

Tableau 13. Les cas de test du composant PQS

Cas de test	État initial	Entrées (données de test)	Méthode à tester	État final	Sorties à vérifier
1	Idle	Un queryPlan Un receiver	evaluateQuery(qp, re)	Active	Les tuples (les données générées par le capteur)
2	Active	Identificateur de la requête QId	stopQuery(QId)	Idle	La requête est arrêtée
3	Idle	QId	stopQuery(QId)	Idle	
4	Active	Un queryPlan Un receiver	evaluateQuery(qp, re)	Active	Les tuples (les données générées par le capteur)
5	Active		getSensorsInfo()	Active	Les informations des capteurs
6	Idle		getSensorsInfo()	Idle	Les informations des capteurs
7	Active	Sid : identifiant de capteur Attr : nom d'attribut nValue : nouvelle valeur priority	updateSensor(SId, attr, nValue, priority)	Active	Ok
8	Idle	Sid Attr nValue priority	updateSensor(SId, attr, nValue, priority)	Idle	OK

L'implémentation des deux opérations `isInState(State)` et `setToState(State)` est très simple (Figure 58).

setToState(State)

```
case: State in
Active:
evaluateQuery(qp, re)
Idle
stopAllQuery()
default: continue
```

isInState(State)

```
case: State in
Active: return
(active = = true)
Idle : return
(active = = false)
default: return false
```

Figure 58. Implémentation des opérations setToState et isInstate du composant PQS

5.2.2.2. Intercepteur de sorties

Les données de capteurs sont représentées par des tuples de données qui comprennent trois ensembles de données de nature différentes : les mesures des capteurs qui représentent les mesures faites par les capteurs, l'estampille de temps qui désigne le moment de la réalisation de la mesure et les propriétés des capteurs, par exemple leur identification, leur localisation, leur type, leur unité de mesure, *etc.*

La structure d'un exemple de requête est illustrée dans la Figure 59. On remarque que les tuples de données générés par les capteurs sont passés par les *receivers* et *senders* aux différents niveaux du système (PQS, GQS, SQS).

Pour les cas de test avec la méthode `evaluateQuery(QueryPlan qp, Receiver re)`, il est nécessaire d'intercepter les données générées par des capteurs. Pour cela, nous avons ajouté une méthode dans le *sender* au niveau de PQS.

Comme les données des capteurs sont générées périodiquement et en temps réel, on ne peut pas vérifier toutes les données produites par les capteurs. Une solution possible consiste à collecter les tuples de données générés pendant un intervalle spécifié, par exemple 2 minutes, pour exécuter un test. Une autre façon de procéder, consiste à collecter un nombre spécifié de tuples, par exemple 10 tuples, pour exécuter un test. C'est cette dernière option que nous avons utilisée dans nos expérimentations.

Les valeurs à vérifier sont les propriétés des capteurs (la localisation, le type de capteur et l'unité de mesure) et les mesures faites par les capteurs. Par exemple, on peut vérifier si la température fournie par un capteur de température est trop basse ou trop haute.

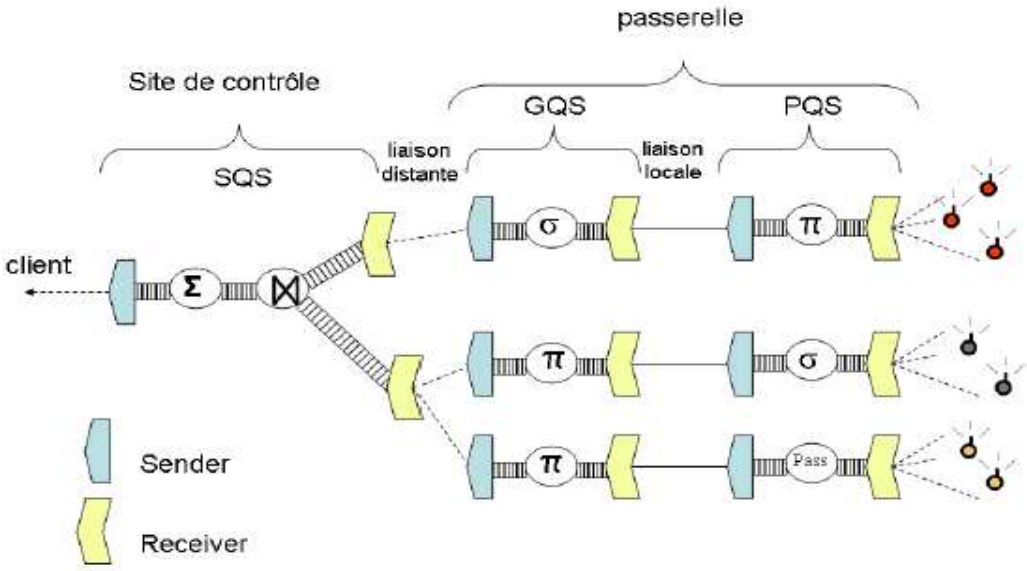


Figure 59. Structure d'une requête composée de plusieurs sous-requêtes chacune s'exécutant sur une entité différente [GUR08]

5.2.2.3. Contexte en ligne

Dans le cas de ce système de gestion de données de capteurs hétérogènes, les données sont envoyées par des capteurs périodiquement et en temps réel. C'est pourquoi la solution qui consiste à bloquer le composant pendant le processus de test est à éviter. La solution de clonage des composants est très gourmande en mémoire. Par conséquent, il nous semble que

l'utilisation d'une session de test ou l'abandon du processus de test sont les solutions les mieux adaptées.

- Pour les cas de test avec les méthodes `evaluateQuery(QueryPlan qp, Receiver re)` et `stopQuery(QId)`, on opte pour l'utilisation d'une session de test qui garantit que les données de test et les données réelles ne sont pas mélangées durant le processus de test.
  - `evaluateQuery(QueryPlan qp, Receiver re)` : l'interface de test permet de créer un `queryPlan` pour la requête et un `receiver` pour recevoir les données de capteurs. Donc, il n'y a pas d'interférence avec la fonctionnalité normale du composant.
  - `stopQuery(QId)` : on peut arrêter une requête créée pour le test sans influencer la fonctionnalité normale du composant.
- Le test de la méthode `getSensorsInfo()` n'a aucune influence sur la fonctionnalité normale du composant.
- Le test de la méthode `updateSensor(Integer SId, String attr, Object nValue, int priority)`nécessite la sauvegarde des informations du capteur avant le test, et leur restauration à la fin du test.
- Pour les cas de test qui nécessitent un état initial "*Idle*", c'est-à-dire pour amener le composant PQS à l'état "*Idle*", il faut arrêter toutes les requêtes traversant ce composant. Les requêtes créées pour le test n'influencent pas la fonctionnalité normale du composant. Mais dans le cas où il y a des requêtes pour le composant, le cas de test en cours est abandonné et le composant testeur poursuit avec les autres cas de test.

5.2.3. Les résultats obtenus

5.2.3.1. Surcoût en mémoire du diagnostic

Pour estimer le surcoût en mémoire du diagnostic, nous avons mesuré la mémoire physique occupée par le service DISCO, c'est-à-dire la mémoire de stockage sur le disque dur du service DISCO, ainsi que la mémoire utilisée durant l'exécution du service DISCO.

5.2.3.1.1. Mémoire physique occupée par le service DISCO

Les mesures de la mémoire physique nécessaire pour le service de diagnostic dans le système de gestion de données de capteurs sont présentées dans le Tableau 14 :

Tableau 14. Surcoût du service de diagnostic dans le système de gestion des capteurs.

Critères	**Surcoût de service de diagnostic**	**Surcoût dans chaque composant Dépend des cas de test (6 cas de test)**	**Application**	**% comparé avec l'application**
Taille des classes compilées	27.4 kB	9.2 kB	503.49 kB	~ 7.27 %
Taille des bundles	23.82 kB	4.51 kB	5.063 MB	~ 1.1%

Ces mesures nous permettent d'observer que le service de diagnostic n'est pas gourmand en mémoire de stockage.

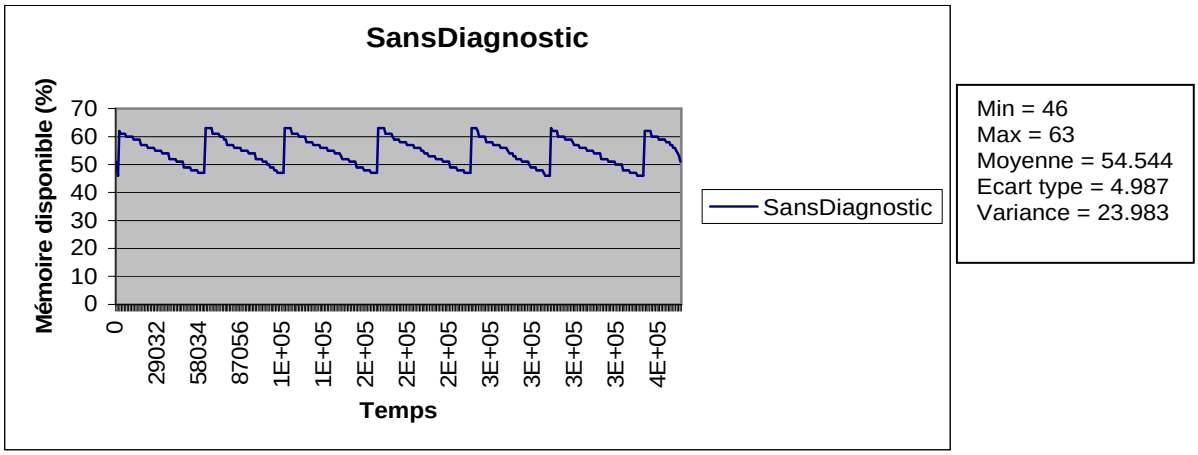
5.2.3.1.2. Mémoire utilisée durant l'exécution du service DISCO

Pour estimer la mémoire utilisée durant l'exécution du service DISCO, nous avons mesuré la mémoire disponible lors de l'exécution de l'application sans diagnostic et avec diagnostic.

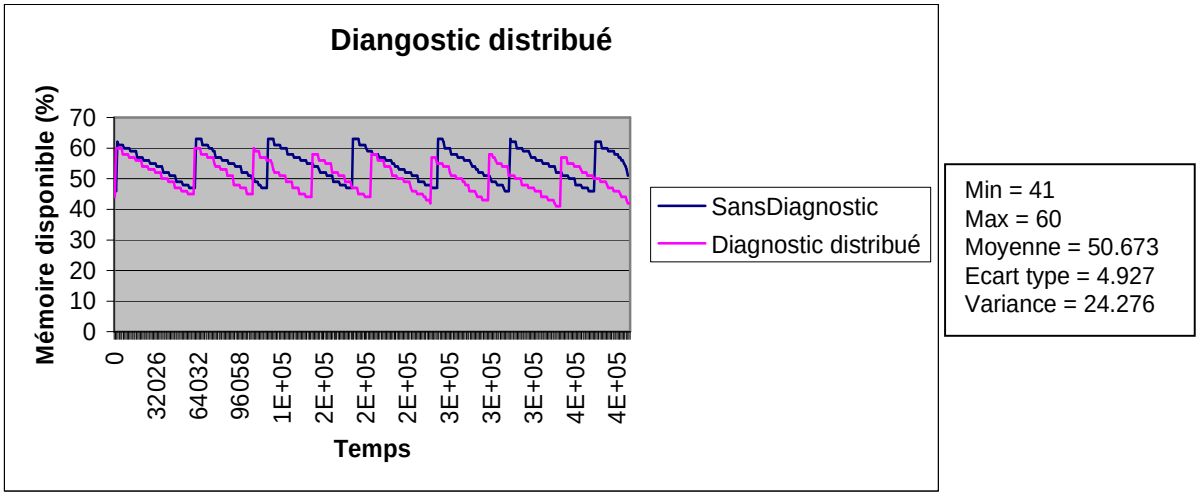
Le système de gestion de capteurs considéré pour ces expérimentations présente les caractéristiques suivantes :

- Il se compose de deux passerelles, avec 7 PQSs dans chaque passerelle. Chaque passerelle constitue un groupe de diagnostic. Dans chaque groupe, il y a un composant fautif.
- Période de diagnostic : 100000 ms.
- Utilisation de 6 cas de test : 1,2,3,4,5,6 (Tableau 13). Les cas de test avec la méthode `evaluateQuery()` testent 4 tuples de données générées par le capteur. Ce paramètre est identique pour toutes les expérimentations du système de gestion de capteurs.

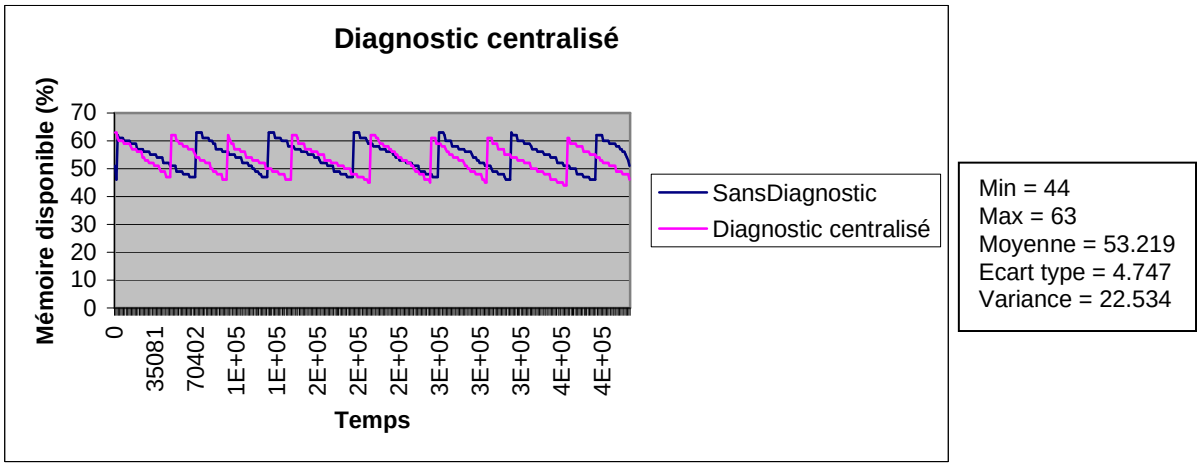
Pour simuler un composant PQS fautif, nous avons fait générer au composant capteur des valeurs de température erronées. Ce sont les cas de test 1 et 4 (dans le Tableau 13) qui peuvent détecter ce type de faute.



(a) Système sans diagnostic



(b) Système avec diagnostic distribué



(c) Système avec diagnostic centralisé

Figure 60. Mémoire disponible pour le système de gestion de données des capteurs

La Figure 60 montre l'évolution de la mémoire disponible de l'application sans diagnostic (a), avec diagnostic distribué (b) et avec diagnostic centralisé (c). La mémoire disponible est calculée comme étant la différence entre la mémoire utilisée par l'application et la mémoire totale dédiée à la Machine Virtuelle Java (~5M).

Les résultats obtenus, illustrés par la Figure 60, montrent que les surcoûts liés au service de diagnostic sont très faibles. En effet, la mémoire utilisée par le service est comprise entre 2 et 9 % de la mémoire totale.

5.2.3.2. Comparaison du diagnostic distribué et du diagnostic centralisé

5.2.3.2.1. Nombre de tests exécutés

La Figure 61 montre l'évolution du nombre de tests exécutés par le diagnostic centralisé et par le diagnostic distribué par rapport le nombre de composants fautifs pour le système de gestion de données de capteurs dans le cas de 14 composants. Les résultats présentés sont des moyennes obtenues pour 20 essais.

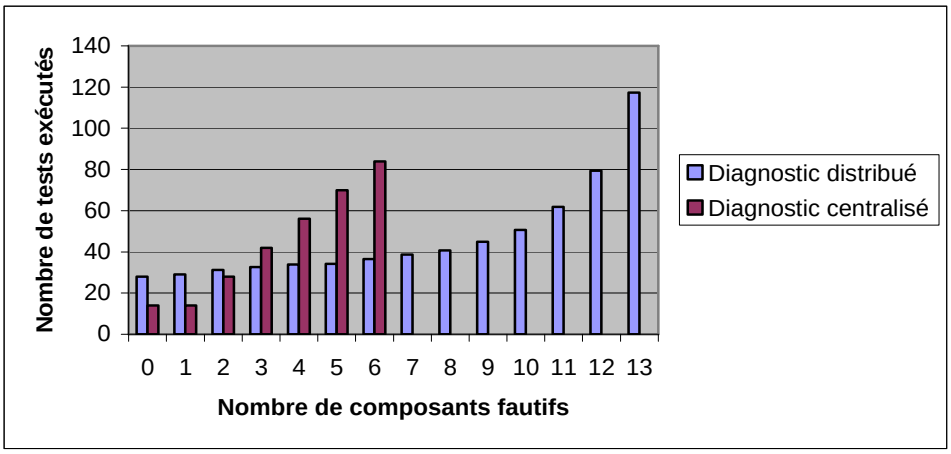


Figure 61. Nombre de tests exécutés par le diagnostic centralisé et le diagnostic distribué pour le système de gestion de données de capteurs

Ce résultat nous permet de confirmer les observations faites sur la plate-forme d'expérimentation, concernant la réduction du nombre de tests exécutés par le diagnostic distribué quand le système a plus de deux composants fautifs.

5.2.3.2.2. Evolution du temps de diagnostic en fonction du nombre de composants

Les conditions de cette expérimentation avec le système de gestion de données des capteurs sont :

- Le système comprend un groupe de diagnostic qui a 14 composants.
- Il y a un composant fautif.
- 6 cas de test sont considérés pour chaque composant.

La Figure 62 décrit l'évolution du temps d'exécution de l'algorithme de diagnostic centralisé et du diagnostic distribué en millisecondes par rapport au nombre de composants pour le système de gestion de données de capteurs.

Les résultats présentés sont des moyennes obtenues pour 20 essais.

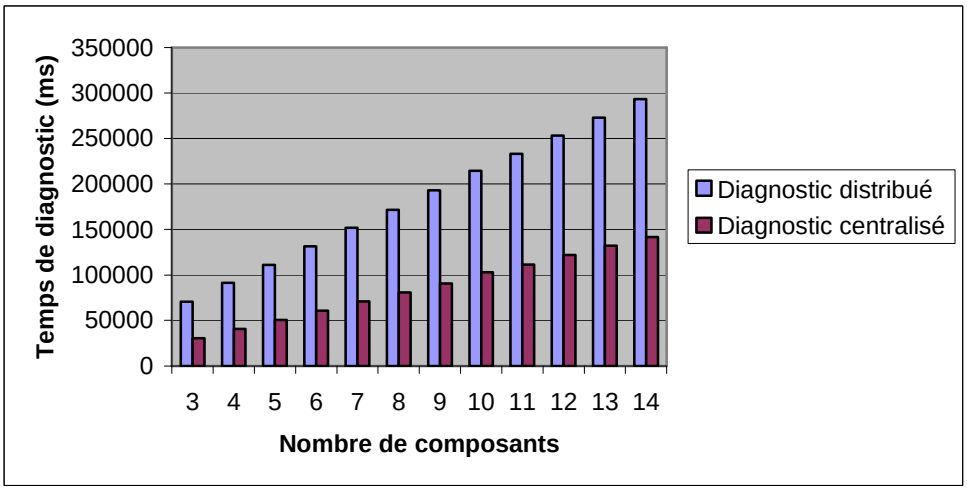


Figure 62. L'évolution du temps de diagnostic en fonction du nombre de composants pour le système de gestion de données de capteurs

Le graphe de la Figure 62 nous permet de confirmer les observations de la plate-forme d'expérimentation :

- Le temps d'exécution de la méthode de diagnostic distribué est plus important que celui de la méthode de diagnostic centralisé dans ce cas où le système a un composant fautif.
- Quand l'application a un composant fautif, le temps de diagnostic distribué est presque deux fois plus grand que celui de la méthode de diagnostic centralisé.

5.2.3.2.3. Evolution du temps de diagnostic en fonction du nombre de composants fautifs

Les conditions d'expérimentation pour le système de gestion de données des capteurs sont :

- Le système a 14 composants dans un seul groupe de diagnostic.
- 6 cas de test sont considérés pour chaque composant.

L'évolution du temps du diagnostic centralisé et du diagnostic distribué par rapport au nombre de composants fautifs pour le système de gestion de données de capteurs est décrite dans la Figure 63.

Les résultats présentés sont des moyennes obtenues pour 20 essais.

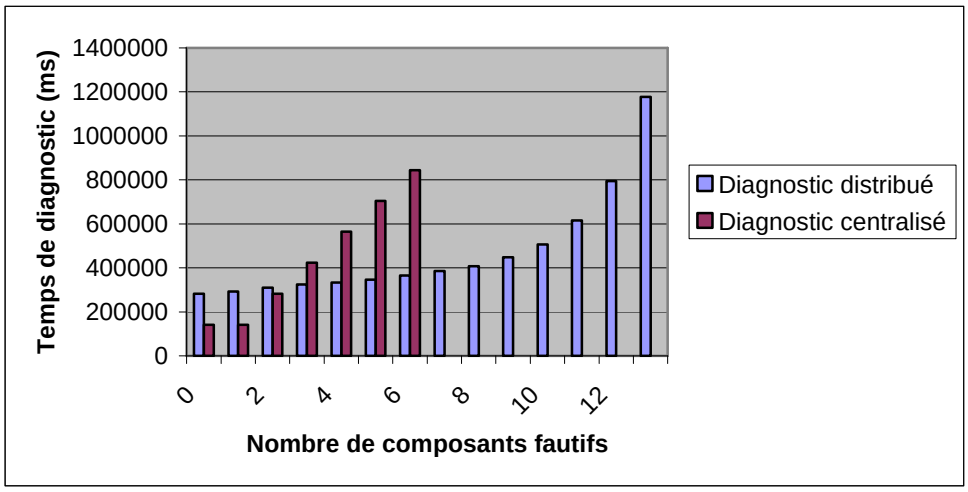


Figure 63. L'évolution du temps de diagnostic en fonction du nombre de composants fautifs pour le système de gestion de données de capteurs

Le graphe de la Figure 63 nous permet de confirmer les observations faites sur la plate-forme d'expérimentation :

- Quand le système a peu de composants fautifs (0, 1 ou 2 composants fautifs), le temps d'exécution de la méthode de diagnostic distribué est plus grand que celui de la méthode de diagnostic centralisé.
- Mais quand le système a plus de composants fautifs (plus de 2), le temps d'exécution du diagnostic distribué est plus petit que celui du diagnostic centralisé.

- Le temps d'exécution de la méthode de diagnostic centralisé est très sensible au nombre de fautes. Il présente une évolution quasi proportionnelle au nombre de composants fautifs.
- Le temps d'exécution de la méthode de diagnostic distribué est moins sensible au nombre de fautes.
- Le temps d'exécution de la méthode de diagnostic distribué est plus sensible au nombre de composants fautifs lorsque ce nombre est relativement élevé. En effet, le dernier cas correspond à un nombre de fautes égal au nombre de composants moins 1 (t=n-1). C'est le pire cas pour l'algorithme de diagnostic distribué.

5.3. Conclusion

Dans ce chapitre, nous avons présenté la validation expérimentale du service de diagnostic DISCO sur une plate-forme d'expérimentation implémentée avec la technologie de composants OSGi. Nous avons mesuré le surcoût dû à l'introduction du service de diagnostic DISCO et analysé les performances relatives du diagnostic centralisé et du diagnostic distribué. Le surcoût dû au diagnostic est globalement faible dans le cas des expérimentations présentées. Dans la littérature, des évaluations théoriques sont données pour le nombre de tests nécessaires pour les algorithmes choisis [PRE67] [BIA91]. Nous remarquons également que les résultats expérimentaux que nous avons obtenus confortent ces résultats théoriques. En particulier, la réduction du nombre de tests exécutés dans le cas du diagnostic distribué par rapport au diagnostic centralisé (à partir de plus de 2 composants fautifs). Par ailleurs, le diagnostic centralisé et le diagnostic distribué présentent des caractéristiques différentes, ce qui peut orienter le choix du concepteur pour l'une ou l'autre approche. Nous avons présenté aussi le portage du service de diagnostic DISCO sur un cas d'étude industriel – le système de gestion à grande échelle de capteurs hétérogènes. Sur ce cas d'étude, les résultats nous confirment que le surcoût dû au diagnostic est faible.

D'autres analyses expérimentales peuvent compléter celles présentées ici, en particulier en considérant des applications avec un nombre élevé de composants et en évaluant l'apport que peut avoir le partitionnement en groupes de diagnostic. Le portage sur d'autres cas industriels nous permettra d'éprouver la portabilité du service DISCO, notamment lorsque la procédure de génération des cas de test sera complètement automatisée.

Chapitre 6

CONCLUSION ET PERSPECTIVES

Les technologies à base de composants logiciels sont de plus en plus utilisées pour développer des systèmes complexes, et améliorer la composition, la réutilisation, la modularité et la configurabilité. Parallèlement, les logiciels actuels deviennent de plus en plus distribués et opèrent dans des environnements hautement dynamiques. La sûreté de fonctionnement de ces applications est donc indispensable pour permettre la livraison de services corrects aux utilisateurs du système. Dans ce contexte, cette thèse avait pour objectif global de contribuer à la tolérance aux fautes des applications à base de composants logiciels en utilisant des techniques de diagnostic, dont le principe est de mettre en place des tests inter-composants afin de détecter et localiser les composants défaillants. Dans ce chapitre, nous rappelons les objectifs précis de la thèse et nous faisons un bilan de nos contributions. Les principales perspectives de recherche en lien avec ces travaux sont également décrites.

6.1. Rappel des objectifs

Le principal objectif de ce travail est le développement d'un service de diagnostic pour des composants logiciels, en tenant compte des contraintes suivantes :

- **Propriété non-fonctionnelle** : le diagnostic de composants défaillants, qui fait l'objet de cette thèse, doit être traité comme une propriété non-fonctionnelle. Il doit considérer des critères de transparence, de séparation et composabilité, de visibilité, de réutilisabilité et de portabilité afin de décharger le développeur le plus possible de la mise en œuvre des mécanismes liés à la sûreté de fonctionnement. Notre solution est d'implémenter un service de diagnostic qui est le mieux adapté au contexte de nos travaux. En effet, cette approche permet d'ajouter élégamment des mécanismes de tolérance aux fautes à une application à base de composants logiciels, de façon portable et configurable. Cette approche offre également une très bonne séparation entre les mécanismes du diagnostic et l'application.
- **Flexibilité** : les composants logiciels s'exécutent dans des environnements complexes qui peuvent évoluer dynamiquement. Les applications construites à base de composants doivent s'adapter de façon autonome aux changements de

disponibilité des composants. Notre solution a pour but de proposer un service de diagnostic flexible qui tient compte des évolutions que subissent les applications.

- **Architecture générale :** notre but est de développer une solution de diagnostic générale, et indépendante le plus possible des applications et de leurs modèles de composants. En procédant à l'abstraction des éléments techniques et des concepts liés à chaque modèle de composant logiciel, il a été possible de définir une approche qui favorise la réutilisation et qui peut être exploitée dans différents contextes.

En considérant ces objectifs, le bilan des contributions de cette thèse est présenté dans la section suivante.

6.2. Bilan des contributions

La principale contribution de cette thèse a été de proposer un service de diagnostic comme solution au problème de la tolérance aux fautes dans les applications à base de composants logiciels. Il a fallu tout d'abord résoudre la problématique des tests inter-composants en ligne en vue de proposer des solutions générales, pouvant s'appliquer à tout modèle de composant logiciel. Par la suite, nous avons proposé une architecture pour le service de diagnostic DISCO en vue de son implémentation. Cette architecture est générale, et indépendante des applications et de leurs modèles de composants. En outre, elle est flexible et s'adapte à l'évolution dynamique des applications à base de composants. Enfin, pour valider nos travaux, nous avons procédé à des analyses et évaluations expérimentales du service DISCO sur différents modèles d'applications en vue de déterminer le surcoût dû au diagnostic et d'évaluer ses performances.

6.2.1. Tests inter-composants en ligne

Le service de diagnostic proposé fournit la possibilité de détecter et de localiser les composants fautifs durant l'exécution du système. Il se base sur des tests inter-composants en ligne. La mise en place de ces tests a nécessité la résolution des problèmes suivants :

- L'implémentation des tests : un composant est considéré comme une boîte noire avec des interfaces fournies et requises. Chaque interface fournie correspond à un ensemble d'opérations que le composant met à disposition d'autres composants. Chaque interface requise est un ensemble d'opérations dont le composant a besoin pour son exécution. D'une manière similaire, les fonctionnalités de test sont d'autres services que le composant fournit à son environnement. Le composant testeur peut donc utiliser cette interface de la même façon que les autres interfaces fonctionnelles. Nous intégrons une interface de test dans les composants pour fournir des fonctionnalités de test.
- Le contexte en ligne : les tests peuvent interférer avec la fonctionnalité normale du composant. Pour ce problème, nous avons passé en revue différentes solutions possibles. Il est important de noter qu'aucune solution n'est optimale et que le choix d'une solution plutôt qu'une autre dépend du contexte considéré.

Les travaux concernant les tests inter-composants ont donné lieu à deux publications : Quatita'07 [BUI07a] et ERCS'07 [BUI07b].

6.2.2. Service de diagnostic DISCO

Nous avons proposé une architecture pour le service de diagnostic DISCO en vue de son implémentation. Cette architecture est générale et indépendante des applications et de leurs modèles de composants. En outre, elle est flexible, s'adapte à l'évolution dynamique des applications à base de composants. L'architecture du service de diagnostic se compose des éléments principaux suivants : un site central qui gère tous les groupes de diagnostic ; dans chaque groupe, il y a un élément qui ordonnance tous les tests inter-composants de son groupe ; les algorithmes de diagnostic ; les moniteurs de ressources, et un collecteur qui journalise les erreurs. L'implémentation détaillée de ces composants, ainsi que la procédure de diagnostic sont présentées dans le chapitre 4.

Les travaux concernant cette architecture, les fonctions de base de notre service, les interfaces des composants et les interactions entre les composants ont été publiés et présentés dans les conférences internationales ACM EFTS'07 [BUI07c], IEEE SIES'08 [BUI08] et Qualita'09 [BUI09a].

6.2.3. Implémentation et validation

Les réalisations pratiques de cette thèse ont concerné l'implémentation d'une plate forme d'expérimentation du service DISCO sur le modèle de composant OSGi. Ce choix est motivé par les propriétés dynamiques du modèle de composant OSGi. En effet, la technologie de composant OSGi fournit un environnement d'exécution dynamique qui permet d'installer et désinstaller des composants à tout moment, et d'assembler des interfaces et des composants dynamiquement ; autrement dit, la structure des applications peut évoluer dynamiquement. Cette caractéristique permet une réalisation plus souple et flexible du service DISCO.

Cette plate-forme nous permet de modifier de manière souple les paramètres de diagnostic, par exemple la durée de test, le nombre de composants, *etc.* pour analyser les performances du service DISCO. Les résultats d'expérimentation obtenus sont présentés dans le chapitre 5. Nous avons procédé à des analyses et évaluations expérimentales du service DISCO en vue de déterminer le surcoût et d'évaluer les performances du diagnostic pour montrer son apport dans la tolérance aux fautes des applications à base de composants.

Ensuite, le service DISCO a été porté sur un cas d'étude industriel - le système de gestion à grande échelle de données de capteurs hétérogènes qui a été développé dans le cadre d'une collaboration entre le laboratoire LIG et France Télécom. Il est implémenté avec la technologie de composant OSGi. Ce travail a donné lieu à une publication dans IEEE Depend'09 [BUI09b].

6.3. Perspectives

Les travaux que nous avons présentés ouvrent la voie à plusieurs perspectives possibles. Ces perspectives sont principalement de deux natures : soit théorique, soit pratique.

Du point de vue théorique, trois études nous paraissent intéressantes à poursuivre : le développement d'un modèle de fautes adaptés au test en ligne de composants logiciels, le développement d'une approche de partitionnement qui tient compte des caractéristiques des applications sous test et enfin, en lien avec les deux premières perspectives, l'enrichissement des tests et du diagnostic pour les rendre encore mieux adaptés aux applications à base de composants logiciels.

Du point de vue pratique, il s'agit essentiellement d'automatiser la procédure de génération des cas de test pour simplifier l'utilisation du service DISCO. Enfin, pour pouvoir

intégrer le service DISCO dans une démarche complète de tolérance aux fautes, il serait intéressant d'étudier l'interfaçage du diagnostic et de la reconfiguration.

Ces différentes perspectives sont détaillées ci-dessous.

- **Modèle de fautes adapté au test en ligne des composants logiciels**. Selon la manifestation temporelle des fautes, on peut distinguer trois types de fautes [LAP96] :
  - Les fautes permanentes : ce sont des fautes qui se manifestent toujours quand elles sont exécutées.
  - Les fautes intermittentes : se manifestent occasionnellement et de manière très dépendante du matériel ou du logiciel.
  - Les fautes transitoires : sont des fautes temporaires dues à l'environnement du système.

  Une analyse intéressante consisterait alors à prendre en compte les fautes intermittentes et transitoires pour proposer des tests adaptés. Par ailleurs, l'étude du lien entre les défauts du matériel sous-jacent et les fautes au niveau des composants logiciels permettrait de proposer une approche de diagnostic globale, qui tient compte de l'ensemble du système (matériel et logiciel) et les interactions entre les composants dans le système.
- **Partitionnement**. Quand l'application comporte plusieurs composants, il est nécessaire de la diviser en groupes pour assurer l'extensibilité du service de diagnostic. En effet, cela permettra d'éviter un coût élevé de transmission des informations de test et de diagnostic. Le partitionnement en groupes de diagnostic est basé actuellement dans le service DISCO sur la localisation géographique des composants. Or, le partitionnement pourrait se baser sur d'autres critères plus adaptés, par exemple, le nombre de tests à exécuter, le temps de communication, *etc.*
- **Enrichissement des tests et du diagnostic**. Cette perspective découle automatiquement des deux perspectives précédentes. En effet, la prise en compte d'un modèle de fautes plus riche nécessite l'adaptation des tests. De même, la mise en place d'un partitionnement intelligent de l'application entraîne la nécessité d'adapter les algorithmes de diagnostic pour tenir compte des particularités de chaque groupe de diagnostic. L'étude de cette perspective pourrait aboutir à la proposition de nouveaux algorithmes de test et de diagnostic.
- **Automatisation de l'intégration de l'interface de test dans les composants.** Dans le service DISCO, avant le déploiement dans l'application, tous les composants doivent être analysés pour générer des machines d'états et des cas de test. Une interface de test qui comprend tous ces cas de test est ajoutée dans chaque composant. Une façon de diminuer le coût de développement est de générer automatiquement les cas de tests à partir de la spécification de l'application sous forme de modèle d'état. Dans la littérature, les travaux dans [OFF03] et [SEI08] mettent en place une génération de cas de test à partir d'une spécification d'application faite grâce à modèle d'états UML.
- **Interfacer le diagnostic avec un service de reconfiguration**. L'objectif ici est de mettre en place une approche de tolérance aux fautes complète basée sur le

diagnostic en ligne. Les solutions de reconfiguration existantes pourront être adaptées pour une utilisation conjointe avec le service DISCO.

- **Test inter-composant et diagnostic dans les systèmes matériels/logiciels.** Le test des systèmes matériels et sur puce se fait classiquement par le testeur externe ou par auto-test matériel. Cependant, les techniques reposant sur les équipements de test externes (*Automatic Test Equipment ATE*) se dirigent vers une impasse à cause de l'inaccessibilité des nœuds internes depuis l'extérieur de la puce et l'utilisation de mémoires de plus en plus larges. Face aux problèmes posés par le test depuis l'extérieur de la puce, une solution est d'intégrer davantage de fonctionnalités consacrées au test, fournissant ainsi des puces de plus en plus auto-testables. Cependant, les techniques d'auto-test matériel introduisent de la circuiterie supplémentaire dédiée au test, et ces circuits souffrent de ce matériel annexé.

  Avec l'avènement des systèmes sur puce (SoC) qui permettent exécuter du logiciel embarqué, une nouvelle technique de test appelée auto-test logiciel (*Software-Based Self-Test*) est étudiée. Il s'agit de réutiliser les capacités des SoC à exécuter du logiciel embarqué, à des fins d'auto-test. Il s'agit d'une approche non intrusive, de bas coût, flexible et programmable [GIZ04]. On peut citer les travaux d'auto-test présentés dans [GIZ08][KRA08][APO09]. En outre, dans le cas des composants matériels, il y a beaucoup plus de transistors défaillants pendant la durée de fonctionnement des puces que juste après leur production, donc, les auto-tests logiciels en ligne sont aussi étudiés comme dans [GIZ09][XEN07].

  Le travail présenté dans cette thèse s'appuie sur des tests inter-composants et le diagnostic en ligne pour des applications à base de composants logiciels. Il s'inspire en cela des travaux de diagnostic en ligne de composants matériels (processeurs) interconnectés par un réseau.

  L'analogie de nos travaux avec ceux relatifs à l'auto-test logiciel nous incite à analyser de plus près l'extension de notre approche aux cas des systèmes matériels/logiciels. Ceci suppose une analyse préalable des interactions entre les composants matériels et les composants logiciels d'un système (voir perspective ci-dessus sur le modèle de faute) ainsi que le développement d'une approche de diagnostic globale qui exploite les tests de composants logiciels ainsi que l'auto-test logiciel de composants matériels.

BIBLIOGRAPHIE

[ADE] http://www-adele.imag.fr/

[ADL] ADL Fractals, http://fractal.objectweb.org/tutorials/adl/index.html.

[AMM08] Ammann, P., Offutt, J., et Xu, W., "*Coverage Criteria for State Based Specifications*", Formal Methods and Testing, Springer, pp. 118-156, 2008.

[APO09] Apostolakis, A. *et al.*, "*Test Program Generation for Communication Peripherals in Processor-Based SoC Devices*", IEEE Design & Test of Computers 26(2), pp. 52-63, 2009.

[ARL00] Arlat, J., "*Composants logiciels et sûreté de fonctionnement*", Edition Lavoisier, 2000.

[ASP] AspectJ, http://www.eclipse.org/aspectj/.

[ATK02] Atkinson, C., et Groß, H.-G., "*Built-in contract testing in model-driven, component-based development*", ICSR Workshop on Component-Based Development Processes, 2002.

[BAC00] Bachman, F. *et al.*, "*Volume ii : Technical concepts of component-based software engineering*", Rapport technique ESC-TR-2000-007, Software Engineering Institute, Mai 2000.

[BAR02] Barwell, F., Blair R., Crossland, J., Case, R., Forgey, B. *et al.*, "*Professional VB .NET*", Number 1861007167, Worx Press, 2002.

[BAR93a] Barborak, M., Makek, M. et Dahbura, A., "*The consensus problem in fault-tolerant computing*", ACM Computing Surveys, Vol.25, No.2, pp. 171-220, June 1993.

[BAR93b] Barborak, M. et Malek, M., "*Partitioning for efficient consensus*", IEEE Hawaii Intl. Conf. on System Technology Sciences, pp. 438-446, 1993.

[BAS00] Bass, L. *et al.*, "*Market assessment of component-based software engineering*", Rapport technique CMU/SEI-2001-TN-007, Software Engineering Institute, Mai 2000.

[BAT00] Batista, T. et Rodriguez, N., "*Dynamic Reconfiguration of Component-Based Applications*", Proceedings of the international Symposium on Software Engineering For Parallel and Distributed Systems, IEEE Computer Society, Washington DC, Juin 2000.

[BAU01] Baudry, B., Hanh, V. L., Jezequel, J.-M. et Traon, Y. L., "*Trustable components: Yet another mutation-based approach*", W. E. Wong, editor, Mutation testing for the new century, pp. 47–54, Kluwer Academic Publishers, 2001.

[BCS02] Bruneton, E., Coupaye, T. et Stefani, J. B., "*The Fractal Composition Framework Version 1.0 "*, Object Web Consortium, Juillet 2002.

[BEI90] Beizer, B., "*Software Testing Techniques*", Van Nostrand Reinhold, 1990.

[BEN97] Benkahla, O., Aktouf, C. et Robach, C., "*Diagnostic des architectures parallèles : un état de l'art*", Technique et Science Informatiques, vol. 16, n°2, Février 1997, pp. 195-224.

[BEY03] Beydeda, S. et Gruhn, V., " *The Self-Testing COTS Components (STECC) Strategy – A new form of improving component testability"*, ACTA Press, pages 222-227, USA, 2003.

[BEY04] Beydeda, S., "*The Seft-Testing COTS Components (STECC) Method*", ISBN 3-89975-462-X, Martin Meidenbauer Verlag, München, 2004.

[BGL06] Bottaro, A., Gérodolle, A. et Lalanda, P., "*Pervasive spontaneous composition*", First IEEE International Workshop on Service Integration in Pervasive Environments, Lyon, France, Juin 2006.

[BIA91] Bianchini, R. et Buskens, R. W., "*An adaptative distributed system level diagnosis algorithm and its implementation*", International IEEE Symposium on Fault-Tolerant Computing, IEEE CS Press, pp. 616-626, Montréal, Canada, 1991.

[BIN00] Binder, R., "*Testing Object-Oriented Systems-Models, Patterns, and Tools*" Addison-Wesley, 2000.

[BOU05] Bouchenak, S., De Palma, N. et Krakowiak, S., "*Tolérance aux Fautes dans les Grappes d'Applications Internet*", French Chapter of the ACM-SIGOPS, 4ème Conférence Française sur les Systèmes d'Exploitation (CFSE-2005), Le Croisic, France, Avril 2005.

[BUI07a] Bui, T. Q. et Aktouf, O., "*Inter-component testing for system-level diagnosis of embedded component based applications*", Proceedings of the 7th Multidisciplinary International Conference on Quality and Reliability, pp. 246-254, Tanger, Marocco, Mars 2007.

[BUI07b] Bui, T.Q, et Aktouf, O., "*On-line Testing of Software Components for Diagnosis of Embedded Systems*", Proceeding of the 4th International Conference on Embedded and Real-Time Computing Systems, pp.330-336, Volume 22, Prague, Czech Republic, Juillet 2007.

[BUI07C] Bui, T.Q, et Aktouf, O., "*Diagnosis Service for Embedded Software Component based Systems*", Proceeding of the Second International Workshop on Engineering Fault Tolerant Systems, pp. 14-19, Dubrovnik, Croatia, Septembre 2007.

[BUI08] Bui, T.Q., Aktouf, O., et Dang, M., "*Software Component Diagnosis Service: Architecture Description*", Proceeding of the Third International Symposium on Industrial Embedded Systems, pp. 134-140 , La Grande Motte, France, Juin 2008.

[BUI09a] Bui, T.Q., Aktouf, O., D'orazio, L., et Dang, M., "*Service de Diagnostic pour Composants Logiciels*", acte de la 8ème édition du Congrès international pluridisciplinaire en Qualité et Sûreté de Fonctionnement, Besançon, France, Mars 2009.

[BUI09b] Bui, T.Q., Aktouf, O., Dang, M., Gürgen, L. et Roncancio, C., " *Diagnosis Service for Software Component and its Application to a Heterogeneous Sensor Data Management System*", Proceeding of the Second International Conference on Dependability, Athens, Grèce, Juin 2009.

[CAV96] Cavalli, A., Chin, B. M. et Chon, K., "*Testing methods for SDL systems*", Computer Networks and ISDN Systems 28(12), pp 1669-1683, 1996.

[CHO78] Chow, T., "*Testing software design modeled by finite-state machines*", IEEE Transactions on Software Engineering 4(3), pp 178-187, 1978.

[CLE] CLEOPATRE, http://cleopatre.rts-software.org/

[COA] COACH, http://coach.objectweb.org/.

[DEC] Projet DECOS, http://www.decos.at/.

[DEV01] Deveaux, D., Frison, P. et Jezequel, J.-M., "*Increase software trustability with self-testable classes in java*", Australian Software Engineering Conference (ASWEC), IEEE Computer Society Press, pp. 3–11, 2001.

[EJB] Sun Microsystems, "*Enterprise JavaBeans Technology*", http://java.sun.com/products/ejb/.

[FAV03] Favarim, F., Fraga, J. et Siqueira, F., "*Fault-tolerant CORBA Component*", IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 2003.

[FGUI] "*FractalGUI"*, http://fractal.objectweb.org/current/doc/javadoc/fractal-gui/overview-summary.html.

[FIL04] Filman, R. E., Elrad, T., Clarke, S. et Aksit, M., "*Aspect-Oriented Software Development*", Addison-Wesley, Octobre 2004.

[FRA] Fractal, http://sardes.inrialpes.fr/ecole/livre/pub/Chapters/Fractal/fractal.html

[FRA03] Fraga, J., Siqueira, F. et Favarim, F., "*An Adaptive Fault-Tolerant Component Model*", Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'03F), p. 179, 2003.

[GEF98] Geffroy, J.-C. et Motet, G., "*Sûreté de fonctionnement des systèmes informatiques*", Edition Dunod, 1998.

[GEN97] Gentleman, W. M., "*Effective Use of COTS (Commercial-Off-the-Shelf) Software Components in long Lived Systems*", Proceedings of International Conference on Software Engineering (ICSE'97), Boston, USA, Mai 1997.

[GIZ04] Gizopoulos, D., Paschalis, A. et Zorian, Y., "*Embedded Processor-Based Self-Test*", Springer, 2004.

[GIZ08] Gizopoulos, D. *et al.*, "*Systematic Software-Based Self-Test for Pipelined Processors*", IEEE Trans. VLSI Syst. 16(11), pp. 1441-1453, 2008.

[GIZ09] Gizopoulos, D., "*Online Periodic Self-Test Scheduling for Real-Time Processor-Based Systems Dependability Enhancement*", IEEE Trans. Dependable Sec. Comput. 6(2), pp. 152-158, 2009.

[GRO02] Groß, H.-G., "*Built-in Contrat Testing in Component-based Application Engineering*", CologNet Joint Workshop on Component-based Software Development and Implementation Technology for Computational Logic, Affiliated with LOPSTR, Madrid, Espagne, 19-20 Septembre 2002.

[GUR08] Gürgen, L., Labbé, C., Roncancio, C., Bottaro, A. et Olive, V., "*SStreaMware: a service oriented middleware for heterogeneous sensor data management*", Proceedings of the 5th International Conference on Pervasive Services, ACM Press, pp. 121-130, Sorrento, Italie, Juillet 2008.

[HAR87] Harel, D., "*Statecharts: A Visual Formalism For Complex Systems*", Science of Computer Programming, Vol 8-3, pp 231-274, Elsevier, 1987.

[HEI01] Heineman, G. T. et Council, W. T., "*Component-based software engineering: putting the pieces together*", Addison-Wesley Professional, Août 2001.

[HEJ03] Hejlsberg, A., "*The C# Programming Language*", Number 0321154916, Addison-Wesley Pub Co, 2003.

[HON00] Hong, H.S., Kim, Y.G., Cha, S.D. et Bae, D.H., "*A Test Sequence Selection Method for Statecharts*", J. STVR, 10(4), pp. 203-227, 2000.

[HOR02] Hörnstein, J. et Edler, H., "*Test reuse in cbse using built-in tests*", Workshop on component-based Software Engineering, Composing systems from components, 2002.

[HUA75] Huang, J. C., "*An approach to program testing*", ACM Computing Surveys 7(3), pp 113-128, 1975.

[ICM] iCMG, http://www.componentworld.nu/.

[IDL] IDL Syntax et Semantics, http://www.omg.org/cgi-bin/doc?formal/99-07-07.pdf

[J2EE] Sun Microsystems, "*Java 2 Platform, Enterprise Edition (J2EE)*", http://java.sun.com/j2ee/.

[JEZ01] Jezequel, J.-M., Deveaux, D. et Traon Y. L., "*Reliable objects: Lightweight testing for oo languages*", IEEE Software, pp. 76–83, 2001.

[KET04] Ketfi, A. et Belkhatir, N., "*A metamodel-based approach for the dynamic reconfiguration of component-based software*", The Eighth International Conference on Software Reuse, Madrid, Espagne, Juillet 2004.

[KNO] Knopflerfish, "*Open Source OSGi*", http://www.knopflerfish.org/.

[KOP02] Kopetz, H. et Bauer, G., "*The Time-Triggered Architecture*", Proceedings of the IEEE Int'l Conf. on Embedded Systems, Juin 2002.

[KRA08] Kranitis, N. *et al.*, "*Hybrid-SBST Methodology for Efficient Testing of Processor Cores* ", IEEE Design & Test of Computers 25(1), pp. 64-75, 2008.

[LAL01] Lala, P. K., "*Self-Checking and Fault-Tolerant Digital Design*", Morgan Kaufmann Publishers, USA, 2001.

[LAP96] Laprie, J. C, et al, "*Guide de la sûreté de fonctionnement* ", 2ème édition, Cépaduès Éditions, ISBN 2-85428-382-1, 1996.

[LEE94] Lee, S. et Shin, K. G., "*Probabilistic Diagnosis of Multiprocessor Systems*", ACM Computing Surveys, vol. 26, No. 1. 1994.

[LIB03] Liberty, J. et Hurwitz, D., "*Programming ASP.NET*", Number 0596004877, O"Reilly, 2003.

[MAR01] Martins, E., Toyota, C.M. et Yanagawa, R. L., "*Constructing Self-Testable Software Components*", Proceedings of the 2001 International Conference on Dependable Systems and Networks, p. 151-160, Göteborg, Suède, Juillet 2001.

[MAR02a] Marangozova, V. et Hagimont, D. "*An Infrastructure for CORBA Component Replication*", 1st IFIP/ACM Working Conference on Component Deployment, Berlin, Allemagne, Juin 2002.

[MAR02b] Marangozova, V. et Hagimont, D., "*Non-Functional Replication Management in the CORBA Component Model*", Proceedings of the 8th International Conference on Object-Oriented, Information Systems, 2002

[MEY79] Meyers, G., "*The Art of Software Testing*", Wiley, New York, 1979.

[MIL99] Miller, J., "*Estimating the Number of Remaining Defects after Inspection*", Softw. Test. Verif. Reliab. 9, 167–189 (1999).

[NET] Microsoft, ".*NET*", http://microsoft.com/net, 2002.

[OBJ99] Object Management Group, "*CORBA components : Joint revised submission*". Août 1999.

[OFF03] Offutt, J., Shaoying, L., Abdurazik, A. et Ammann, P., "*Generating Test Data from State-Based Specifications*", J. STVR, 13(1), 2003, pp. 25-53.

[OFF99] Offutt, J. et Abdurazik, A., "*Generating tests from UML specifications*", Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99), Fort Collins, CO, Octobre 1999.

[OSC] Oscar, "*An open source implementation of the Open Services Gateway Initiative (OSGi)*", http://forge.objectweb.org/projects/oscar/.

[OSGi] OSGi, "*Open Service Gateway initiative*", http://www.osgi.org/.

[PHA90] Phalippou, M. et Groz, R., "*Evaluation of an empirical approach for computer-aided test case generation*", 3rd Int. Workshop on Protocol Test Systems, pp 131-147, Etats-Unis, 1990.

[PIM79] Pimont, S. et Rault, J. C., "*An approach towards reliable software*", Proc. of the 4th Int. Conf. On Software Engineering, Allemagne, pp 220-230, 1979.

[PRE67] Prerapata, F. P., Metz. G. et Chien, R. T., "*On the connection assignment problem of diagnosticable system*", IEEE Transactions on Electronic Computers, vol. EC-16, n°6, p. 848-854, Décembre 1967.

[RMI] Java RMI (Remote Method Invocation), http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp/

[ROM02] Roman, E., Ambler, S. et Jewell, T., "*Mastering Enterprise JavaBeans*", Wiley Computer Publishing, second edition, 2002.

[RUN] Runtime Java Class, http://java.sun.com/j2se/1.4.2/docs/api/.

[SAB85] Sabnani, K. K. et Dahbura, A., "*A new technique for generating protocol tests*", Proc. of the 9th Data Communication Symposium, ACM Press, pp 36-43, Canada, 1985.

[SCE02] David Sceppa, "*Microsoft ADO.NET (Core References)*", MicrosoftPress, 2002.

[SEI08] Seifert, D., "*Test Case Generation from UML State Machine*", Technical report inria-00268864, Dedale, Lorraine, Laboratory of IT Research and its Applications (Loria), 2008.

[SER] Service Binder, "*Automatic service dependency management in OSGi*", http://gravity.sourceforge.net/servicebinder/.

[SIL06] Silly-Chetto, M. , "*Developing Embedded Applications with Selectable Components of the Cleopatre Free Open Source Library*", 2nd IEEE International Conference on Information & Communication Technologies: from Theory to Application, Damascus, Syria, Avril 2006.

[SMF] "*IBM Service Management Framework*", http://www-306.ibm.com/software/wireless/smf/.

[SUL06] Suliman, D., Paech, B., Borner, L., *et al.*, "*The MORABIT approach to runtime component testing*", Annual International Computer Software and Applications Conference, pp. 171-176, Chicago, 2006.

[SZY02] Szyperski, C., "*Component Software: Beyond Object-Oriented Programming*", Second edition, ACM Press, Component Software Series, Addison-Wesley, 2002.

[TRA99] Traon, Y. L., Deveaux, D. et Jezequel, J.-M., "*Self-testable components: from pragmatic tests to design-to-testability methodology*", Technology of Object-Oriented Languages and Systems, pages 96–107. IEEE Computer Society Press, 1999.

[URA92] Ural, H., "*Formal methods for test sequence generation*", Computer Communications 15(5), pp 311-325, 1992.

[VIN97] Vinoski, S., "*Corba : Integrating diverse applications within distributed heterogeneous environments*", IEEE Communications Magazine, Février 1997.

[VOA98] Voas, J. M., "*The Challenges of Using COTS Software in Component-Based Development*", IEEE Computer, Vol. 31, No. 6, pp. 44-45, Juin 1998.

[WAN00] Wang, Y. *et al.*, "*On Built-In Test Reuse in Object-Oriented Frame-work Design*", ACM Computing Surveys, 32(1), Mars, 2000.

[WTM] Windows Task Manager; http://www.microsoft.com.

[XEN07] Xenoulis, G., Psarakis, M., Gizopoulos, D. et Paschalis, A. M., "*On-Line Periodic Self-Testing of High-Speed Floating-Point Units in Microprocessors*", Twelfth International Conference on the Applications of Density Functional Theory, pp. 379-387, 2007.

[XML] W3C, "*Extensible Markup Language*", http://www.w3.org/ XML/.

ANNEXE A

Algorithmes de diagnostic utilisés

Cette annexe décrit les deux algorithmes de diagnostic adaptés de la littérature pour le cas des composants logiciels et actuellement implémentés dans le service DISCO.

A.1. Algorithme de diagnostic centralisé

Cet algorithme [PRE67] fait l'hypothèse de la présence simultanée d'un maximum de t composants défaillants durant une session de diagnostic. Afin d'assurer la détection et la localisation des t composants défaillants, il est nécessaire et suffisant que chaque composant soit testé par au moins t autres composants différents.

Chaque composant testeur transmet directement ses résultats de test à un composant central, supposé fiable. Ce composant central analyse alors l'ensemble des résultats de tests et détermine l'état des composants du système. L'ensemble des résultats de tests analysé est appelé syndrome.

L'implémentation de cette approche de diagnostic centralisé nécessite de déterminer précisément les relations de test entre les composants de manière statique. L'algorithme décrit sur la Figure 64 permet de réaliser cela.

```
//création d'une liste des relations de test pour un diagnostic centralisé
List[] testGraph ;
// Etablir des relations de test à partir de la liste des composants componentsList []
n = componentsList.size() ;
// n est le nombre de composants
for (int i=0; i<n; i++)
{
// Etablir les relations de test du composant i au composant i+1 (mod n), i+2 (mod n), i+3 (mod n), …, i+t (mod n).
      for (int j=1; j<=t; j++)
      {
            testRelation = (componentsList.get(i),
                              componentsList.get((i+j) mod n);
           testGraph.add(testRelation);
}
```

Figure 64. Algorithme de diagnostic centralisé

Soit l'exemple d'un système comportant 7 composants. On étudie le comportement du diagnostic pour les cas où l'on a t=1, t=2 et t=3.

Cas 1 : Système avec l'hypothèse d'un seul composant fautif (t=1)

Le graphe de test est comme suit :

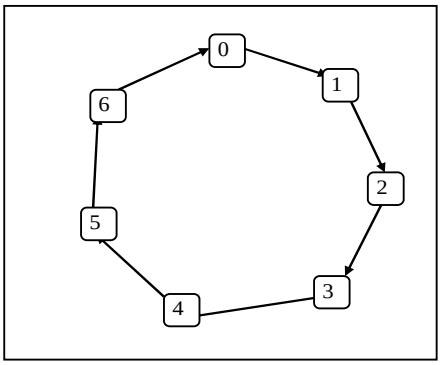


Figure 65. Le graphe de test avec t=1

Dans ce cas, chaque composant teste seulement un autre composant (Figure 65).

On va considérer toutes les configurations de fautes qui comportent un seul élément fautif.

Soit a_{ij} : le résultat du test du composant j par le composant i, i,j = 0,6.

Le résultat d'un test est binaire : faux (1), vrai (0). Il peut aussi être arbitraire (x) lorsque le composant testeur est fautif.

Composant fautif	a_{01}	a_{12}	a_{23}	a_{34}	a_{45}	a_{56}	a_{60}
Aucun	0	0	0	0	0	0	0
0	X	0	0	0	0	0	1
1	1	x	0	0	0	0	0
2	0	1	x	0	0	0	0
3	0	0	1	x	0	0	0
4	0	0	0	1	x	0	0
5	0	0	0	0	1	x	0
6	0	0	0	0	0	1	x

On remarque que deux défauts différents présentent des syndromes différents, c'est-à-dire que les syndromes associés aux configurations de fautes autorisées sont tous distincts. Il est donc toujours possible de déterminer sans ambiguïté la configuration de fautes présente, cela signifie qu'en analysant le syndrome, on peut trouver facilement le composant fautif. La Figure 66 montre l'algorithme d'analyse du syndrome pour trouver le composant fautif dans le cas t=1.

```
// testResults est la liste qui contient tous les résultats des tests selon l'ordre des relations de test dans la liste testGraph.
if (testResults !=null)
{
      int n = testResults.size();
      for (int i=0; i<n; i++)
      {
            if ((testResults.get(i%n)= = 0)
                  &(testResults.get((i+1)%n)= = 1))
            {
                 faultyComponent = componentsList.get((i+2)%n) );
            }
     }
}
```

Figure 66. Analyse du syndrome de l'algorithme de diagnostic centralisé

Cas 2 : t=2

Le graphe de test est comme suit :

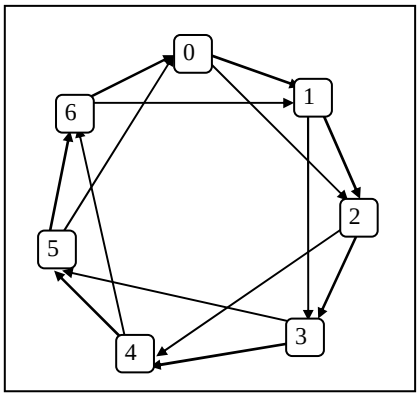


Figure 67. Le graphe de test avec t=2

Dans ce cas, selon l'algorithme de diagnostic centralisé, chaque composant doit être testé par 2 autres composants (Figure 67). Après avoir déterminé le syndrome, pour toutes les paires de composants fautifs, de la même façon que le cas 1, on peut trouver facilement les composants fautifs.

Cas 3 : t=3

De la même manière que les deux autres cas, on a le graphe de test suivant :

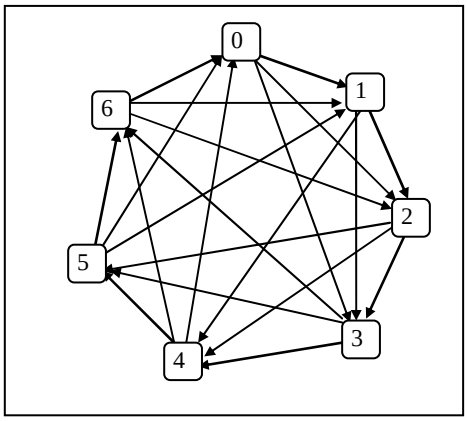


Figure 68. Le graphe de test avec t=3.

Chaque composant doit tester 3 autres composants (Figure 68), et on a des syndromes différents pour chaque triplet de composants fautifs. En analysant le syndrome obtenu, on peut trouver facilement les composants fautifs.

A.2. Algorithme de diagnostic distribué

L'algorithme distribué [BIA91] permet à tout composant de déterminer l'état du système à partir d'informations de tests produites localement et de celles transmises par d'autres composants.

L'idée principale de cet algorithme est qu'un composant n'accepte que les informations en provenance des composants qu'il teste et qu'il trouve corrects. Globalement, un composant opère de manière périodique selon le schéma suivant :

- le composant C_i teste le composant C_j comme étant correct,
- le composant C_i reçoit des résultats de tests transmis par le composant C_j,
- le composant C_i teste une nouvelle fois le composant C_j,
- le composant C_i suppose que les informations de test transmis par C_j sont valides si C_j est toujours correct.

(Les n composants sont numérotés C_0, C_1, C_2, …, C_{n-1}).

Cet algorithme de diagnostic distribué peut diagnostiquer jusqu'à n-1 composants fautifs simultanés, n étant le nombre de composants dans le système. Cet algorithme permet une amélioration considérable du nombre de tests par rapport au diagnostic centralisé.

Dans l'algorithme de diagnostic distribué, les structures de données maintenues au niveau d'un composant C_x sont un tableau $infoTest_x$ qui contient n éléments, indexés par l'identificateur de composant, i, comme $infoTest_x[i]$, ($0 \leq i \leq n-1$). $infoTest_x[i]=j$ indique que C_x a reçu l'information de diagnostic d'un composant correct indiquant que C_i a testé C_j et l'a trouvé correct.

$infoTest_2[0]=2$
$infoTest_2[1]=x$
$infoTest_2[2]=3$
$infoTest_2[3]=6$
$infoTest_2[4]=x$
$infoTest_2[5]=x$
$infoTest_2[6]=0$

Figure 69. Structure de données maintenue au niveau du composant C_2

La Figure 69 montre le tableau $infoTest_2$ maintenue au niveau du composant C_2 pour un système de 7 composants avec les composants C_1, C_4, C_5 fautifs. Notons que 'x' représente une valeur arbitraire (0 ou 1).

L'algorithme de diagnostic distribué fonctionne au niveau de chaque composant en recherchant d'abord un autre composant correct unique et en mettant à jour ensuite l'information de test locale avec l'information provenant de ce composant. En pratique, ceci est accompli comme suit. Tous les composants sont contenus dans une liste. Le composant C_x identifie le composant correct suivant dans la liste. Ceci est accompli par C_x en testant séquentiellement les composants consécutifs C_{x+1}, C_{x+2},..., jusqu'à ce qu'un composant correct soit trouvé. L'information de test reçue du composant correct est supposée valide et est utilisée pour mettre à jour l'information locale (Figure 70).

```
  /* Algorithme de diagnostic distribué */
/* exécuté au niveau de chaque composant Cx, 0 ≤ x ≤ n-1 */
   y=x ;
    result = false ;
   While { result == false)
         y=(y+1) mod n ;
          testRelation = (Cx , Cy) // c'est-à-dire Cx teste Cy
          result = execute(testRelation);
    }
    Cy forward infoTesty to Cx;
/* metre à jour l'information locale */
    infoTestx[x]=y;
    for i=0 to n-1
    {
          if (i ≠ x)
        {     infoTestx[i]= infoTesty[i];
          }
    }
```

Figure 70. Exécution des tests et transmission des résultats pour l'algorithme de diagnostic distribué

Par exemple, dans la Figure 71, le composant C_0 teste le composant C_1, le trouve fautif et continue à tester. Par la suite, le composant C_0 teste le composant C_2, le trouve correct et s'arrête de tester. Le composant C_2 trouve le composant C_3 correct et s'arrête de tester immédiatement. Le composant C_3 doit tester trois composants avant de trouver un composant correct. Le tableau $infoTest_2$ maintenu au niveau du composant C_2 pour cet exemple est représenté dans la Figure 69.

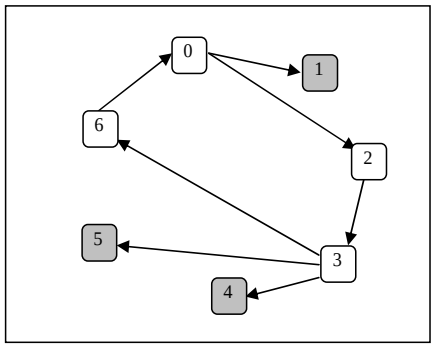


Figure 71. Un exemple de l'algorithme de diagnostic distribué

Le diagnostic est accompli par n'importe quel composant en suivant les chemins corrects de ce composant à d'autres composants corrects. L'algorithme de diagnostic à exécuter par un composant C_x est donné sur la Figure 72. L'algorithme utilise l'information stockée dans $infoTest_x$ pour déterminer l'état de faute des composants du système. Ses résultats sont stockés dans le tableau State,où State[i] est un élément de l'ensemble {correct, fautif}.

```
/*Diagnostic        */
/* exécuté par chaque composant Cx, 0 ≤ x ≤ n-1 */
for i=0 to n-1
 {
   State[i]=faulty ;
 }

ptr=x;
repeat {

        State[ptr]= fault-free ;
       ptr = infoTestx[ptr] ;

}until (ptr=x)
```

Figure 72. Détermination de l'état des composants dans l'algorithme de diagnostic distribué

ANNEXE B

Système de carte bancaire

La description des composants en langage de description d'interface IDL3 (Interface Description Language) [IDL] est la suivante :

// Composant *CodeServer*

```
component CodeServer{
     provides Validate the_validate;
     publishes CodeEvent to_client;
};
interface Validate {
     validatePin(Pin);
};
```

// Composant *ConsultServer*

```
component ConsultServer {
     provides Consult the_consult;
     consumes ChangeEvent the_change;
 };
interface Consult {
     long consultAccount(accountNumber);
};
```

// Composant *WithdrawServer*

```
component WithdrawServer {
     provides Withdraw the_withdraw;
     uses Consult the_consult;
     publishes ChangeEvent the_change;
};
interface Withdraw{
     boolean withdrawMoney(accountNumber, amount);
};
```

// Composant *DepositServer*

```
component DepositServer {
     provides Deposit the_deposit;
     uses Consult the_consult;
     publishes ChangeEvent the_change;
     };
interface Deposit
```

```
        {
             boolean depositMoney(accountNumber, amount);
        };
```

// Composant *Client*

```
        component Client {
              uses Validate the_validate;
              uses Consult the_consult;
              uses Withdraw the_withdraw;
              uses Deposit the_deposit;
              consumes CodeEvent from_CodeServer;
            };
```

ANNEXE C

Le système de climatisation

La description des composants de ce système et de leurs interfaces est décrite en IDL3 comme suit :

Composant thermomètre fournit une interface *getEnviTempsThermometer* pour transmettre à l'administrateur sa température.

```
//------------- Composant Thermomètre ----------------------
component Thermometer {
        provides GetEnviTempThermometer the_GetEnviTempThermometer;
  };
interface GetEnviTempThermometer
{
        long getTemp();
  };
```

Tout comme le composant thermomètre, le composant capteur fournit à l'administrateur la température de son thermostat.

```
//------------- Composant Capteur ---------------------
component Sensor {
        provides GetEnviTempThermostat the_GetEnviTempThermostat;
   };
interface GetEnviTempThermostat
{
        long getTemp();
  };
```

Le composant utilisateur est une interface graphique qui utilise les services fournis par le thermostat. Il utilise aussi un port de type événement provenant du thermostat pour informer l'utilisateur que la température choisie est trop haute ou trop basse.

```
//------------- Composant User ---------------------
  component User{
        uses SetUserTemp the_SetUserTemp;
        uses GetEnviTempThermostat the_GetEnviTempThermostat;
        uses UserStop the_UserStop;
        consumes TextEvent the_TextEvent;
   };
```

Le composant thermostat fournit les services qui permettent de choisir la température désirée, d'établir les valeurs minimale et maximale et d'arrêter le thermostat. Il utilise l'interface *getEnviTempThermostat* pour afficher la température actuelle dans la salle. Il utilise aussi les interfaces *Change* et *Stop* du composant Engine pour demander d'ajuster et d'arrêter la valve d'air chaud et la valve d'air froid.

```
//------------- Composant Thermostat ---------------------
component Thermostat {
        provides SetUserTemp the_SetUserTemp;
```

```
        provides UserStop the_UserStop;
        provides SetMinMaxTemp the_SetMinMaxTemp;
        uses GetEnviTempThermostat the_GetEnviTempThermostat;
        uses Change the_Change;
        uses stop the_Stop;
        publishes TextEvent the_TextEvent;
  };
interface SetUserTemp
{
        void setTemp(in long requestedTemp);
};
interface UserStop
{
        void stop();
  };
interface SetMinMaxTemp
{
        void setMinMax(in long max, in long min);
  };
 eventtype TextEvent
 {
        public string text;
  };
```

Le composant administrateur est une interface graphique qui utilise les services des autres composants pour permettre à l'administrateur de surveiller et contrôler le système. Il utilise aussi un port de type événement pour recevoir les informations que lui envoie le composant gestion.

```
//------------- Composant Administrateurr ----------------------
component Administrator{
        uses Change the_Change;
        uses Consult the_Consult;
        uses Delete the_Delete;
        uses Add the_Add;
        uses Modify the_Modify;
        uses Stop the_Stop;
        uses SetUserTemp the_SetUserTemp;
        uses SetMinMaxTemp the_SetMinMaxTemp;
        uses GetEnviTempThermostat the_GetEnviTempThermostat;
        uses GetEnviTempThermometer the_GetEnviTempThermometer;
        consumes TextEvent the_TextEvent;
  };
```

Le composant gestion se comporte comme une base de données et fournit les services de consultation, suppression, modification, ajout des thermostats et thermomètres.

```
//------------- Composant Gestion ----------------------
component Gestion {
        provides Consult the_Consult;
        provides Delete the_Delete;
        provides Add the_Add;
        provides Modify the_Modify;
        publishes TextEvent the_TextEvent;
  };
 interface Consult
```

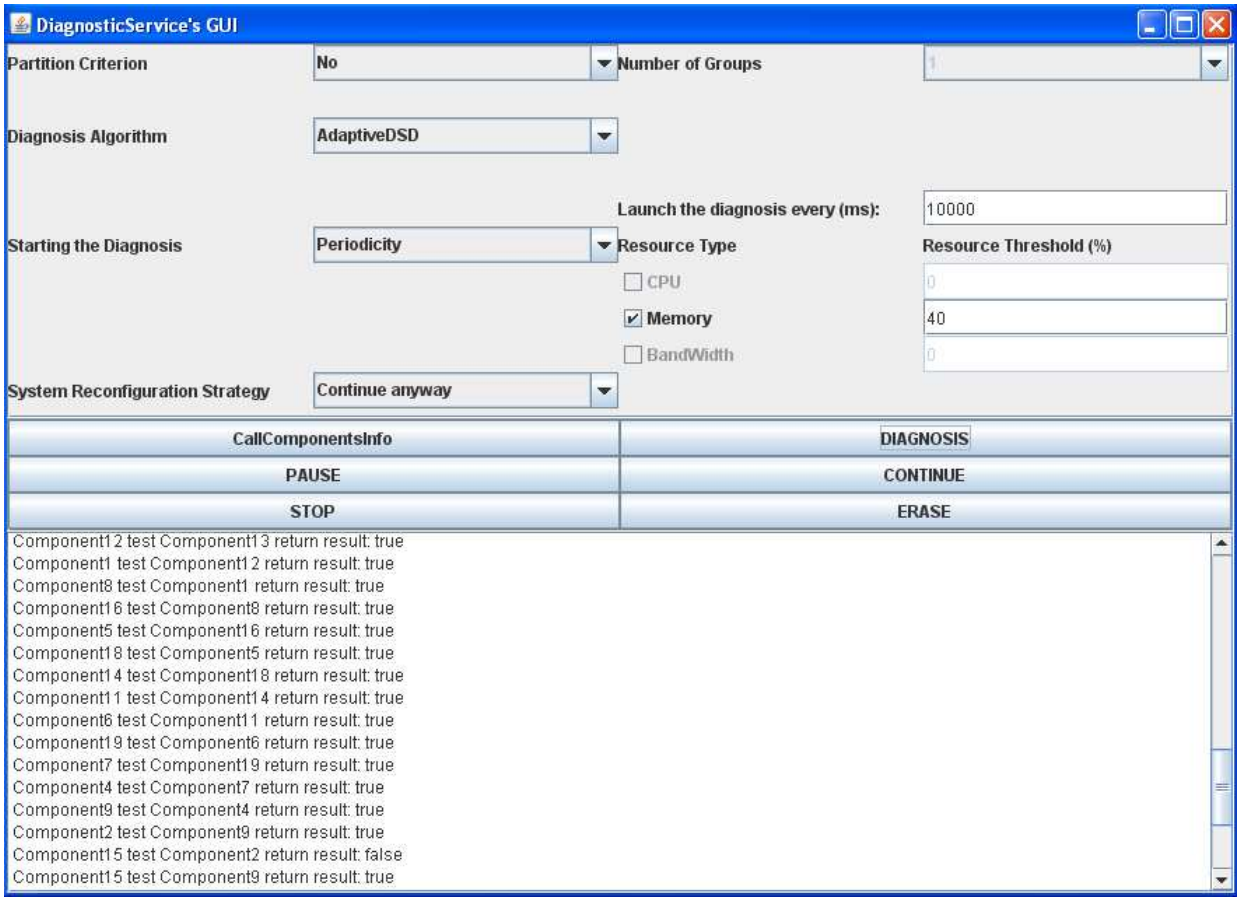
```
{
        station consultStation(in string name);
  };

  interface Delete
{
        void deleteStation(in string name);
  };

interface Add
{
        void addStation(in string name, in string location, in string
type);
  };

interface Modify
{
        void modifyStation(in string name, in string location, in
string type);
  };
```

Le composant EngineTemp fournit les interfaces pour ajuster ou arrêter la valve d'air chaud et la valve d'air froid.

```
//------------- Composant EngineTemp ----------------------

component EngineTempChange {
        provides Change the_Change;
        provides Stop the_Stop;

  };

interface Stop
{
        void stop(in string name);

  };

interface Change
{
        void change(in string name, in long delta);

};
```

ANNEXE D

L'interface graphique du composant DiagnosisCentral



Cette interface permet à l'utilisateur du service de diagnsotic de spécifier les paramètres du diagnostic, par exemple l'algorithme de diagnostic ; le mode d'exécution du diagnostic (une seule exécution ou plusieurs exécutions périodiques) ; les seuils de disponibilité des ressources permettant l'exécution des tests.

TABLE DES MATIERES

RESUME

L'informatisation croissante de divers domaines de la vie courante entraîne la nécessité de pouvoir placer une forte confiance dans les services délivrés par les systèmes informatiques. Un des rôles principaux de la sûreté de fonctionnement est l'intégration des méthodes et techniques destinées à conférer à un système l'aptitude à délivrer un service dans lequel on puisse avoir confiance, et à s'assurer que cette confiance est justifiée.

Les technologies à base de composants logiciels sont de plus en plus utilisées pour développer des systèmes complexes, et améliorer la composition, la réutilisation, la modularité et la configurabilité de ces systèmes. En parallèle, les logiciels actuels deviennent de plus en plus distribués et opèrent dans des environnements hautement dynamiques. La sûreté de fonctionnement des applications développées à base de composants logiciels devient alors un défi important.

Dans ce contexte, nous proposons un service de diagnostic DISCO (**DI**agnosis **S**ervice for **CO**mponent-based applications) qui augmente la fiabilité des applications à base de composants. Le principe de ce service est de mettre en place des tests inter-composants afin de permettre la détection et la localisation des composants défaillants. Les approches de test inter-composant et de diagnostic proposées utilisent des fonctionnalités particulièrement intéressantes car elles peuvent améliorer la sûreté de fonctionnement des applications avec un compromis compétitif entre les performances et les coûts. Les avantages de telles approches sont l'autonomie de l'application du point de vue de la sûreté de fonctionnement et la réduction des coûts liés à la tolérance aux fautes. Ces aspects sont très importants dans de nombreux systèmes actuels.

Dès que le processus de diagnostic est terminé, le système sera capable d'isoler les composants défaillants, en ignorant leurs sorties et en lançant une opération de reconfiguration afin d'assurer la fiabilité à long terme du système. Notre objectif est de proposer un service de diagnostic flexible et qui s'adapte aux évolutions des applications à base de composants. Pour assurer l'ouverture et la réutilisabilité de notre service, nous avons proposé une architecture générale qui est basée sur un modèle générique de composants et l'avons validé pour le modèle de composants OSGi.

Mots clés : sûreté de fonctionnement, diagnostic, test inter-composant, application à base de composants.

ABSTRACT

The increasing computerization of different fields in our society nowadays has led to the need for a strong confidence in services provided by computing systems. One of the great merits of the concept of dependability is the integration of methods and techniques to give a system the ability to deliver a service in which users have a confidence and to ensure that this confidence is justified.

Software component technology is used increasingly to develop complex software systems, for enhancing composition, reuse, modularity and configurability. Software systems are becoming more distributed and operate in highly dynamic environments. Therefore, the dependability of component-based applications is an important research issue.

In this context, we propose a diagnosis service that enhances the dependability of component-based applications. The principle of diagnosis techniques is to establish inter-component tests to detect and locate faulty components. The proposed inter-component and diagnosis approaches provide very interesting functionalities since they may enhance application dependability with a competitive cost-performance trade-off. The advantages of diagnosis are application autonomy and cost-effectiveness. Such advantages are very important for current systems.

Our approach is implemented as a diagnosis service DISCO (**DI**agnosis **S**ervice for **CO**mponent-based applications). This service takes the responsibility of detection and location of faulty components during runtime system execution. Once the diagnosis process is completed, the system is able to isolate the diagnosed faulty components, by ignoring their output and initiating a reconfiguration operation such that the reliability of the system can be maintained in the long run. Our goal is to provide a flexible diagnosis service that adapts to the evolutions of component-based applications. To ensure the openness and the reusability of our service, we propose a general architecture that is based on a general model of components and implement it with the OSGi models.

Keywords: dependability, diagnosis, inter-component tests, component-based applications.