INSA RENNES

Institut National des Sciences Appliquées
Ecole publique d'ingénieurs

TELELOGOS

Manage Yourself

Rapport Final

Projet de 4ème année informatique

Equipe :

Etienne Alibert,
Florian Barbedette,
Pierre Chesneau,
Mathias Deshayes,
Sevan Hartunians,
Mathieu Poignet.

Encadrant :
Laurence Rozé

Le 26/05/2010

Le présent rapport est le rapport final du projet ManageYourself réalisé dans le cadre de la 4ème année au département informatique de l'INSA de Rennes.

Il a pour but de développer l'avancement final du projet et de décrire les modifications apportées par rapport aux phases précédentes du projet.

Ce rapport détaillera tout d'abord les rectificatifs apportés au projet par rapport aux phases de spécification et de conception. Il présentera ensuite un manuel utilisateur des différentes parties du logiciel, avant d'exposer le compte rendu des phases de test. Enfin il décrira l'état final du projet.

Sommaire

I. Modifications apportées aux spécifications et à la conception

A. Système Expert

i. Spécifications

Les spécifications du système expert n'ont pas subi de modifications importantes lors de sa réalisation. Ainsi les seules modifications apportées ont été :

L'ajout de la prise en charge des chaines de caractères en tant que valeur des variables dans les conditions

Ceci pour permettre à notre système expert de prendre en compte des variables telles des noms d'applications ou de fichiers. La spécification originale utilisait pour résoudre ce problème des booléens, mais cela aurait alourdi les règles plus que nous ne pouvions l'accepter.

Le changement d'appel des opérations correctives

Initialement prévues pour être des dll, nous somme revenus sur ce choix et avons implémenté les opérations correctives sous formes d'exécutables indépendants les un des autres. Cela permet à l'administrateur une plus grande liberté pour la création des actions correctives (Le langage de développement est maintenant indépendant de celui de notre application) Et cela évite, en cas de bug dans le correctif, d'entrainer le plantage complet du système expert.

L'autonomie du système expert

Le système expert n'est plus autonome, mais nécessite d'être appelé à intervalles réguliers par l'applicatif de surveillance. Cela pour éviter de devoir fixer l'emplacement des rapports à plusieurs endroits. De cette manière a chaque nouveau rapport, l'applicatif de surveillance appelle le système expert en lui passant en paramètre l'emplacement du dernier rapport.

L'architecture logicielle du système expert n'a pas non plus subi de modification importante, où tout du moins ces modifications ont été relativement simples à apporter.

ii. Conception

Les changements dus aux modifications des spécifications ont été très simples à effectuer : l'ajout de la prise en charge des variables de type texte n'ont nécessité que de rajouter des classes étendant la classe Valeur (Notre spécification comportant plusieurs types de variables, nous avions choisi de concevoir un système capable de facilement accueillir de nouveaux types de variables).

Le changement de méthode pour appeler une action corrective à été simple. En effet nous n'avons plus qu'à sauvegarder qu'un seul élément relatif à l'action corrective (son emplacement), contrairement à l'ancienne version qui nécessitait la connaissance de l'emplacement de la dll et de son point d'entrée.

La modification la plus importante de la conception a été le changement de méthode pour les comparaisons entre valeurs : En effet lors de la conception, nous avions choisi de redéfinir les opérateurs de comparaison pour les classes contenant les valeurs. Cette solution ne s'est pas avérée viable et nous avons choisi d'implémenter les interfaces IEquatable et IComparable. Cela a simplifié le développement de ces classes et a permis de rendre ces classes compatibles avec les structures de données génériques de C#.

B. Module de Compilation mrf vers C#

Notre système se basant sur Javacc, il a suffit à chaque fois de faire correspondre la grammaire avec les modifications (prise en compte par exemple des guillemets pour les chaines de caractères) puis de créer / modifier les règles de productions correspondantes.

Outre les modifications de la grammaire (demandées pour la prise en charge de nouveaux types de variables), il n'y a eu aucune autre modification sur ce module. Celui-ci comme prévu attend en entrée le fichier de règle créé par l'interface administrateur et génère en sortie le code C# correspondant.

C. Module de Reporting

Dans les rapports de conceptions précédents, nous avions fait l'hypothèse que ManageYourselfReporting.exe serait une application console qui ne devait pas apparaître dans le gestionnaire des taches. Une fois la réalisation un peu plus avancée, nous nous sommes aperçus que cette approche n'était pas très satisfaisante. L'utilisateur ne pouvait pas couper la surveillance sans passer par un autre logiciel qui servirait à couper les processus console. En effet sous Windows mobile, le gestionnaire de tâche ne fait apparaître que les applications qui possèdent une fenêtre graphique. Pour ces raisons, nous avons créé une fenêtre graphique qui accompagne notre exécutable et permet de l'arrêter à tout moment.

L'attribut activeapplication était à l'origine un attribut qui contenait la chaine rendue par la méthode statique Systemstate.ActiveApplication() qui était du type « dernière application lancée <- application lancée ». Après un entretien avec l'employé de TELELOGOS qui suit notre projet, il est apparu plus intéressant pour la partie apprentissage du projet de pouvoir connaître l'ensemble des applications lancées sur le mobile et plus uniquement les 2 dernières actives. C'est pourquoi nous avons incorporé à la partie surveillance du projet une classe « Process » qui est fournie par Microsoft et permet de faire appel à de nombreuses méthodes utilitaires concernant la gestion de processus. Un appel à la méthode « getProcesses() » permet de récupérer l'ensemble des processus actifs sur la machine. C'est le résultat de cet appel qui est inséré dans les rapports grâce au champ ActiveApplication.

Cette modification a induit des changements relativement importants dans la génération des fichiers « arff » nécessaires à l'apprentissage. Le module d'apprentissage ne peut pas gérer les chaînes de caractères. Il est donc nécessaire de créer un attribut booléen par application présente. En effet, dans l'entête des fichiers « arff » les attributs doivent êtres tous définis au préalable. C'est pourquoi nous avons donc modifié l'application de concaténation des rapports dans le but de pouvoir créer dynamiquement les entêtes des fichiers « arff » en fonction des applications lancées sur le mobile.

D. Communication via MediaContact

Le rapatriement des rapports du PDA vers le serveur se fait via un processus qui déplace les fichiers binaires du dossier OUT de la station vers le dossier IN du serveur. Ce rapatriement est nécessaire afin de réaliser l'apprentissage

Le programme d'apprentissage prenant en entrée un seul fichier « .arff », il faut transformer les rapports récupérés (écrits en binaire) en une ligne d'un fichier « .arff ». Ces fichiers devront ensuite être concaténés dans un fichier « apprentissage.arff » contenant un entête énumérant les différents attributs utilisés dans le fichier « .arff » utile pour l'apprentissage. Cet entête est généré automatiquement à partir des attributs disponibles dans les rapports à analyser.

Pour la concaténation des rapports reçus, nous avions initialement prévu de le faire via MediaContact. En effet, ce logiciel permet l'exécution de commandes système Windows sur le serveur et donc, via la commande « type », il aurait été possible de concaténer le contenu d'un répertoire avec cet entête. Cependant, chaque station ayant son propre dossier sur le serveur, il faut parcourir une arborescence de répertoire, ce qui n'est pas possible via l'emploi d'une commande système.

Nous avons donc, dans ce but réalisé un programme Java. Cette application a trois fonctions:

- elle permet de transformer les fichiers binaires en fichier « .arff »
- elle concatène le contenu de chaque sous-dossier (dossiers IN relatifs au PDA) au fichier « apprentissage.arff » utilisé en entrée du module d'apprentissage
- elle supprime les fichiers « .arff » et « .bin » contenus dans ces sous-dossiers (en effet ces fichiers causeraient des redondances de données et donc fausserait l'apprentissage)

La transformation des fichiers binaires en fichiers « .arff » se fait via l'appel d'un exécutable « binToArff.exe » qui prend en paramètre le fichier binaire et le nom du fichier « .arff » contenant les données au format « .arff ». Ce programme décode le binaire du rapport et le transforme en une ligne du fichier « .arff ». Cet exécutable est appelé sur tous les fichiers binaires récupérés par MediaContact.

Notre première idée était de parcourir les sous dossiers de manière récursive. Cependant l'appel d'un programme dans une fonction récursive produisait des résultats difficilement prévisibles. Certains fichiers étaient créés et pas d'autres. De plus la pile de la JVM saturait souvent en traitant de nombreux rapports. Nous avons donc décidé de coder ce parcours de répertoires de façon itérative, en effet l'arborescence de fichiers à parcourir est fixe de par la structure utilisée dans le logiciel MediaContact.

Nous avons donc produit un jar qui permet la transformation des fichiers binaires en « .arff », concatène ces fichiers « .arff » au fichier initial contenant la liste des attributs présents (nécessitée du format « .arff ») et supprime le contenu des rapports déjà concaténés au fichier « apprentissage.arff ».

Nous avons créé un processus permettant de lancer l'apprentissage. Ce processus se compose de 2 tâches. Ces 2 tâches sont des appels systèmes sur le serveur (fonction présente dans le logiciel MediaContact), chaque tâche correspond à l'exécution d'un jar.

La première tâche lance le jar nommé apprentissage.jar situé dans le dossier /jar du répertoire /Data de MediaContact. Cet exécutable prend en paramètre le fichier « apprentissage.arff » qui est construit suivant la méthode expliquée précédemment. Cet exécutable réalise l'apprentissage à proprement parlé en utilisant la bibliothèque Weka suivant la méthode présentée dans la partie relative à l'apprentissage. Il prend en paramètre un identifiant (dans notre cas « apprentissage »), représentant à la fois un fichier « .arff » et un fichier « .att » reprenant la liste des attributs. Le fichier de sortie est un fichier « .weka », ce fichier weka n'étant pas directement interprétable par un administrateur, il faut ensuite le transformer via lectureRegle.

La deuxième tâche lance le jar lectureRegle.jar situé dans le dossier jar sur le fichier weka généré par « apprentissage.jar ». Le résultat est un fichier « .appr » qui contient les règles apprises lisible dans un langage naturel.

II. Manuels Utilisateur

A. Système Expert et module de compilation

Le système expert n'a pas vocation à être appelé directement par l'utilisateur final, nous ne détaillerons donc que l'aspect administrateur.

Le Système expert est fortement lié au module de compilation présent sur le serveur.

i. Module de compilation mrf vers C#

Utilisation

Le module de compilation coté serveur se présente sous forme d'un fichier Jar ou de plusieurs fichier .class (java). Dans le cas du jar, l'appel s'effectue ainsi :

Java –jar <nom fichier jar> header.cs fichier.mrf footer.cs resultat.cs

L'appel aux fichiers class quant à lui s'effectue de la manière suivante :

Java mrfCompiler header.cs fichier.mrf footer.cs resultat.cs

Le Module de compilation coté serveur prend donc en entrée 3 éléments :

- Un fichier de header (header.cs).
- Un fichier mrf (règles .mrf)
- Un fichier de footer (footer.cs)
- Un nom de fichier résultat (resultat.cs)

Le fichier « header.cs » contient tous les éléments qui seront copiés (Bit à bit) dans le fichier généré par la compilation. Le contenu de ce fichier sera placé avant la partie générée par la compilation à proprement parler. Dans le cas d'une utilisation normale, cela permet de mettre dans

le fichier divers éléments supplémentaires tels des déclarations de classes, des inclusions, des déclarations de variables, etc.

Le fichier « .mrf » contient les règles qui seront converties en leur équivalent C# (chaque règle au format SI … ALORS …. FSI sera converti en code C# créant les objets et relations entre les objets nécessaires.

Le fichier « footer.cs » est l'équivalent du fichier header hormis qu'il se situe à la fin du fichier généré.

Le fichier « resultat.cs » est le nom du fichier produit par le module de compilation. Il n'y a aucune contrainte sur le nom de ce fichier, même s'il est conseillé de le nommer avec une extension en .cs.

La génération suppose qu'à l'endroit où elle se situe elle dispose d'un objet nommé kb de type `KnowledgeDatabase` **et** que le fichier courant contient les lignes :

```
using System;
using System.Collections.Generic;
using System.Text;
using InferenceEngine;
```

Si ces lignes ne sont pas présentes ou que l'objet kb n'est pas accessible, le code généré par l'applicatif serveur ne sera **pas** correct !

Lors de la compilation, la validité du fichier « .mrf » est contrôlée. Si une erreur survient, la compilation s'interrompt et un message d'erreur est affiché.

Pour être valide, le fichier « .mrf » doit respecter la grammaire indiquée ci-dessous :

Grammaire des fichiers « .mrf » (Manageyourself Rule File) :

FichierMrf	→ ident DeclFaits DeclRegles
DeclFaits	→ DeclFait*
DeclFait	→ « ATTRIBUT » ident TypeIdent
TypeIdent	→ « NUMERIC »| « STRING »| Enumere
Enumere	→ « {» ident (« , » ident)* « } »
DeclRegles	→ (« SI » Cond (« ET » cond)* « ALORS » Consequence (« ET » Consequence)* « FSI »)+
Cond	→ ident Op Operande
Operande	→ ident | nombre | string
Op	→ « = »| « <=»| « >=» | « < » | « > »| « != »
Consequence	→ « APPEL » ident | ident Op Valeur
Valeur	→ ident |nombre| string

Il existe 3 principaux types d'erreurs :

- l'erreur de type IO :

Cette erreur survient si le programme ne trouve pas ou ne peut pas accéder aux fichiers passés en paramètres. Assurez-vous de l'existence des fichiers et contrôlez les droits de l'application.

- les erreurs lexicales ou syntaxiques :

Ces erreurs affichent toutes un message contenant approximativement la ligne et le caractère au niveau duquel l'erreur à été détectée. Assurez-vous que votre fichier « .mrf » respecte bien la syntaxe.

- les erreurs sémantiques :

Ces erreurs indiquent la présence de faits non instanciés à un endroit inattendu, ou d'une valeur à un emplacement inattendu. Vérifiez la cohérence de vos règles.

Le code généré peut ensuite être compilé en utilisant la ligne de commande suivante :

%SystemRoot%\Microsoft.NET\Framework\v3.5\Csc.exe /noconfig /nowarn:1701,1702 /nostdlib+ /errorreport:prompt /warn:4 /define:TRACE;Smartphone /reference:".\InferenceEngine.dll" /reference:"%DOSSIER%\Microsoft.NET\SDK\CompactFramework\v3.5\WindowsCE\mscorlib.dll" /reference:"%DOSSIER%\Microsoft.NET\SDK\CompactFramework\v3.5\WindowsCE\System.Core.dll" /reference:"%DOSSIER%\Microsoft.NET\SDK\CompactFramework\v3.5\WindowsCE\System.Data.DataSetExtensions.dll" /reference:"%DOSSIER%\Microsoft.NET\SDK\CompactFramework\v3.5\WindowsCE\System.Data.dll" /reference:"%DOSSIER%\Microsoft.NET\SDK\CompactFramework\v3.5\WindowsCE\System.dll" /filealign:512 /optimize+ /out:MainAppInferenceEngine.exe /target:exe resultat.cs AssemblyInfo.cs

Pour fonctionner correctement, cette commande doit être effectuée dans un dossier comportant :

- le fichier généré par le module précédent
- le fichier « InferenceEngine.dll » (moteur d'inférence en lui-même)
- le fichier « AssemblyInfo.cs » (comportant les informations relatives à l'assembly généré)

Modifications

Les sources du module de compilation sont fournies sous forme d'un projet Eclipse (http://www.eclipse.org). Le projet en lui-même est un projet Javacc. Pour ouvrir correctement le projet, Eclipse doit disposer du plugin Javacc que vous pouvez obtenir ici : http://eclipse-javacc.sourceforge.net/

Le module de compilation est constitué de 2 fichiers principaux :

- Le fichier « mrfCompiler.jj » : c'est à partir de ce fichier qu'es construit le programme de reconnaissance (compilation) en java. C'est ce fichier qu'il faut modifier pour changer la grammaire des fichiers « .mrf ».
- Le fichier « outputFile.java » contient toute la partie dédiée à la génération de code. C'est ce fichier qu'il faut modifier si vous souhaitez modifier le code source généré.

ii. Système Expert

Utilisation

Le système expert se base sur un objet de type KnowledgeDatabase. Cet objet présente plusieurs fonctions permettant d'ajouter/modifier/supprimer des règles ou des faits.

Ces fonctions sont documentées dans le code source, nous ne détaillerons donc pas ici leur utilisation.

Modification

Pour ajouter de nouveau types de variables, vous devez ajouter une nouvelle classe au projet InferenceEngine.dll. Cette classe doit étendre la classe Value et implémenter les interfaces Iequatable et Icomparable.

Pour ajouter de nouvelles actions, vous devez ajouter une classe étendant la classe Action du projet.

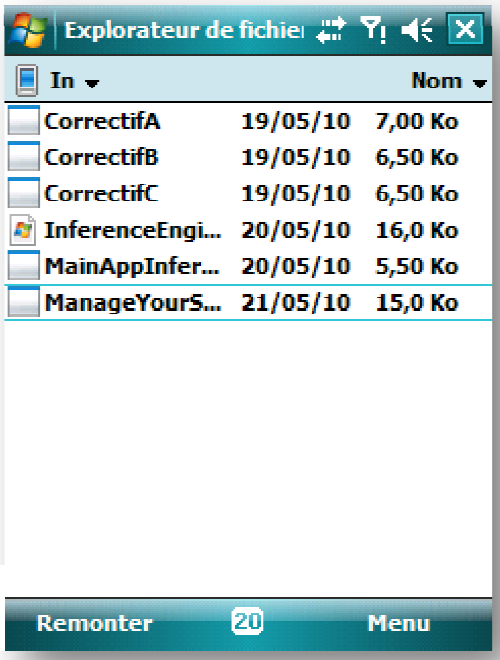
B. Module de Reporting

Nous considérerons pour la suite de ce manuel que l'arborescence correcte des répertoires à été créé par le logiciel MediaContact et que les différents exécutables nécessaires au bon fonctionnement du système sont également présent.

i. Mise en place du système :

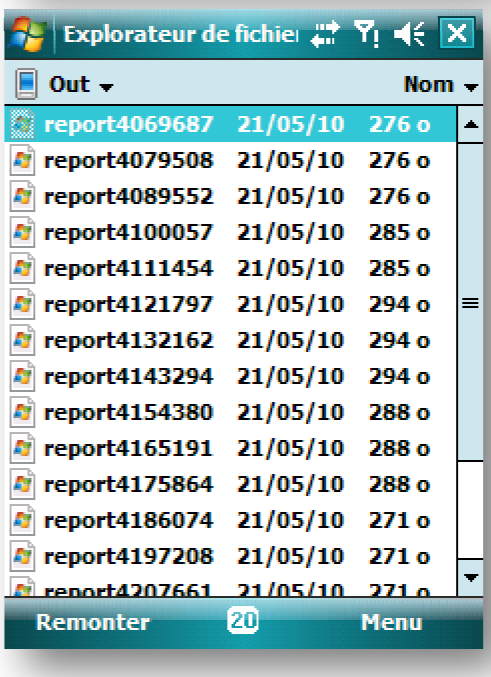
Une fois l'installation de MediaContact sur le mobile terminée, l'utilisateur trouvera dans \MediaContact\MCC\MediaContact\Data\In le contenu suivant : (la figure 1 présente le contenu de ce dossier)

- « ManageyourselfReporting.exe » : le logiciel de surveillance des paramètres système et de création de rapports.
- « MainAppInferenceEngine.exe » : les faits compilés utilisés par le système expert
- « InferenceEngine.DLL » : le système d'inférence utilisé par l'application MainAppInferenceEngine.exe
- l'ensemble des correctifs qui seront placés dans ce répertoire de manière automatisé par média contact. (À l'initialisation aucun correctif ne sera présent)



ontenu du répertoire « In » après
de médiaContact

Le répertoire « \MediaContact\MCC\ MediaContact\Data\Out » sert à l'enregistrement des rapports créés par « ManageYourselfReporting.exe ». Ce dossier sera vide à l'initialisation et MediaContact utilise cet emplacement pour rapatrier les rapports vers le serveur. La figure 2 présente un exemple de contenu de ce dossier après le lancement de l'outil de Reporting.



ire « Out » après
lfReporting.exe

ii. Lancement de l'outil de reporting :

Le lancement de manageyourselfReporting.exe va faire apparaître l'écran de la figure 3.

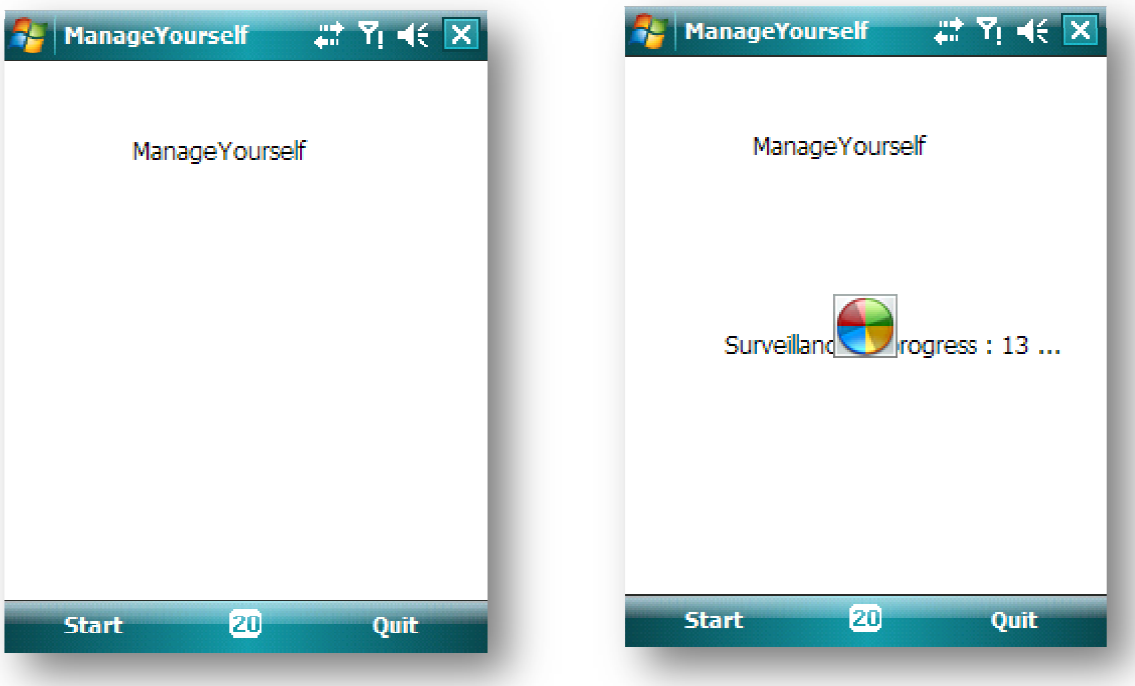


Figure 3 : écran de démarrage de l'application

Une fois le bouton « Start » enclenché, l'affichage va se modifier et indiquer que la surveillance est en cours, un léger ralentissement peut être visible ainsi que l'apparition d'un indicateur de surcharge processeur. Une fois cette phase en cours, le logiciel produit des rapports de façon périodique et transparente pour l'utilisateur.

L'utilisateur peut appuyer à n'importe quel moment sur le bouton « Quit » pour arrêter la surveillance.

C. Communication via MediaContact

i. Le processus RecupConcatRapport:

Il permet de récupérer tous les rapports des dossiers à un intervalle de temps à définir via MediaContact. Une fois ces rapports récupérés, il appelle un programme concatenation.jar situé dans le dossier « /jar ». Ce programme transforme les fichiers binaires contenus dans le dossier spécifié en paramètre en fichiers « .arff » (interprétable par « apprentissage.jar »), en utilisant le programme « BinToArff.exe » (situé aussi dans /jar), il concatène ces fichiers dans le fichier « apprentissage.arff » contenu dans le dossier « OUT » du serveur, et supprime les anciens fichiers concaténés « .bin » et « .arff ».

« Concatenation.jar » prend en paramètre le nom du fichier auquel concaténer tous les fichiers « .arff » contenus dans le répertoire passé en 2ème paramètre.

ii. Le processus ExécutionApprentissage:

Ce processus permet de réaliser le chainage des actions nécessaires à l'apprentissage.

La première tâche prend en paramètre « apprentissage », paramètre représentant à la fois le fichier « .arff » et un fichier « .att » reprenant la liste des attributs contenus dans le « .arff » (même syntaxe que l'entête arff mais sans les « @ »). La sortie de cette première tâche est un fichier « .weka » non interprétable par un administrateur. La deuxième tâche est exécutée sur le résultat (« .weka ») de l'apprentissage et produit un fichier « .appr » contenant les règles apprises directement lisibles.

iii. Le processus EnvoiSEInitial

Ce processus est le processus à lancer lors du premier envoi du système expert à une station distante.

Il copie les fichiers ManageYourSelfReporting.exe, InferenceEngine.dll, MainAppInferenceEngine.exe du dossier C:\MediaContact\MCS\MediaContact\Data\IN\ vers le dossier In des stations distantes.

Ces fichiers sont nécessaires pour le premier lancement du Système Expert sur téléphone (ManageYourselfReporting.exe).

iv. Le processus EnvoiSEMaj

Ce processus est le processus à lancer lors d'une mise à jour du système expert. Il permet de transformer le fichier mrf généré par l'interface administrateur en exécutable qui est ensuite envoyé sur les stations clientes

Il exécute le script mrfToExe situé dans le dossier C:\MediaContact\MCS\MediaContact\Data\IN\ qui permet cette transformation du format mrf vers un fichier exécutable par le téléphone.

Il copie le fichier MainAppInferenceEngine (le moteur d'inférence) de C:\MediaContact\MCS\MediaContact\Data\IN\ vers le dossier Station In

D. Module d'apprentissage et interface administration

i. Décomposition en packages

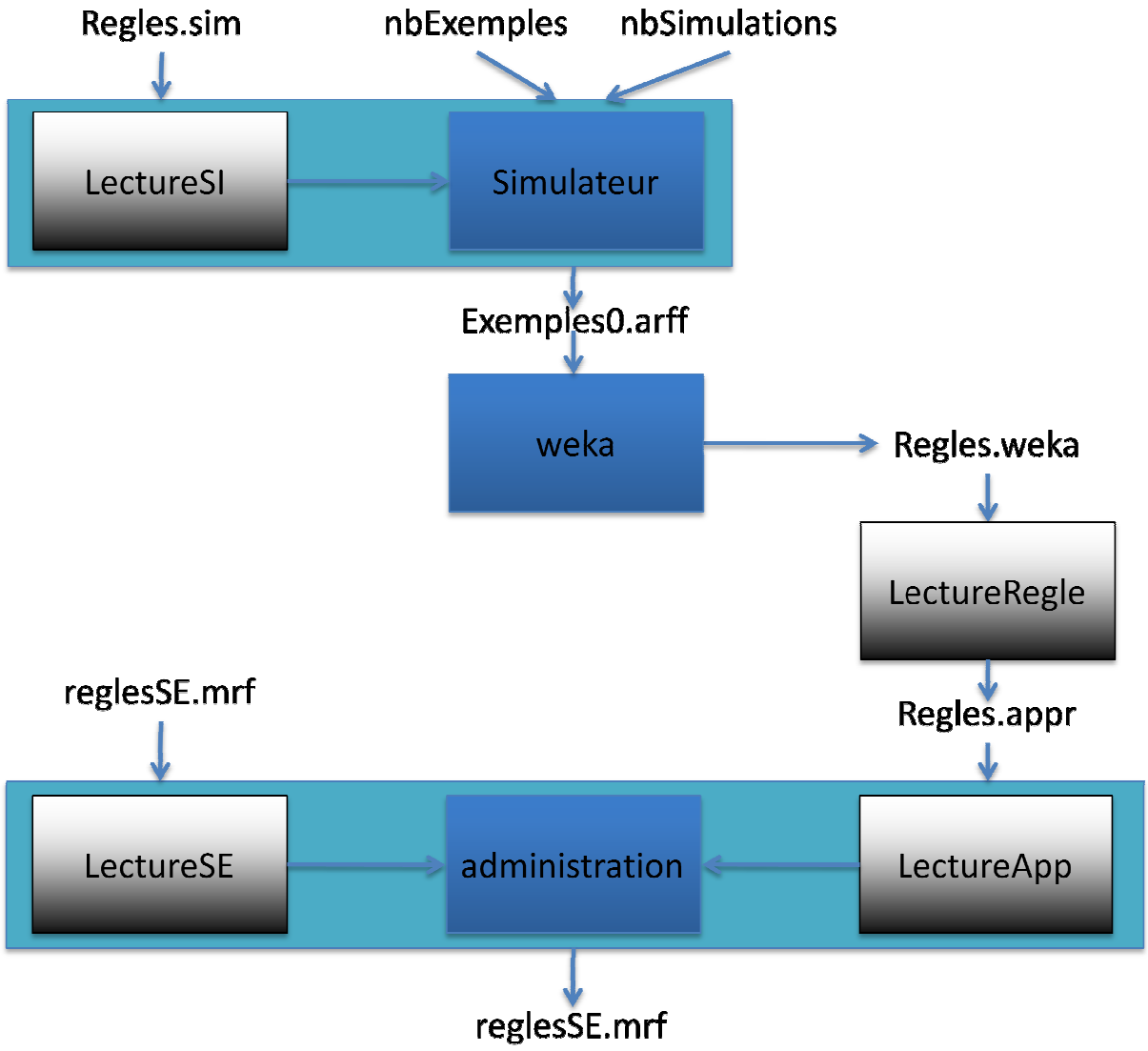


Figure 5 Schéma de décomposition du module d'administration

Package Outils

Contient toutes les classes pour réaliser une machine à règles, c'est à dire une machine contenant tout un ensemble de règles de la forme :

SI c1 et c2 FSI

Ce package ne contient pas de programme principal.

Package Simulateur

Ce package contient toutes les classes permettant de réaliser un simulateur. Une règle simulateur est une règle de la forme : SI c1 et c2 ALORS plantage=chaine FSI. La chaine de caractères est « non » s'il n'y aucun plantage, Windows si c'est un plantage Windows ou un nom d'application si c'est une application qui a planté.

Le fichier main contient le programme principal permettant de créer et d'exécuter un simulateur. Il prend en argument :

- le fichier décrivant les règles du simulateur
- le nombre d'exemples que l'on veut générer à chaque fois
- le nombre de fois où l'on génère les exemples

Package Classifieur

Il contient une unique classe permettant de créer par le code un classifieur et de générer le fichier des règles avec une extension « .weka ».

Package administration

Ce package contient toutes les classes permettant à l'administrateur d'étiqueter toutes règles apprises.

ii. Exemples d'utilisation

Le programme principal est la classe Learning1 contenu dans le jar « apprentissage.jar ». Celui-ci attend en paramètre le fichier « .appr » sans son extension. Il générera un fichier du même nom dont l'extension est « .mrf ». Le fichier « .appr » contient les règles apprises, c'est-à-dire de la forme : SI c1 et c2 ALORS plantage=chaine FSI. Le fichier « regleSE.mrf » contient des règles de la forme SI c1 et c2 ALORS a1 et a2 FSI, où a1 et a2 sont des actions.

Dans ce chapitre nous allons montrer la chaine pour pouvoir :

- générer les exemples par le smartphone6
- apprendre les règles
- étiqueter les règles et générer le fichier « .mrf » qui contient les règles du système expert

Génération des exemples

La première étape consiste à créer le fichier simulateur6 qui contient les règles du simulateur. Le contenu du fichier simulateur6.sim est le suivant :

Simulateur6.sim

```
Simulateur6
ATTRIBUT memoire NUMERIC
ATTRIBUT applicationA {lancee,nonLancee}
ATTRIBUT applicationB {lancee,nonLancee}
ATTRIBUT applicationC {lancee,nonLancee}
ATTRIBUT derniereVersionA {oui,non}
ATTRIBUT derniereVersionB {oui,non}
ATTRIBUT derniereVersionC {oui,non}
ATTRIBUT batterie {normale,faible,vide}
ATTRIBUT plantage {non,windows,applicationA,applicationB,applicationC}

SI memoire>200 ET applicationC=lancee ALORS plantage=windows FSI
SI memoire>240 ET applicationA=lancee ALORS plantage=applicationA FSI
SI applicationA=lancee ET applicationB=lancee ALORS plantage=applicationB FSI
```

La deuxième étape consiste à exécuter le main du package simulateur, avec pour argument simulateur6, 1000 et 8.

Main(« simulateur6 »,1000,3)

Un extrait du fichier exemple généré est le suivant :

Exemples2.arff

```
@relation simulateur6
@attribute memoire NUMERIC
@attribute applicationA {lancee,nonLancee}
@attribute applicationB {lancee,nonLancee}
@attribute applicationC {lancee,nonLancee}
@attribute derniereVersionA {oui,non}
@attribute derniereVersionB {oui,non}
@attribute derniereVersionC {oui,non}
@attribute batterie {normale,faible,vide}
@attribute plantage {non,windows,applicationA,applicationB,applicationC}
@data
213,lancee,lancee,lancee,oui,non,non,normale,windows
20,nonLancee,nonLancee,lancee,oui,non,oui,faible,non
232,nonLancee,lancee,lancee,non,non,oui,faible,windows
138,lancee,nonLancee,nonLancee,oui,oui,non,faible,non
205,lancee,lancee,lancee,oui,oui,oui,vide,windows
120,nonLancee,nonLancee,lancee,oui,non,oui,vide,non
163,nonLancee,nonLancee,lancee,oui,oui,non,vide,non
175,lancee,lancee,nonLancee,non,non,non,vide,applicationB
168,nonLancee,nonLancee,lancee,oui,oui,oui,normale,non
```

```
134,lancee,lancee,lancee,oui,non,non,normale,applicationB
50,lancee,lancee,lancee,non,oui,non,vide,applicationB
186,lancee,nonLancee,nonLancee,non,oui,oui,faible,non
85,lancee,lancee,lancee,oui,oui,non,vide,applicationB
246,lancee,lancee,nonLancee,non,non,non,normale,applicationA
```

Le simulateur génère aussi un fichier attribut ne contenant que les attributs.

Simulateur6.att

```
simulateur6
ATTRIBUT memoire NUMERIC
ATTRIBUT applicationA {lancee,nonLancee}
ATTRIBUT applicationB {lancee,nonLancee}
ATTRIBUT applicationC {lancee,nonLancee}
ATTRIBUT derniereVersionA {oui,non}
ATTRIBUT dernieVersionB {oui,non}
ATTRIBUT dernieVersionC {oui,non}
ATTRIBUT batterie {normale,faible,vide}
ATTRIBUT plantage {non,windows,applicationA,applicationB,applicationC}
```

Apprentissage des règles

La première étape consiste à recopier le fichier « exemples2.arff » en « simulateur6.arff ».

La deuxième étape consiste à exécuter le programme principal de monClassifieur avec comme argument « simulateur6 ». Le fichier simulateur6 contenant les règles après apprentissage est alors généré.

Simulateur6.weka

```
simulateur6
ATTRIBUT memoire NUMERIC
ATTRIBUT applicationA {lancee,nonLancee}
ATTRIBUT applicationB {lancee,nonLancee}
ATTRIBUT applicationC {lancee,nonLancee}
ATTRIBUT dernieVersionA {oui,non}
ATTRIBUT dernieVersionB {oui,non}
ATTRIBUT dernieVersionC {oui,non}
ATTRIBUT batterie {normale,faible,vide}
ATTRIBUT plantage {non,windows,applicationA,applicationB,applicationC}

 Regles
J48 pruned tree
------------------

memoire <= 200
|   applicationB = lancee
```

```
|   |   applicationA = lancee: applicationB (577.0)
|   |   applicationA = nonLancee: non (576.0)
|   applicationB = nonLancee: non (1204.0)
memoire > 200
|   applicationC = lancee: windows (306.0)
|   applicationC = nonLancee
|   |   applicationA = lancee
|   |   |   memoire <= 240
|   |   |   |   applicationB = lancee: applicationB (58.0)
|   |   |   |   applicationB = nonLancee: non (56.0)
|   |   |   memoire > 240: applicationA (47.0)
|   |   applicationA = nonLancee: non (176.0)

Number of Leaves  :  8

Size of the tree :      15
```

Génération des règles

Pour générer les règles du système expert, il suffit de reprendre le fichier « simulateur6.weka » et de sauvegarder toutes les règles conduisant à un plantage dans le fichier « simulateur6.appr ». Ceci se fait en exécutant le programme principal de LectureRegle avec comme argument simulateur6.

Simulateur6.appr

```
simulateur6
ATTRIBUT memoire NUMERIC
ATTRIBUT applicationA {lancee,nonLancee}
ATTRIBUT applicationB {lancee,nonLancee}
ATTRIBUT applicationC {lancee,nonLancee}
ATTRIBUT derniereVersionA {oui,non}
ATTRIBUT derniereVersionB {oui,non}
ATTRIBUT dernieVersionC {oui,non}
ATTRIBUT batterie {normale,faible,vide}
ATTRIBUT plantage {non,windows,applicationA,applicationB,applicationC}

SI memoire <= 200 ET applicationB = lancee ET applicationA = lancee ALORS plantage=applicationB FSI
SI memoire > 200 ET applicationC = lancee ALORS plantage=windows FSI
SI memoire > 200 ET applicationC = nonLancee ET applicationA = lancee ET memoire <= 240 ET applicationB = lancee ALORS plantage=applicationB FSI
SI memoire > 200 ET applicationC = nonLancee ET applicationA = lancee ET memoire > 240 ALORS plantage=applicationA FSI
```

Associations d'actions aux règles trouvées

Pour associer une action à une règle trouvée, il suffit de lancer le programme principal de la classe learning1. S'il n'existe pas déjà un fichier de règles, un fichier vide est créé, sinon le fichier en question est chargé.

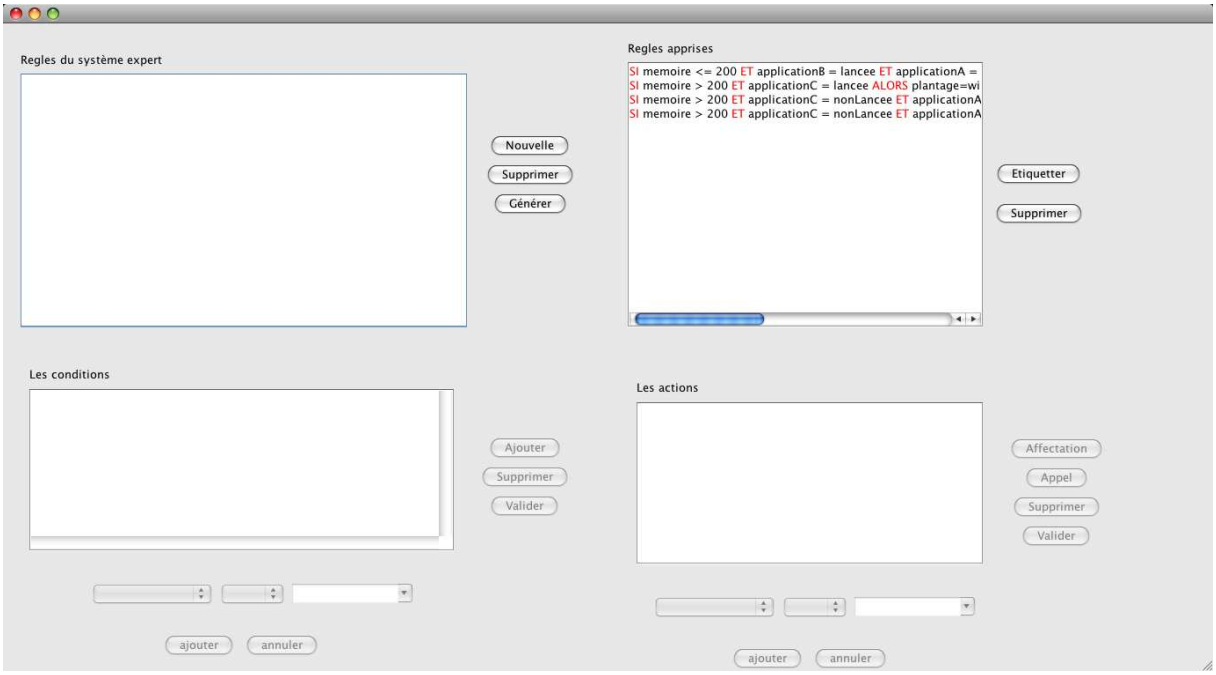


Figure 6 Vue principale de l'Interface Administrateur

III. Tests

A. Système Expert et module de Compilation

i. Système expert

Le système expert à été le premier module à être développé. Pour nous assurer de la qualité de notre logiciel nous avons procédé à des tests réguliers du module.

Tout au long du développement, nous avons effectué des tests unitaires des différentes fonctions de notre programme. De plus la conception très « éclatée » du système expert a permis de réduire au maximum la complexité de chaque fonction. Ainsi les tests unitaires ont été simples à réaliser ainsi qu'à tester. Par exemple, la fonction principale d'inférence, qui vérifie l'existence des faits et la validité de la règle, ne fait que 5 lignes.

Une fois chaque classe testée de cette manière, nous somme passé aux tests du module complet. Pour ce faire nous avons défini des règles simples à comprendre que nous avons traduits en code pour notre système expert. De cette manière nous avons pu nous assurer que les différents operateurs que nous souhaitions possibles pour notre système expert fonctionnaient correctement.

Nous avons ensuite pu vérifier que le chainage de règles fonctionnait correctement (Un fait du système provoque l'inférence d'un nouveau fait, puis ce nouveau fait déclenche une action). Ces tests effectués nous nous sommes intéressés au module de compilation.

ii. Module de compilation

Les tests pour ce module ont été différents des tests classiques. En effet ce module se base sur l'outil javaCC, et donc les tests unitaires sont impossibles à réaliser puisque chaque fonction est très fortement liée aux autres et est générée par javaCC. Nous avons donc effectué des tests plus globaux mais plus poussés. Nous avons tout d'abord vérifié que le compilateur obtenu reconnaissait bien la grammaire que nous souhaitions, sans nous préoccuper du code généré. Nous avons ainsi pu corriger des problèmes de reconnaissance (mots-clefs non cohérents avec ce que nous avions définis dans la grammaire, non reconnaissance des espaces dans les chaines de caractères protégées par des guillemets). Nous nous sommes également assurés qu'un fichier qui ne suivait pas la grammaire était bien détecté comme erroné par notre compilateur.

Nous avons ensuite testé le code C# généré. Nous avons corrigé de nombreuses erreurs à cet endroit car avant ce test nous ne nous étions pas assuré de la validité du code généré. Nous avons ainsi corrigé des erreurs de syntaxe, mais également des fonctions appelées incorrectement (ordre ou nombre des paramètres). Après ces quelques corrections nous avons pu procéder à des tests complets de la chaine fichier « .mrf » vers exécutable système expert. A cet endroit, nous n'avons détecté aucune erreur.

Nous avons finalement testé l'exécutable généré en lui-même, là encore sans détecter de problème.

B. Module de reporting

Nous avons développé la partie surveillance en plusieurs étapes possédant chacune une phase de tests unitaire. Cette approche nous à permis d'anticiper les problèmes.

Pour commencer à développer l'application de surveillance nous nous sommes concentrés sur l'écriture des rapports sans tenir compte du reste de l'architecture de notre logiciel. Ce premier programme était une application console qui écrivait dans un fichier texte la valeur des attributs de la classe rapport. Nous avons fait varier la valeur des attributs en changeant des paramètres sur le téléphone (L'émulateur fournit avec le SDK n'est pas satisfaisant pour tester en profondeur notre application car il simule un grand nombre de valeur du système : ainsi par exemple l'attribut ACLineStatus de la classe « Report » est à une valeur non définie, l'émulateur n'est ni branché ni débranché). Nous avons donc essayé sur un vrai mobile pour confirmer son bon fonctionnement. Les rapports écrits par le prototype était corrects et variait bien de manière conforme à nos prévisions.

Dans un second temps nous avons décidé de transformer ces rapports textuels en rapports binaires. Cette étape de conception fut également l'occasion d'ajouter une méthode de lecture de rapport en plus de celle d'écriture (Le prototype est toujours une application console). Le format binaire permet de gagner beaucoup de place dans l'écriture des rapports. Cet aspect étant critique pour une application embarquée, il était important de vérifier sont fonctionnement. La méthode écriture a donc subit des modifications pour écrire des octets en lieu et place des chaînes de caractères. Nous avons gardé 4 types de donnés : entier (4 octets), booléens (1 octet) et un attribut reste en chaîne de caractères pour représenter l'application courante. Cet attribut à également subit des modifications décrite dans la partie correspondante de ce document. Pour cette phase du développement nous avons également ajouté une méthode pour lire les rapports binaires. Cette méthode complète la classe report et lui donne toutes les fonctions qui permettent de s'en servir et d'effectuer des tests. En effet la méthode read() va être utilisée par le système expert pour lire les rapports. Pour valider le fonctionnement de la nouvelle classe « report » nous avons écrit différents rapports et essayé de les lire avec la fonction nouvellement écrite. Aucune différence n'étant constatée entre la lecture et l'écriture sur un nombre de rapports conséquent, nous avons décidés de valider le fonctionnement de ce module.

La dernière phase de la conception fut l'occasion de redéfinir l'interface entre l'utilisateur et l'application de Reporting. En effet, jusqu'à lors le programme était une simple application console. Cette dernière ne s'affiche pas pour l'utilisateur sous Windows Mobile (ni dans les applications en cours ni dans le gestionnaire de tâches). Il est ainsi très peu pratique de couper la surveillance. Il faut passer par un outil externe de gestion de processus et tuer le thread à la main. Nous avons donc décidés de créer une application qui comportera une fenêtre graphique et donnera la possibilité à l'usager de la quitter à tout moment. Cette fenêtre apparaît donc bien dans le gestionnaire des tâche et son comportement est identique à celui de l'application console. Cette phase à également été l'occasion de mettre en place différents timers en lieu et place de l'attente active qui était jusqu'à lors utilisée pour effectuer la surveillance. L'utilisation de ces timers permet de gagner en performance et de beaucoup moins ralentir la surveillance pour les usagers. L'attente active ralentissait considérablement le mobile et il devenait quasi-impossible de changer d'application.

C. Module d'apprentissage et interface Administration

Pour effectuer les tests unitaires nous avons utilisé Junit. A chaque classe a été associée une classe de test et un ou plusieurs tests par méthode de la classe. Le tableau ci-dessous récapitule les classes et le nombre de tests passés pour chacune d'entre elle.

Attribut	Nombre de tests
Attribut	9
AttributDiscret	5
AttributEntier	3
Condition	3
Domaine	5
DomaineDiscret	4
DomaineEntier	5
OpAttribut	5
OpDomaine	3
OpValeur	3
Operande	8
Operateur	5
Regle	3
Simulateur	11

D. Test de la chaîne complète

Finalement, nous avons effectué un test de tous nos modules. Les tests précédents nous ont évité de nombreux problèmes. Cependant ce test à fait apparaitre un problème d'encodage des caractères. Ce problème a pu être résolu en ajoutant une tâche de changement d'encodage avant d'utiliser les fichiers sur le serveur. Une fois ce problème résolu, la chaîne a correctement fonctionné.

Le test de la chaîne complète consistant en :

- envoyer via MediaContact le système expert
- provoquer des plantages suivant un critère défini
- réceptionner les rapports du mobile
- lancer l'apprentissage sur ces rapports
- vérifier la règle obtenue et s'assurer qu'elle est cohérente avec les plantages provoqués
- ajouter une action corrective (factice) à cette règle
- régénérer les règles du système expert
- envoyer le système expert à nouveau sur le téléphone
- vérifier que le système expert détecte et déclenche correctement l'action corrective

IV. Bilan d'avancement du projet

À l'heure actuelle, notre projet est fonctionnel : Le module de création de rapports ainsi que le système expert qui étaient les objectifs principaux de notre projet fonctionnent complètement.

Du coté du module de création de rapport, l'ajout de données dans ce même rapport est relativement simple et ne nécessite que peu de modifications. Cela permettra l'extensibilité de notre projet dans le futur.

Le Système expert en lui-même est capable de gérer des données aux formats texte, enum ou numériques, quelque soit leur nombre. L'ajout de donnée dans les rapports ne nécessite donc que la modification de la classe chargée de lire les données depuis le rapport.

La communication entre le serveur et les clients fonctionnent en passant via MediaContact. Il suffit d'automatiser les appels aux différents programmes et scripts pour rendre le système totalement autonome.

L'apprentissage, partie que nous n'étions pas sûr de développer au début de notre projet, fonctionne également. L'ajout d'éléments dans les rapports est pris en charge (tout du moins pour les types enum, strings et numériques). L'apprentissage sur d'autre type de données n'est cependant pas géré.

En conclusion, nous avons passé plus de temps que prévu sur les phases de pré-étude, de spécification et de conception. Malgré tout le temps passé sur les premières phases du projet nous a permis d'effectuer une implémentions rapide. Le projet a donc été complété dans les temps.