EXTRACCIÓ DE TASQUES LOCALS D'UN ITINERARI PROTEGIT PER ARQUITECTURES CRIPTOGRÀFIQUES

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Xavier Franquet Conte i dirigit per Guillermo Navarro Arribas.

Bellaterra, Juny de 2006

El firmant, Guillermo Navarro Arribas, professor del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Xavier Franquet Conte

Bellaterra, Juny de 2006

Firmat: Guillermo Navarro Arribas

A la meva família

Agraïments

M'agradaria agrair al Carles Garrigues i a l'Álvaro Moratalla per tot el suport tècnic que m'han ofert durant els mesos que ha durat la realització del projecte.

També vull agrair tots els consells i crítiques que m'ha donat el meu director de projecte, Guillermo Navarro, i que m'han ajudat a que aquesta memòria acabés sent mínimament intel·ligible.

Finalment agraeixo a la meva família el suport constant que m'ha anat donant durant tota la carrera i, especialment, durant la realització d'aquest projecte.

Índex

Índex de figures

Capítol 1

Introducció

Els agents mòbils són la solució a moltes aplicacions que necessiten una execució distribuïda. Un agent mòbil es podria definir com un programa que és capaç de migrar la seva execució des d'una màquina cap una altra. A diferència del codi mòbil (un programa que s'executa en una màquina remota) els agents mòbils són capaços de decidir per ells mateixos quan i cap a on fer les migracions. Això obre les portes a un gran ventall d'aplicacions que necessiten d'aquesta execució distribuïda.

Aquests agents no poden sobreviure ni migrar per si sols de màquina a màquina, per donar-los suport és necessari l'us d'agències (o plataformes) a les màquines que els facin de hoste. Aquestes plataformes els recullen i els proporcionen els recursos necessaris per tal de que es puguin executar i, quan l'agent ho decideixi, ajudar-los a migrar cap una altra plataforma.

La ruta que segueixen els agents quan van migrant de plataforma en plataforma pot ser dinàmica (no es possible conèixer-la a temps de compilació per que es crea segons els resultats obtinguts a cada plataforma) o pot estar marcada per un itinerari, aquest itinerari contindrà la llista de les agències que seguirà l'agent durant la seva execució. L'itinerari pot estar, o no, protegit per tal de que les plataformes no el puguin llegir ni modificar.

La protecció d'un itinerari la defineixen les arquitectures d'agents mòbils, que indiquen de quina manera s'ha de construir l'itinerari de l'agent i quines funcions

criptogràfiques s'hi han d'aplicar. Les arquitectures estan escrites en el llenguatge de protecció d'itineraris (IPL).

Actualment hi ha tres arquitectures dissenyades. L'arquitectura més senzilla de les tres és la *simple* [MB03], la qual no aplica cap funció criptogràfica a les dades, només les empaqueta adequadament.
Una altra arquitectura és la *nested* [RMB02], que aplica una encriptació asimètrica als nodes per on passa l'agent. A més aquesta arquitectura situa les dades de cada node dintre del node anterior aniuant-les.
La tercera és l'arquitectura *scrambled* [MB03], que es basa en encriptar amb una clau simètrica les dades dels nodes i amb clau asimètrica la clau que s'ha de fer servir per desencriptar-los.

Aquest projecte esta fet dintre del grup SeNDA (*Security of Networks and Distributed Applications*) del dEIC (departament d'Enginyeria de la Informació i de les Comunicacions). Els projectes que s'han dut a terme aquest any en el grup SeNDA han seguit una metodologia interessant, tota la feina del projecte l'hem anat documentat en una wiki i tant directors de projecte com projectistes ens hem anat reunint periòdicament a la sala Shannon del dEIC per tal d'exposar els canvis que hi hem dut a terme, a més de comentar les evolucions de cada projecte i els problemes que s'hagin pogut trobat, per solucionar-los en equip. A banda d'aquestes reunions de grup també s'han seguit fent les habituals reunions individuals amb els directors i tutors de projecte.

1.1 Motivacions

Un dels projectes actuals del grup SeNDA és la creació d'un Entorn de Desenvolupament Integrat (IDE) que engloba el disseny, la creació, el llançament i la recollida de dades d'un agent mòbil de forma compacta i gairebé automàtica. El desenvolupament d'aquesta eina s'ha realitzat en gran part en projectes finals de carrera dels darrers anys.

La vida d'un agent creat amb aquesta eina comença dissenyant el seu itinerari en un entorn gràfic molt senzill i escollint l'arquitectura criptogràfica que el pro-

tegirà. Un cop creat aquest itinerari s'ha de generar el codi local a executar a cada plataforma i el codi comú a totes. Després, amb totes aquestes dades, es compila l'agent, i aquesta mateixa IDE el llança a la primera plataforma on s'executarà. En aquest punt l'agent viatja seguint la seva ruta d'agències i finalment, quan l'agent ha finalitzat la seva execució, la IDE el recull i n'extreu els resultats.

La compilació de l'agent es fa amb l'itinerari que hem dissenyat i amb l'arquitectura que s'ha escollit per a protegir-lo. L'arquitectura ha d'estar escrita en un llenguatge de protecció d'itineraris que va ser dissenyat i implementat en un projecte final de carrera del 2005 per en Jordi Garcia [Gar05a].
Actualment, una de les funcionalitats que els manca als agents creats per aquesta interfície, és que a cada plataforma on s'executi l'agent, hi hagi una classe que extregui de l'itinerari les tasques locals a la plataforma. Per a tasques locals ens referim al codi local que s'ha d'executar (diferenciant-lo del codi comú que s'executarà a totes les plataformes), la llista de plataformes on podem migrar després de l'actual, l'itinerari que propagarem a les plataformes on migrem, etc.

Aquesta serà la tasca que intentarà implementar aquest projecte.

1.2 Objectius

L'objectiu principal d'aquest projecte és extreure, de forma automàtica, les tasques locals d'un itinerari a una plataforma determinada. Aquest itinerari, que prèviament haurà estat construït amb una arquitectura escrita en llenguatge IPL (*Itinerary Protection Language*), pot estar protegit per protocols criptogràfics.

Hi ha dues formes d'assolir aquest objectiu, bé construint un algorisme que, partint del codi IPL de l'arquitectura en dedueixi el codi per desfer la protecció o bé fent una extensió del llenguatge IPL que sigui capaç de compilar-se en una classe que extregui aquestes dades.

Hem cregut que la opció més simple, i per tant més eficient, de dur a terme el nostre projecte és la d'implementar una ampliació del llenguatge existent en comptes de fer un algorisme que tan sols amb la descripció de l'arquitectura en IPL n'extregui les tasques locals.

Aquesta decisió obligarà als programadors que desenvolupin una determinada arquitectura, per als itineraris en IPL, en creïn també la forma d'extreure-la. La intenció és que aquesta tasca sigui senzilla de realitzar per a qui ha dissenyat l'arquitectura o per a qui l'entengui.

Un cop dissenyat el llenguatge d'extracció de tasques s'ha de fer un compilador per aquest llenguatge que, compilant-lo amb una arquitectura d'extracció, generi una classe que sàpiga extreure informació d'itineraris generats amb la mateixa arquitectura que l'ha compilat.

Resumint, els objectius esperats per aquest projecte són els següents:

- Analitzar el llenguatge IPL de construcció d'itineraris per tal de ser capaços d'ampliar-lo.
- Dissenyar una extensió per aquest llenguatge que permeti definir com extreure les tasques locals.
- Implementar un compilador d'aquest nou llenguatge.
- Escriure les arquitectures d'extracció *simple* i *nested* amb el nou llenguatge.
- Aconseguir compilar aquestes dues arquitectures i comprovar que generin classes extractores correctes.
- Comprovar el bon funcionament de les classes extractores amb itineraris reals
- Integrar la capacitat d'extreure tasques d'itineraris amb la IDE de gestió i creació d'agents.

1.3 Planificació

El projecte l'hem planificat per a dur-lo a terme en 4 mesos. Començarem amb la documentació mitjans de gener i finalitzarem la implementació a mitjans de

maig. Després, els mesos maig i juny els dedicarem a fer les proves finals de la implementació i a escriure la memòria.

A continuació mostrem com hem repartit la feina per aquests mesos:

Gener - Febrer Estudiar protocols criptogràfics per a la protecció d'itineraris *simple* i *nested* així com el llenguatge de protecció d'itineraris (IPL) per a descriure'ls.

Març Dissenyar l'extensió d'aquest llenguatge per a extreure les tasques locals i intentar emprar-lo per escriure algunes arquitectures existents.

Març - Abril - Maig Crear el compilador d'aquesta ampliació del llenguatge.

Maig Crear la versió final de les arquitectures *simple* i *nested* per a la nostra ampliació del llenguatge i comprovar el bon funcionament del compilador en casos reals.

Maig - Juny Escriure la memòria.

Juny Entregar el projecte, la memòria i fer la presentació.

1.4 Estructura

Els objectius abans esmentats els tractarem al llarg dels següents capítols:

Capítol 2: Definicions Prèvies Aquest capítol serà una guia per a assolir els coneixements previs a la realització d'aquest projecte. El seu contingut serà indispensable per a poder entendre i seguir el desenvolupament del projecte, bàsicament explicarem cinc conceptes:

- Introducció als Agents Mòbils.
- Itineraris i Arquitectures Criptogràfiques d'Agents Mòbils.
- *Itinerary Protection Language*.
- Eina de construcció d'agents IDE-Marisma.

- Introducció al llenguatge de programació de compiladors JavaCC.

Capítol 3: Anàlisi i Disseny del Llenguatge En aquest capítol farem l'anàlisi de com ha de ser l'ampliació del llenguatge, així com de les funcionalitats que requerirà el nostre compilador d'arquitectures.

Capítol 4: Disseny i Implementació del Compilador Aquest capítol explicarà pas a pas com hem fet per a construir el compilador a partir de l'anàlisi que n'hem dut a terme. Això inclou tant la decisió de quin llenguatge de programació usar com la fase lexicogràfica, sintàctica, semàntica i la de generació de codi.

Capítol 5: Proves Per tal de comprovar el bon funcionament del nostre compilador, dissenyarem dues arquitectures criptogràfiques d'extracció amb el llenguatge que prèviament hem creat. Seguidament comprovarem que el nostre compilador sigui capaç de compilar-les i extreure'n satisfactòriament les tasques locals per a dos itineraris diferents.

Capítol 6: Conclusions Finalment farem una valoració del grau d'assoliment dels objectius del projecte així com de les possibles línies de continuació.

Capítol 2

Definicions Prèvies

Després d'haver vist, a la introducció, els objectius que provarà d'assolir el nostre projecte, anem a estudiar els requisits previs que necessitarem per a realitzar-lo. Començarem fent un petit recordatori de què són els Agents Mòbils i introduirem la funció dels Itineraris i de les Arquitectures criptogràfiques.
Més tard farem una petita introducció al projecte que duu a terme el SeNDA de crear un Entorn de Desenvolupament Integrat (IDE) per al disseny, desenvolupament i gestió intuïtiva d'Agents Mòbils.
Seguidament explicarem més en detall el llenguatge de programació d'itineraris que forma part d'aquesta IDE i que va ser desenvolupat l'any passat en un projecte final de carrera. La definició complerta d'aquest llenguatge es pot trobar al projecte final de carrera d'en Jordi Garcia [Gar05a] i a la tesina d'en Carles Garrigues [Gar05b].
Finalment farem un petit incís sobre l'ús del compilador de compiladors que hem fet servir per a realitzar la implementació del projecte, el JavaCC.

2.1 Agents Mòbils

Hi ha aplicacions distribuïdes on el clàssic esquema de client-servidor no és adequat i és necessari tenir processos que vagin d'una màquina a una altra recollint o tractant dades. Per a aquestes aplicacions és necessari fer servir un altre paradig-

ma de la programació, els agents mòbils. Un agent mòbil és un programa que té les propietats de ser:

Autònom Vol dir que pren decisions per compta pròpia.

Reactiu Reacciona davant canvis al seu entorn.

Proactiu Ell mateix pot provocar canvis a l'entorn.

Comunicatiu És capaç de comunicar-se amb d'altres agents o recursos.

Mòbil En un moment donat pot migrar la seva execució a una altra màquina.

Els agents mòbils no es poden executar si no és amb l'ajuda una agència (o plataforma). Les agències recullen l'agent, l'executen i, si fa falta, el fan migrar cap a d'altres plataformes. Hi ha diverses implementacions d'agències, però la més utilitzada és la implementació de JADE (*Java Agent Development Framework*) que és de codi obert i que compleix les especificacions de la FIPA (*Foundation for Intelligent Physical Agents*: L'estàndard més utilitzat dels que regulen els agents mòbils).
El camí que segueixen els agents al recórrer aquestes agències s'anomena itinerari. Aquest itinerari pot estar protegit amb arquitectures criptogràfiques per tal de que les agències no siguin capaces de llegir-lo, modificar-lo o fer-lo servir en contra de l'agent. Tant el terme Itinerari com el d'Arquitectura seran explicats amb detall més endavant.

Els agents mòbils són molt adequats per a executar aplicacions del tipus Mar de Dades, aquestes aplicacions distribuïdes necessiten accedir a molts recursos repartits en màquines diferents. Si en una d'aquestes aplicacions no féssim servir agents mòbils, hauríem de fer una petició a les màquines remotes per a que ens enviessin les dades i tractar-les a la nostra màquina. Es pot veure que això és molt poc efectiu, ja que aquestes dades poden ocupar molt espai i el temps d'enviar-les pot ser massa gran, o bé pot ser que sobre les dades hi hagin restriccions legals que no permetin que siguin enviades a d'altres màquines. Si en aquest mateix esquema fem servir agents mòbils (Figura 2.1) l'únic traspàs d'informació que

hi haurà entre màquines serà el propi agent, que pot ser un programa petit, i els resultats que aquest obtingui de tractar les dades a les màquines remotes.

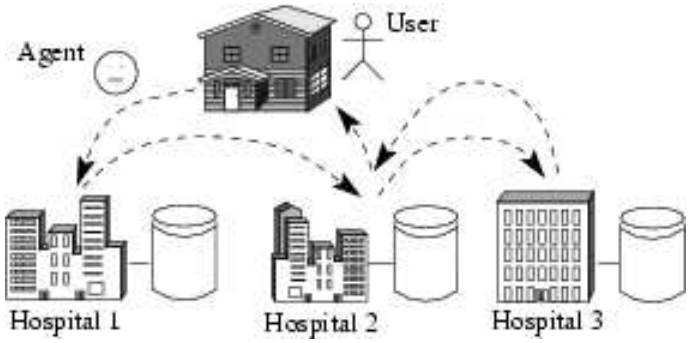

Figura 2.1: Exemple d'execució d'un agent mòbil en Mar-de-Dades [Rob05]

2.2 Itineraris i Arquitectures d'Agents Mòbils

Itineraris

Com el seu nom indica, l'itinerari d'un agent és el camí que aquest seguirà durant la seva execució.

D'itineraris n'hi ha de dos tipus depenent d'on estiguin implementats, els implícits i els explícits. Els itineraris implícits estan escrits dintre del codi d'execució de l'agent, això vol dir que el codi de l'itinerari esta barrejat amb el codi que executa l'agent a cada plataforma. Per a itineraris petits això no es cap problema, però aquesta forma de fer no és adequada per a itineraris més complexos ja que es fa impossible seguir el flux de migracions mirant el codi.

L'altre tipus són els itineraris explícits. Aquests itineraris tenen una estructura separada del codi que és on esta escrit l'itinerari. Fan servir una repetició recursiva d'elements del tipus Seqüència, Alternativa i Clonació. Els elements Seqüencials (Figura 2.2), després de la seva execució, migren a la següent plataforma que els indiqui l'itinerari; els elements d'Alternativa (Figura 2.3) executen una condició que els indica a quina plataforma de la llista de possibles han de migrar, mentre que els de Clonació (Figura 2.4) migren a diverses plataformes a la vegada i

segueixen l'execució a totes elles. La utilització d'aquest últim tipus d'itineraris permet l'us d'Arquitectures d'Agents Mòbils per a protegir-los.

Figura 2.2: Itinerari seqüencial.

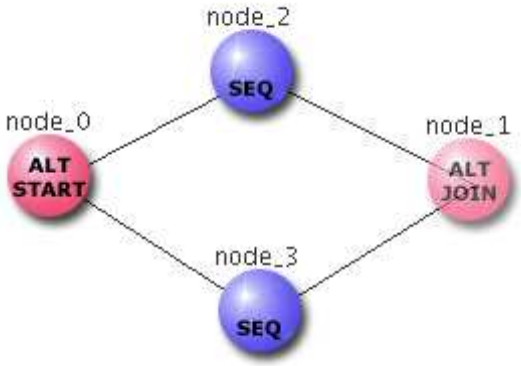

Figura 2.3: Itinerari amb alternativa.

Arquitectures d'Agents Mòbils

Una Arquitectura d'Agents Mòbils defineix l'estructura amb la qual es codificarà un itinerari explícit. Aquestes arquitectures pot ser que apliquin funcions criptogràfiques sobre les dades de l'itinerari, protegint-les de plataformes que no les haurien de poder llegir. L'ús d'arquitectures deslliga al programador d'agents mòbils d'haver de dissenyar el sistema de migracions del seu agent i el protocol per a protegir-lo, fent que aquest agent sigui més fàcil d'escriure i molt més segur en un entorn hostil.

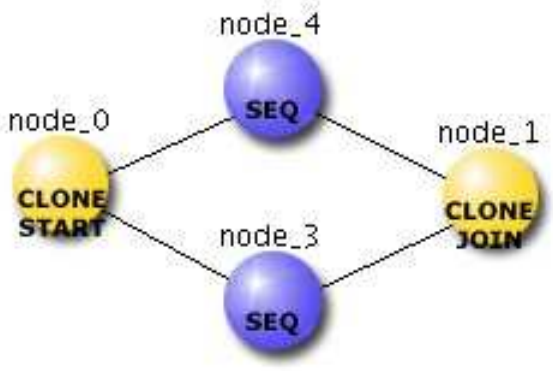

Figura 2.4: Itinerari amb clonació.

L'inconvenient és que no és una tasca senzilla la d'integrar una arquitectura a l'itinerari d'un agent, ja que tant l'arquitectura com l'itinerari a protegir poden ser bastant complexos. Per aquesta raó s'està creant una eina de disseny, creació i llançament d'agents anomenada IDE-Marisma que és capaç de crear l'itinerari codificat de l'agent mitjançant la ruta que ha de seguir l'itinerari i l'arquitectura que el protegirà. Aquesta IDE serà explicada més endavant.

2.3 *Itinerary Protection Language*

El llenguatge IPL o *Itinerary Protection Language* ([Gar05a] i [Gar05b]) és un llenguatge que, seguint una sintaxi molt semblant a la descripció formal de les arquitectures, pretén simplificar la seva implementació. És un llenguatge basat en regles on, la primera que hi trobem, és la regla ITINERARI. Aquesta regla és per on es comença a construir l'itinerari. Després del nom de la regla pot venir definit l'àmbit on té sentit, com per exemple si és un node Seqüencial, d'Alternativa o de Clonació. Seguidament ve el contingut de la llista que generarà, que és una successió dels elements que conformaran l'itinerari resultant, separats per l'operand d'unió d'elements compostos (llistes), el caràcter DOSPUNTS (*:*).
Els elements d'aquest contingut poden ser altres regles (amb paràmetres o sense), variables o *strings* o bé assignacions de variables a qualsevol d'aquests anteriors.

A més, aquests elements poden portar un sufix del tipus *#a...b* que indica a quin àmbit temporal es refereix l'element. Per exemple, si trobem un element que sigui *NODE#i...e* equivaldrà a trobar *NODE#i : NODE#i+1 : NODE#i+2 : ... : NODE#e-1 : NODE#e*.
Les diferents lletres que podem trobar en aquest sufix tenen cadascuna un significat, per exemple, la lletra *i* significa el node actual, la *j* el node següent, la *k* l'anterior, etc. Una vegada vist el llenguatge una mica per sobre anem a veure un parell exemples d'arquitectures escrites amb ell.

A la Figura 2.5 podem veure l'arquitectura *simple* escrita en llenguatge IPL. A la primera regla de l'arquitectura (ITINERARI) podem veure que és una llista de nodes (*NODE#i...e*). A les següents regles es defineix la regla NODE en cadascun dels seus àmbits (seqüencial, alternativa, clonació i unió). Tots aquests àmbits tenen en comú que comencen amb una llista on hi son el codi local del node, el tipus de node i el HostName de la següent plataforma. Els nodes amb més d'una plataforma de destí (nodes d'alternativa i de clonació), després de la informació del node, tenen descrits els sub-itineraris de cada node.

```
ITINERARI = [ NODE#i...e ] ;
NODE{SEQ} = [ LocalCode#i : "seq" : HostName#sf ] ;
NODE{ALT_START} = [ LocalCode#i : "alt" :
    HostName#sf...sl ] : ITINERARI#sf...sl ;
NODE{CLONE_START} = [ LocalCode#i : "clone" :
    HostName#sf...sl ] : ITINERARI#sf...sl ;
NODE{JOIN_START} = [ LocalCode#i : "join" :
    HostName#sf ] ;
```

Figura 2.5: Arquitectura *simple* escrita en llenguatge IPL.

L'Arquitectura *nested* [RMB02] escrita en llenguatge IPL la podem veure a la Figura 2.6. Aquesta arquitectura comença amb el primer node (*NODE#i*) a la

regla ITINERARI. La definició dels nodes es fa a la regla NODE amb tots els seus àmbits. La diferència principal respecte la *simple* és que a tots els nodes se'ls aplica una encriptació asimètrica (*aencrypt(clau, dades)*) i que el següent node (*NODE#j*) sempre esta dintre de la descripció de l'anterior.

```
ITINERARI = [ NODE#i ] ;
NODE{SEQ} = aencrypt( ki, [ [ localCode#i : "seq" :
    hostName#j ] : NODE#j ]) ;
NODE{ALT_START} = aencrypt( ki, [ [ localCode#i : "alt" :
    hostName#sf...sl ] : ITINERARI#sf...sl : NODE#j ] ) ;
NODE{CLONE_START} = aencrypt( ki, [ [ localCode#i :
    "clone" : hostName#sf...sl ] : ITINERARI#sf...sl :
    NODE#j ] ) ;
NODE{CLONE_JOIN} = aencrypt( ki, [ [ localCode#i :
    "join" : hostName#j ] : NODE#j ]);
```

Figura 2.6: Arquitectura *nested* escrita en llenguatge IPL.

No és l'objectiu d'aquesta memòria fer una descripció detallada d'aquest llenguatge, sinó explicar-lo una mica per sobre per a poder seguir el desenvolupament del projecte, la seva descripció complerta, així com les fases que es van haver de recórrer per arribar-hi es pot trobar al projecte final de carrera d'en Jordi Garcia [Gar05a].

2.4 IDE-Marisma

El projecte IDE-Marisma pretén integrar, en un sol entorn, el disseny de l'itinerari d'un agent, la codificació d'aquest itinerari amb l'arquitectura que faci falta, la creació de l'agent i el seu posterior llançament cap a la plataforma on ha de començar la seva execució.

El treball amb aquesta IDE comença dissenyant la ruta que ha de seguir l'agent per les plataformes i escollint l'arquitectura que el protegirà. Com ja hem comentat al punt anterior, aquesta IDE porta incorporat un compilador d'arquitectures que per ell mateix agafa la ruta de l'agent i l'arquitectura i en retorna l'itinerari codificat. Això és un gran avenç ja que la feina que fa aquest compilador és molt pesada de fer-la a mà. A part d'aquesta compilació, la IDE empaqueta l'agent i, en futurs projectes, serà capaç de llançar-lo a la primera plataforma on s'ha d'executar. Podem veure un *screenshot* d'aquesta IDE a la Figura 2.7.

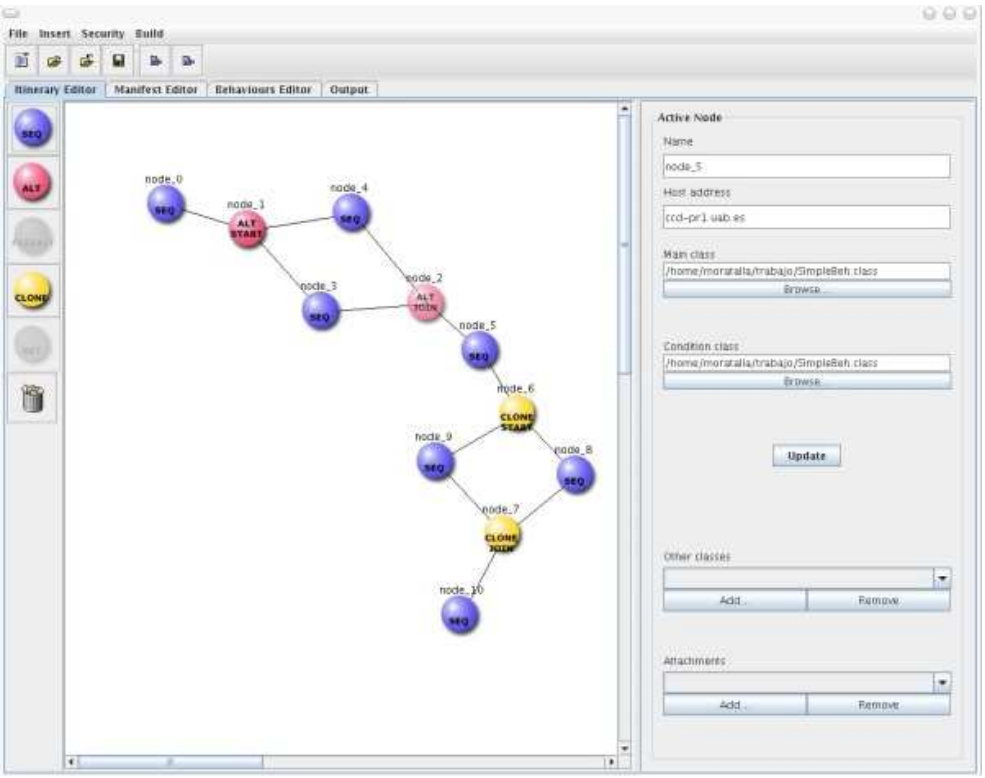

Figura 2.7: Captura de pantalla de la IDE MARISM-A

2.5 JavaCC

Per tal d'implementar el compilador hem fet servir una eina desenvolupada per la comunitat de Java.net anomenada JavaCC (*Java Compiler Compiler*). La sintaxi

que fa servir aquest compilador de compiladors és molt semblant al EBNF (*Extended Backus-Naur Form*) amb que descrivim formalment les gramàtiques dels compiladors.

Vegem un exemple d'aquesta adaptació. Una regla BNF molt simple, que gestiona els paràmetres d'una funció, com la següent:

```
<llistaParam> ::= PO [ [ REF ] <param> { COMA [ REF ]
     <param> } ] PT;
```

on PO i PT denoten els símbols de parèntesi obert i parèntesi tancat, COMA representa el símbol ",ï REF qualssevol símbol que pugui voler dir que aquell paràmetre és un paràmetre per referencia; al passar-la a JavaCC queda així:

```
void llistaParam():
{}
{
    <PO>
    (
        ( <REF> )?
        param()
        (
            <COMA>
            ( <REF> )?
            param()
        )*
    )?
    <PT>
}
```

Alhora de posar la semàntica i generació de codi a la sintaxi definida en JavaCC només cal afegir codi java entre claus on faci falta. Per exemple, si volem, sobre l'exemple anterior, implementar un comptador per a saber el número de paràmetres ho farem així:

```
void llistaParam():
{
    int numeroParametres = 0;
}
{
    <PO>
    (
        ( <REF> )?
        param()
        {
            numeroParametres = 1;
        }
        (
            <COMA>
            ( <REF> )?
            param()
            {
                numeroParametres += 1;
            }
        )*
    )?
    <PT>
}
```

Per a més informació sobre aquesta eina es poden consultar els tutorials de la *Memorial University* de Canadà [Nor05] o anar a la pàgina web http://www.java.net.

2.6 Resum

En aquest capítol hem definit breument els conceptes necessaris per a poder seguir amb facilitat el desenvolupament d'aquest projecte. Hem començat el capítol fent

una definició general del paradigma dels Agents Mòbils, més endavant ens hem centrat en una part més concreta d'aquest paradigma, els itineraris i les arquitectures, i concretant més hem introduït el llenguatge que es fa servir per descriure la protecció dels itineraris.

A continuació hem explicat el projecte de l'Entorn de Desenvolupament Integrat (IDE - *Integrated Development Environment*) i finalment hem introduït el llenguatge que hem usat per a escriure el compilador d'arquitectures, el JavaCC.

Capítol 3

Anàlisi i Disseny del Llenguatge

Fins ara hem adquirit els coneixements necessaris per a poder entendre la majoria dels conceptes que s'exposaran en aquesta memòria. En aquest capítol analitzarem, a partir del treball previ que va fer en Jordi Garcia en el seu projecte de compilació d'agents mòbils, com ha de ser el llenguatge que creem i les limitacions que tindrà. Començarem amb l'anàlisi de com estarà format el llenguatge i després seguirem per la descripció del llenguatge que ens ha quedat. Aquesta serà una descripció bastant acurada que esperem que serveixi, per a qui la llegeixi, per entendre bé el llenguatge.

3.1 Anàlisi

Alhora de començar a dissenyar l'ampliació del llenguatge de protecció d'itineraris (a partir d'ara anomenat *Itinerary Protection Language* o IPL) hem contemplat les següents possibilitats:

1. Fer les extraccions amb una sintaxi com la del Lisp. Aquesta opció seria prou bona, ja que els itineraris estan implementats en llistes i el Lisp és un llenguatge on el tractament de llistes és molt potent.

2. Inventar un llenguatge nou segons les necessitats que es vagin trobant.

3. Fer el l'ampliació del llenguatge amb una sintaxi molt propera al llenguatge de creació d'itineraris IPL.

En un primer cop d'ull, l'opció que sembla més bona és fer el llenguatge seguint la sintaxi de Lisp, ja que això simplificarà molt la feina de tractar amb llistes i la de recórrer els itineraris, però al final ens hem decantat per fer el llenguatge seguint una sintaxi com la del llenguatge IPL de construcció d'itineraris.

Aquesta decisió s'ha pres per diverses raons. Primerament, no trobem correcte que el llenguatge amb que esta escrita una arquitectura de construcció d'itineraris sigui gaire diferent al llenguatge que es farà servir per escriure la seva arquitectura d'extracció. Una altra raó és que, com que el llenguatge IPL és, per ell mateix, l'evolució d'una primera versió del llenguatge basada en Lisp, el tractament de les llistes no serà una tasca complicada.

Després d'aquest anàlisi de quines pautes haurà de seguir el llenguatge hem començat amb la seva fase de disseny.

3.2 Descripció del llenguatge

El llenguatge, igual que el de construcció d'itineraris, esta basat en regles. Però, a diferencia del primer, el nostre en tindrà de dos tipus diferents, les de desmembració de l'itinerari que arriba i la de reconstrucció d'un nou itinerari per a les següents plataformes.
Les regles de desmembració s'encarregaran de descriure com es tractarà l'itinerari d'entrada per tal d'extreure les tasques locals a la plataforma, mentre que les regles de reconstrucció indicaran com construir els itineraris que enviarem a les plataformes successores per tal de que aquest tingui el mateix format que el d'entrada i les agències el puguin extreure.

Després d'una primera versió simple del llenguatge, per tal d'anar completant-lo, hem intentat implementar l'extracció de les arquitectures *simple*, *nested* ([RMB02]) i *scrambled* ([MB03]). Aquestes implementacions ens han ajudat trobar funcionalitats necessàries per al llenguatge i a generalitzar-lo per tal de que sigui capaç de descriure la majoria, sinó totes, les arquitectures d'extracció.

A continuació descriurem els dos passos en que esta descompost el nostre llenguatge. Per començar parlarem del desmembrament i tractament de l'itinerari d'arribada i per finalitzar de la reconstrucció dels itineraris per a les següents plataformes.

Desmembrament de l'itinerari d'arribada

La intenció del desmembrament és la de recol·lectar les dades importants de l'itinerari. Això es fa parsejant l'itinerari i decidint què fer amb cadascun dels elements que el formen tal i com es descriu en el IPL d'extracció.

Les regles d'aquesta part tenen la següent sintaxi:

```
NOM_REGLA [(CODI_DE_CONTROL)] = TRACTAMENT
```

On:

NOM_REGLA És un identificador alfabètic, tot en majúscules i sense espais. És el nom de la regla que estem definint. Aquest servirà per a que d'altres regles puguin referir-s'hi. Des de les regles es podran anar cridant a d'altres amb funcions més específiques.
La primera regla que trobem és LOCALITINERARI i és la que s'encarregarà de tractar l'itinerari d'entrada de la classe.

CODI_DE_CONTROL És una part opcional de la regla. És el codi que s'executarà abans del tractament de l'itinerari d'entrada de la regla. Les diferents instruccions estaran separades pel caràcter PUNT I COMA (*;*) i sempre ha d'haver-hi un operador de *Return* que serà el que finalment es tractarà a la regla.
Si una regla no té codi de control es sobreentén que el codi de control és: "*Return Self;*"

TRACTAMENT Aquesta és la part de la regla que indica com s'ha de tractar la llista d'entrada. Tot el que fa és treballar amb llistes.
La sintaxi del TRACTAMENT és:

```
[ ITEM1 : ... | RESTA_LLISTA ]
```

Els ITEMs d'aquesta llista poden ser:

- un símbol especial com ara LocalCode, HostName, NextAgencyList o NodeType. Si apareix algun d'aquests tokens, el tros d'itinerari que correspongui és guardarà a la variable global que indiqui el símbol.
- una VARIABLE per a composar l'itinerari de les següents plataformes. Una variable és un identificador alfabètic tot en minúscules. Funciona igual que els símbols especials, quan apareix, pren el valor del tros de llista on estigui col·locada.
- una REGLA. Quan aparegui el nom d'una regla al tractament, la part d'itinerari que li pertoqui serà tractada per aquella altra regla.
- o bé una altra llista (*[TRACTAMENT]*) amb més definicions de tractament a dins. Això permet tractar elements que són dintre d'altres elements d'una llista.

Es poden combinar una VARIABLE amb algun dels items nombrats anteriorment afegint un operand d'IGUAL (=) entre els dos. Per exemple:

```
VARIABLE = LocalCode
```

o bé

```
VARIABLE = REGLA
```

o bé

```
VARIABLE = [ ELEMENT : ELEMENT ]
```

Això farà que s'instancïi la variable amb el que pertoqui i, a més, s'executi el tractament que indiqui el símbol, regla o llista. Si, per exemple, volgéssim afegir tota una llista a una variable i a la vegada tractar-la amb una altra regla, la sintaxi seria la següent:

```
REGLA1 = [ var1 = REGLA2 ];
```

Aquest exemple farà que s'agafi el primer element de la *REGLA1*, es posi a la variable *var1* i se li apliqui la *REGLA2*.

Per a treballar llistes amb el nostre llenguatge hem definit dos operands bàsics, aquests es trobaran entre els elements de les llistes.
Són els següents:

DOS_PUNTS (*:*) Separa un element d'un altre element de la llista.

TAIL (|) Aquest operador retorna la cua de la llista a partir de l'element on estiguem.

Al codi de control hi podem trobar els següents modificadors:

Self Es tracta com una variable que refereix a la llista d'entrada de la regla. Es fa servir per si volem executar alguna funció sobre l'entrada de la regla en el codi de control.

Return Fa que l'expressió que el segueix sigui la que desmembri el tractament de la regla.

Un exemple d'aquests dos modificadors és el següent:

```
REGLA1(Return adecrypt(Self);) =
       [ var1 : REGLA2 | var2 ];
```

Aquest exemple aplicarà una desencriptació asimètrica sobre l'itinerari d'entrada (*adecrypt(Self);*) i, gràcies al *Return*, serà la llista que tracti la regla.

Fins ara hem vist com es descriu el tractament de l'itinerari d'entrada amb l'especificació de l'arquitectura d'extracció. Ara anem a veure la sintaxi que descriu com es generen els itineraris per a les següents plataformes.

Reconstrucció dels itineraris de sortida

La finalitat de la reconstrucció és poder definir com serà l'itinerari que enviarem a les següents plataformes on salti l'agent. Al igual que el desmembrament, la sintaxi de la reconstrucció també es fa amb regles, però només amb una, la regla NEXTITINERARI.

Cal a dir que la intenció és que es generin tants itineraris com elements hi hagi a la variable NextAgencyList, així que si la llista és buida (no hi ha plataformes de destí) no es generarà cap itinerari. És possible que els itineraris que es generin siguin iguals per a cada plataforma de destí, això depèn de qui dissenya l'arquitectura.

La reconstrucció es basa en una regla anomenada NEXTITINERARI que té una sintaxi una mica diferent que les regles de desmembrament.

```
NEXTITINERARI = RECONSTRUCCIO ;
```

On:

RECONSTRUCCIO conté tots els elements que tindran els itineraris de les següents plataformes.
La seva sintaxi és molt simple, n'hi ha prou amb anar afegint variables, prèviament instanciades a les regles de desmembrament, a una llista buida que serà l'itinerari resultant. A aquestes variables se'ls poden aplicar funcions externes per adequar-les a l'arquitectura que les tracta. Per exemple:

```
NEXTITINERARI = itinerari ;
```

o bé

```
NEXTITINERARI = [ @variable1 :  aencrypt(variable2) |
     restallista(variable3) ];
```

Anem a veure les operacions bàsiques de la reconstrucció:

DOS PUNTS (:) Afegeix un element a l'itinerari.

TAIL (|) Afegeix tots els elements de la llista que el segueix, un a un, a l'itinerari.

Als elements de la reconstrucció hi podem trobar els següents modificadors:

FOREACH (@) A diferència dels dos anteriors, aquest operador no el trobem entre dos items, sinó just davant d'un d'ells modificant-lo. El FOREACH sel·lecciona un element de la llista que precedeix per a cada itinerari que creï. Per exemple:

```
NEXTITINERARI = @llista_de_itineraris ;
```

Aquest exemple retorna una llista d'itineraris on cadascun d'ells és un element de la llista_de_itineraris.

COUNT (#) Aplicat a una variable, en retorna el número d'elements.

3.3 Resum

En aquest capítol hem fet l'anàlisi i disseny del nostre llenguatge. Per començar hem vist les diferents alternatives de llenguatges que podíem crear per tal de fer l'extracció dels itineraris. Després de descartar-ne les altres alternatives, hem decidit fer el llenguatge amb una sintaxi semblant a la sintaxi que té el llenguatge de protecció d'itineraris.

Seguidament hem passat a la descripció del llenguatge que hem dissenyat remarcant dues parts diferenciades, la desmembració de l'itinerari d'entrada i la reconstrucció d'un o més itineraris per a les següents plataformes.

En el següent capítol tractarem els passos que hem seguit per al disseny i la implementació del compilador d'arquitectures d'extracció.

Capítol 4

Disseny i Implementació del Compilador

En els capítols precedents hem introduït la necessitat de crear un compilador d'arquitectures d'extracció i hem enfocat com fer-ho. Després d'un anàlisi del llenguatge de protecció d'itineraris, hem dissenyat com haurà de ser el nostre llenguatge d'extracció i l'hem descrit.

Aquest capítol es centrarà en el disseny i la implementació del compilador en totes les seves fases, començant per les decisions prèvies que hem hagut de prendre, continuant per el disseny de la sintaxi i semàntica del compilador i finalitzant amb la generació del codi de la classe extractora resultant.

Abans de començar amb el desenvolupament com a tal del compilador, hem hagut d'escollir una eina de creació de compiladors per a implementar-lo. N'hem considerat diverses com ara ANTLR, JavaCC o CUP però, al final, ens hem decantat per fer-lo en JavaCC. No hi ha cap raó de pes per a haver escollit JavaCC en comptes d'algun dels altres, ja que totes les alternatives eren viables i la seva complexitat semblant. El que finalment ens ha fet decidir ha sigut que tant els tutorials com els exemples de JavaCC que hem consultat han sigut els més entenedors.

4.1 Fases del compilador

En aquesta secció seguirem, pas a pas, les quatre fases de creació d'un compilador (Figura 4.1).

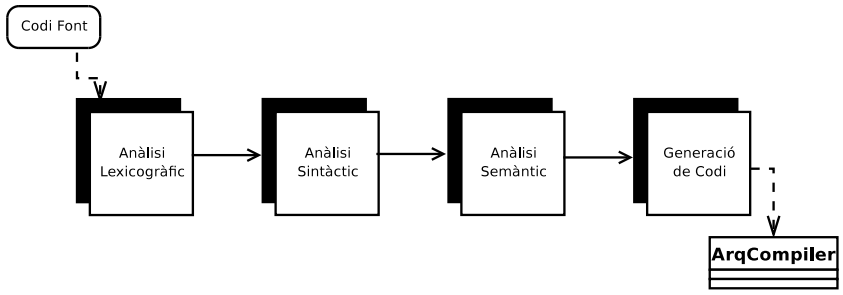

Figura 4.1: Fases del compilador

A l'anàlisi lexicogràfic veurem que alguns símbols o tokens estan definits en format EBNF, aquesta sintaxi inclou els operands `()+` i `()*`.
Quan trobem un `()+` significa que el que hi hagi entre els parèntesi apareixerà entre un cop i tants com faci falta. Quan trobem `()*` vol dir que el que hi hagi entre parèntesi tant pot no aparèixer com fer-ho els cops que faci falta. Dit això vegem cada fase detalladament.

4.1.1 Anàlisi Lexicogràfic

El primer pas per a crear un compilador és crear l'escànner que el parsejarà. És a dir, definir tots els possibles jocs de caràcters (tokens, símbols) vàlids que podem trobar.

La nostra definició de tokens és:

ESPAI: " " L'espai en blanc serà un caràcter d'*skip*. És a dir, quan el trobem l'ignorarem. Servirà per a separar els tokens.

EOL És el token d'*End of Line*. També podríem haver-lo posat com a caràcter d'*skip*, però no ho hem fet per a obligar que cada regla estigui en una línia separada.

EOF És el token d'*End of File*. Només apareixerà quan trobem el final de l'arxiu i serveix per marcar quan s'ha acabat de parsejar l'arquitectura.

"LocalCode", "NodeType", "HostName" i "NextAgencyList" representen les variables importants de l'extracció.

"LOCALITINERARI" És el nom de la primera regla del nostre compilador. És la que comença el procés de desmembrament i tractament de l'itinerari.

"NEXTITINERARI" És el nom de l'ultima regla del nostre compilador. És la que s'ocupa de la reconstrucció de l'itinerari per a les següents plataformes.

REGLA: (["A"-"Z"])+ Aquest token defineix qualsevol altra regla de l'arquitectura. És una paraula alfabètica amb tots els caràcters en majúscula.

ID : (["a"-"z"]) (["A"-"Z","a"-"z","0"-"9","_"])* Representa el nom de variables i funcions. És una paraula que comença en una lletra minúscula i ve seguida de lletres alfanumèriques tant majúscules com minúscules o per un GUIÓ BAIX (_).

PO : "(" És el parèntesi obert. S'utilitza per començar el codi de control de les regles i la llista de paràmetres d'una funció.

PT : ")" És el parèntesi tancat. S'utilitza per finalitzar el codi de control de les regles i la llista de paràmetres d'una funció.

CO : "[" És el claudàtor obert. Indica inici de llista a les regles.

CT : "]" És el claudàtor tancat. Indica el final d'una llista a les regles.

COMA : "," La coma separa paràmetres en la crida de funcions.

PUNTCOMA : ";" Serveix per a marcar el final d'una regla i per a separar instruccions del codi de control.

DOSPUNTS : ":" Fa d'operand d'unió entre dos elements d'una llista.

HEADTAIL : "|" Fa d'operand d'unió entre un element d'una llista i tots els que el segueixen.

IGUAL : "=" Serveix tant per separar el Nom i Codi de control d'una regla, del seu tractament com per fer assignacions de llistes a variables.

FOREACH : "@" Aquest caràcter, davant d'una variable o funció a la regla NEXTITINERARI, indica que cada itinerari creat tindrà un element diferent de la variable en qüestió.

COUNT : "#" Aquest caràcter, juntament amb una variable darrera, retorna el número d'elements que té una variable.

"Self" serveix per referir-se a la llista d'entrada en el codi de control d'una regla.

"Return" fa que s'enviï la llista que el segueix al tractament de la regla.

"Null" Representa una llista buida.

4.1.2 Anàlisi Sintàctica

Ara que ja tenim descrits els tokens que ens podem trobar, hem de definir en quin ordre els volem. D'això se n'encarrega l'Anàlisi Sintàctica.

L'Anàlisi sintàctica del nostre compilador esta escrita en BNF i comença amb la regla `<start>`.

```
<start> ::= { <regla> | EOL } EOF
```

La primera regla de la nostra gramàtica diu que aquesta estarà formada per `<regla>`s intercalades amb *Ends of Line*, el final de la nostra gramàtica serà un *End of File*.

Les regles de la nostra gramàtica tindran la següent sintaxi:

```
<regla> ::= ( ( LOCALITINERARI | REGLA ) <destruccio> |
     NEXTITINERARI <construccio> ) PUNTCOMA EOL
```

Les regles seran o be de destrucció (o desmembrament de l'itinerari) o de construcció. Si son de destrucció, la seva sintaxi començarà pel token LOCALITINERARI o bé pel token REGLA i després la `<destruccio>`. En canvi, si la

regla que estem parsejant és de construcció trobarem el token NEXTITINERARI en primer lloc i després la <construccio>. Tant si hem fet construcció com destrucció, al final de cada regla sempre hi trobarem un PUNTCOMA i un EOL.

```
<destruccio> ::= [ PO ( [ RETURN ] ( SELF | ID [ <instruccio> ] )
     PUNTCOMA )+ PT ] IGUAL [ ID IGUAL ] CO <dadesdest> CT
```

La <destruccio> comença amb el Codi de Control de les regles, encara que aquesta part és opcional. Després del codi de control ve un IGUAL per separar Nom+CodiControl del tractament de la regla i, opcionalment, una assignació a variable.
Finalment parsegem un claudàtor obert per marcar que estem tractant una llista, després el <dadesdest> i finalment trobem un claudàtor tancat que indica que es tanca la llista anterior. Les instruccions del Codi de Control estan entre parèntesis (PO CodiCon PT) i el seu format és una repetició d'instruccions separades per PUNTCOMA. Les instruccions sempre comencen amb un *Return* opcional i després pot haver-hi un *Self*, un identificador ID i, opcionalment, la <instruccio> que fa que aquest identificador no sigui una variable sinó una funció.

<dadesdest>, que tracta les dades de la <destruccio>, comença amb una assignació a variable.

```
<dadesdest> ::= [ ID IGUAL ] ( CO <dadesdest> CT | LOCALCODE |
     NODETYPE | NEXTAGENCYLIST | HOSTNAME |  REGLA | ID )
     [ ( DOSPUNTS | TAIL ) <dadesdest> ]
```

Després de l'assignació hi pot haver:

- Una llista amb un altre <dadesdest> a dins.
- Algun dels tokens de variables globals (LOCALCODE, NODETYPE, NEXTAGENCYLIST, HOSTNAME).
- El nom d'una REGLA.

- Un identificador ID que serà una variable.

Seguidament podem trobar opcionalment un operand d'unió de llistes (DOSPUNTS o TAIL) i després un altre cop <dadesdest>.

La <construccio> comença amb un IGUAL i <dadesconst>.

```
<construccio> ::= IGUAL <dadesconst>
```

A <dadesconst> podem trobar:

- Una llista buida representada per un NULL.
- Una altra llista amb <dadesconst> a dins.
- Un FOREACH opcional seguit de LOCALCODE, NODETYPE, NEXTAGENCYLIST, HOSTNAME o bé ID amb una <instruccio> opcional.

Finalment podem trobar els operands de llista DOSPUNTS o TAIL i altre cop <dadesconst>.

```
<dadesconst> ::= (  NULL | CO <dadesconst> CT | [ FOREACH ]
     ( ID [ <instruccio> ] | LOCALCODE | NODETYPE |
     NEXTAGENCYLIST | HOSTNAME ) )
     [ ( DOSPUNTS | TAIL ) <dadesconst> ]
```

Les instruccions estan definides com un parèntesi obert, opcionalment una llista de <param> separats per COMA i al final de tot un parèntesi tancat.

```
<instruccio> ::= PO [ <param> { COMA <param> } ] PT
```

Els paràmetres comencen amb un COUNT opcional i després o un *Self*, o una ID amb una <instruccio> opcional.

```
<param> ::= [ COUNT ] ( SELF | ID [ <instruccio> ] )
```

Fins aquí hem vist la definició sintàctica del nostre llenguatge. Ara anem a veure les regles semàntiques que l'afecten.

4.1.3 Anàlisi Semàntica

Després de fer l'anàlisi lexicogràfic i definir la sintaxi del llenguatge s'han d'especificar uns requisits que ha de seguir la gramàtica per tal d'evitar que es puguin crear programes que no siguin coherents. Per exemple, no tindria sentit crear una arquitectura d'extracció on no hi hagués la regla LOCALITINERARI o usar variables a NEXTITINERARI que no haguessin estat creades amb anterioritat.

Per a implementar aquestes restriccions hem hagut de crear una classe d'excepcions, aquesta classe (SemanticException.java) llança excepcions del tipus adient segons l'error que trobem durant el parser, i en mostra un text explicatori.

En aquesta secció anem a veure les restriccions principals que hem fet servir per a fer la semàntica del compilador.

- Comprovar que totes les regles que es criden han estat declarades.
- Comprovar que totes les variables a les que s'accedeix hagin estat instanciades.
- Si una regla te Codi de Control, sempre haurà de contenir un *Return*. No n'hi poden haver més ni menys. Si no hi ha codi de control es sobreentendrà que es retorna *Self* automàticament (*Return Self*).
- Tant LOCALITINERARI com NEXTITINERARI són regles obligades a l'extracció.
- Sempre, com a mínim, una extracció ha de modificar el valor de les variables LocalCode, NodeType i NextAgencyList (encara que aquesta última pot ser una llista buida).
- A la regla NEXTITINERARI només hi pot haver un foreach.
- Dintre d'un àmbit d'una regla de desmembració només hi podem trobar una separació de *tail*, i a l'element que afecti serà l'últim de la llista. A la regla de reconstrucció NEXTITINERARI hi podem trobar tants *tail*s com siguin necessaris.

4.1.4 Generació de Codi

Un cop implementats l'anàlisi sintàctic i el semàntic, arriba el torn de la generació de codi. Depenent de com hagin anat els passos anteriors la generació crearà un codi o un altre. La nostra generació de codi crea una classe en llenguatge Java que és capaç d'extreure dades d'un itinerari. Aquesta generació l'hem implementat en un *string* al que se li va afegint el codi final a mesura que es parseja l'arquitectura d'entrada. Finalment s'escriu aquest *string* en un arxiu que serà la classe generada. Els punts principals de la generació de codi són:

- Crear capçalera de la classe i mètode inicial.
- Crear funcions auxiliars per a l'accés a variables públiques.
- Quan s'instancía l'objecte d'extracció s'inicialitzen les variables i es crida a la regla (funció) LocalItinerari passant-li com argument l'itinerari.
- Totes les regles declarades es generaran com una funció privada de la classe, el seu paràmetre serà una llista. Dintre d'aquesta funció s'hi crearan una llista temporal i un iterador sobre aquesta llista.
- El primer que s'ha de fer al entrar a una regla és executar el codi de control i tractar el resultat per a poder executar-la. Quan s'hagi executat el codi de control es copiarà el que hi hagi al *Return* a la llista temporal creada abans. Aquesta llista serà la que parsejarà el compilador.
- Si algun codi de control retorna *null* a la regla, aquesta regla no s'executa (es fa un *Return* al codi java). Això permet fer recursivitats en un itinerari per a buscar nodes candidats.
- Durant el parser de la regla, si trobem:
  - Variables globals: (LOCALCODE, HOSTNAME, NODETYPE, NEXTAGENCYLIST) es tractaran directament.
  - Variables locals: (tipus it a la *nested* [RMB02]) es tractaran mitjançant un HashMap (una llista de tuples de nom_de_variable i valor).

  - Regles: es cridarà a la funció d'aquella regla amb argument el token que s'estigui parsejant (només a la fase de desmembració).
  - Una llista: crearem una llista temporal i un iterador sobre aquesta amb el seu valor per a que es pugui recórrer recursivament.

- L'operand de la regla DOSPUNTS es tractarà incrementant la variable i del `.get(i)` en una posició. Si trobem l'operand TAIL farem un `.subList()` de la llista des de la posició actual fins al final.

Ara que ja hem acabat d'escriure el compilador, vegem la relació de classes en que s'ha quedat.

4.2 Classes resultants

Un cop implementat i compilat el codi del compilador, les classes que en resulten són:

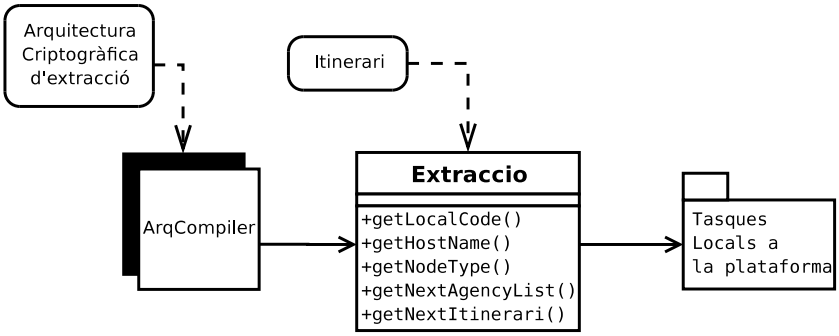

Figura 4.2: Fases d'execució del projecte

Classe del Compilador

Al compilar el compilador es genera la classe ArqCompiler (Compilador d'arquitectures). Aquesta classe és generada automàticament per el JavaCC a través de la sintaxi, semàntica i generació de codi. Els paràmetres d'entrada d'aquesta classe

són l'arxiu on s'escriurà la classe extractora i l'arquitectura criptogràfica d'entrada. L'execució d'aquesta classe va parsejant l'arquitectura d'extracció i generant una classe que és capaç d'extreure-la.

Classe d'Excepcions Semàntiques

Aquesta classe serveix per complementar les comprovacions semàntiques del compilador. Els tipus d'excepció que pot llançar són els següents.

REGNODEF S'intenta cridar a una regla no definida.

DOSRET Es defineixen dos *Return* en el mateix codi de control.

REDEF S'intenta instanciar una variable global (LocalCode, HostName, etc) que ja ha estat instanciada.

VARNODEF S'intenta accedir a una variable no instanciada.

DOSFORE Hi ha dos FOREACH (@) en el NEXTITINERARI.

DOSTAIL Hi ha dos TAIL (|) en una regla de desmembrament.

DUPR Es declara una regla duplicada.

NOFORE S'intenta fer un FOREACH (@) amb una variable no compatible.

NONEXTIT S'intenta cridar la regla NEXTITINERARI des d'un lloc on no és possible fer-ho.

Tant aquesta classe com la classe del compilador tan sols són necessàries quan s'està compilant l'arquitectura criptogràfica. Un cop obtinguda la classe d'extracció no és necessari afegir-les a l'agent.

Classe Criptogràfica

Aquesta classe s'ocupa d'implementar les funcions criptogràfiques bàsiques sobre llistes. La implementació que hem dut a terme ha sigut un prototip d'una classe

criptogràfica. Ho hem fet així per tal de ser capaços d'escriure un itinerari, protegit i encriptat, com un string i poder-lo llegir, ja que si la nostra llibreria encriptés de veritat l'itinerari només veuríem caràcters ascii sense sentit.

Les funcions que implementa la classe són:

aencrypt Implementa una encriptació amb clau asimètrica. A la nostra implementació es conforma afegint la llista d'entrada en una estructura del tipus *["assym_enc", dades]*

adecrypt Implementa una desencriptació amb clau asimètrica. A la nostra implementació, aquesta funció, comprova que el primer element de la llista sigui el string "assym_enc" i en retorna el segon element.

sencrypt Implementa una encriptació amb clau simètrica. Al igual que l'encriptació asimètrica, la nostra implementació posa la llista d'entrada a l'estructura *["sym_enc", dades]*.

sdecrypt Implementa una desencriptació amb clau simètrica. La nostra implementació comprova que el primer element de la llista sigui "sym_enc" i en retorna el segon element.

Tant aquesta classe com la de funcions Externes és necessari que siguin accessible des de la classe Extractora.

Classe de Funcions Externes

Per tal de donar més llibertat als programadors d'agents en la confecció de les seves arquitectures, és possible cridar funcions externes tant des del codi de control de les regles com des del tractament del NEXTITINERARI. Això permet que aquest compilador s'ajusti a una varietat molt gran d'arquitectures.

Algun exemples de funcions externes que hem afegit són:

select_nhead Retorna els *n* elements del començament d'una llista.

select_ntail Retorna tota la llista d'entrada menys els primers *n* elements.

Classe Extractora

Un cop compilada l'arquitectura d'extracció (un arxiu IPL) amb el compilador d'arquitectures (ArqCompiler) n'obtenim una classe extractora. Aquesta operació la podem veure a la Figura 4.2.

La classe extractora que genera el compilador nomes serveix per a tractar l'arquitectura per a la qual ha sigut dissenyada. Si provem d'extreure un itinerari compilat amb una arquitectura que no és la mateixa amb la que s'ha creat la classe extractora es produiran errors.

Els mètodes públics d'aquesta classe són:

getLocalCode Retorna el codi local a executar-se a la plataforma actual.

getNodeType Aquests mètode retorna el tipus de node on som. Serveix per a saber com s'ha de migrar.

getHostName Retorna el *hostname* de la màquina on som si l'arquitectura la defineix. Normalment s'usa per fer comprovacions de seguretat o sel·lecció de nodes.

getNextAgencyList Retorna una llista amb les agències que són candidates per a migrar.

getNextItinerari La llista que retorna són els itineraris que s'haurien de propagar a cadascuna de les plataformes de la llista de *NextAgencyList*.

4.3 Resum

Després d'haver dissenyat, en el capítol 3, com seria el llenguatge, en aquest capítol ens hem centrat en la seva implementació.

Hem començat definint quins símbols podíem trobar i els hem descrit amb detall, seguidament hem definit la sintaxi amb que esta definit el llenguatge i ho hem completat amb la llista les regles semàntiques que necessitem per tal d'evitar que el nostre compilador accepti incoherències.

Finalment hem descrit els punts principals de la generació de codi per tal de que es generi de forma correcta una classe que sigui capaç d'extreure dades d'un itinerari.

Al final de tot hem fet una petita descripció de les classes resultants de la nostra implementació com són les llibreries criptogràfiques, les de funcions externes, etc.

Capítol 5

Proves

Fins ara hem basat la memòria en la fase de disseny i implementació del compilador, ara que aquesta feina ja està enllestida ens centrarem en les proves que hi hem dut a terme.
Per tal d'anar provant el compilador en les seves diverses fases, hem adaptat dues arquitectures ja existents, com són l'arquitectura *simple* i l'arquitectura *nested* (també anomenada RMB [RMB02]), al nostre llenguatge d'extracció. A cada fase del compilador hem compilat diverses vegades aquestes arquitectures, afegint cada cop algun canvi en la seva estructura per tal de comprovar el seu correcte funcionament i la detecció d'errors.

Per a cadascuna de les dues arquitectures hem fet servir dos itineraris que posen a prova la majoria de possibilitats que ens podem trobar, són els següents:

L'itinerari 1, com podem veure a la Figura 5.1, és un itinerari molt simple. Es composa únicament de 3 nodes seqüencials. Comença a la `maq1`, després va cap a la `maq2` i finalitza el seu recorregut a la `maq3`.

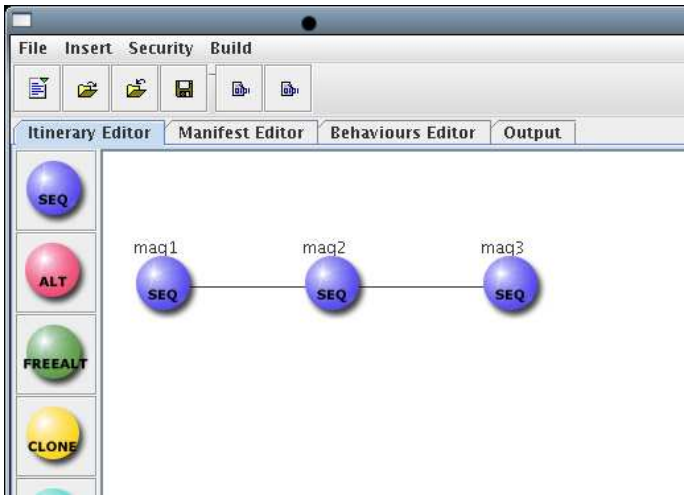

Figura 5.1: Itinerari 1

L'itinerari 2 (Figura 5.2), comença per el node `maq1`.
Aquest primer node veiem que és un node alternativa i que les seves alternatives són anar a la maq2 i a la maq3. Després d'aquest primer node venen les dues alternatives que té. Una és anar a la `maq2` i després a la `maq4` de forma seqüencial per acabar anant al node d'unió de la `maq6`, l'altre alternativa és anar a la `maq3` i després a la `maq5` per acabar també a la `maq6`.

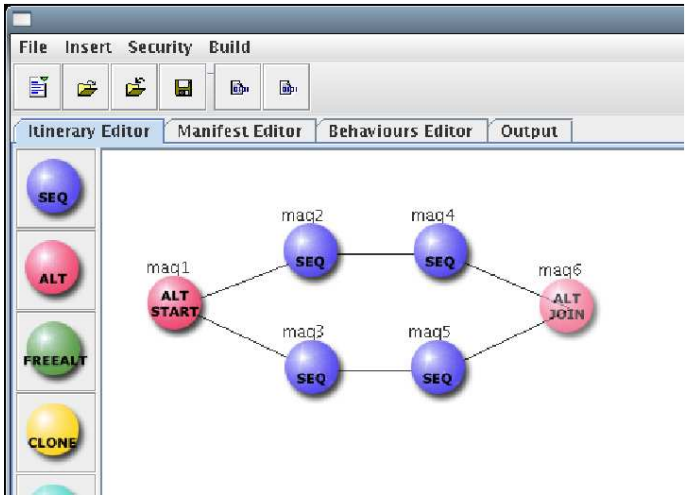

Figura 5.2: Itinerari 2

A continuació descriurem una mica les dues arquitectures que hem fet servir a les proves i els resultats que n'hem obtingut.

5.1 Arquitectures d'Agents Mòbils

Per a fer les proves sobre aquests dos itineraris farem servir dues arquitectures d'agents mòbils. Aquestes són l'arquitectura *simple* i l'arquitectura *nested*.
Quan fem una extracció suposarem que ho hem fet a la `maq1`.

5.1.1 Arquitectura *simple*

L'arquitectura *simple*, com el seu propi nom indica, és l'arquitectura més simple que podem trobar. La seva execució empaqueta les dades de l'itinerari en nodes, agrupant-los amb els nodes predecessors i successors. Aquesta arquitectura no aplica cap tipus d'encriptació criptogràfica sobre les dades, per el que podem dir que no és una arquitectura que protegeixi l'itinerari. Aquesta podria ser l'arquitectura per defecte en projectes que no necessitin seguretat ni moure dades importants.

L'IPL d'aquesta arquitectura és:

```
ITINERARI = [ NODE#i...e ] ;
NODE{SEQ} = [ LocalCode#i : "seq" : HostName#sf ] ;
NODE{ALT_START} = [ LocalCode#i : "alt" :
    HostName#sf...sl ] : ITINERARI#sf...sl ;
NODE{CLONE_START} = [ LocalCode#i : "clone" :
    HostName#sf...sl ] : ITINERARI#sf...sl ;
NODE{JOIN_START} = [ LocalCode#i : "join" : HostName#sf ] ;
```

L'IPL de l'extracció que hem proposat el podem veure a la Figura 5.3

```
LOCALITINERARI = [ [ LocalCode : NodeType : nal =
     NextAgencyList ] | nextnodes ];
NEXTITINERARI = [ Null | @select_nhead(#nal,nextnodes) |
     select_ntail(#nal,nextnodes) ];
```

Figura 5.3: Arquitectura *simple* escrita en IPL d'extracció.

A continuació anem a provar els dos itineraris amb aquesta arquitectura:

Itinerari 1

Comencem amb l'itinerari 1.
Aquest itinerari, un cop compilat amb l'arquitectura *simple*, queda com la següent llista:

```
[[CodiLocal1, SEQ, [maq2]], [CodiLocal2, SEQ, [maq3]],
  [CodiLocal3, SEQ, []]]
```

Un cop aplicada la nostra classe extractora sobre aquesta llista n'obtenim les dades de la Figura 5.4

```
Codi Local: CodiLocal1
Host Name:
Node Type: SEQ
Next Agency List: [maq2]
Next Itinerari List:
- [CodiLocal2, SEQ, [maq3], [CodiLocal3, SEQ, []]]
```

Figura 5.4: Dades obtingudes de l'itinerari1 amb l'arquitectura *simple*

Com podem veure, la classe extractora ha detectat *CodiLocal1* com el codi local a executar a la plataforma actual, també ha detectat que estem en un node seqüencial i que la següent plataforma on ha de migrar és la `maq2`. També ha generat l'itinerari que s'haurà de propagar en aquesta migració. Es pot veure que el HostName esta buit, això no és cap error ja que, com hem dit en capítols anteriors, no totes les arquitectures el fan servir i tant la *simple* com la *nested* en son un exemple.

A continuació farem el mateix amb l'itinerari 2

Itinerari 2

Aquest itinerari és més complicat que el següent, un cop compilat amb l'arquitectura *nested* queda com la següent llista:

```
[[CodiLocal1, ALT_START, [maq2, maq3]], [[CodiLocal2,
  SEQ, [maq4]], [CodiLocal4, SEQ, [maq6]]], [[CodiLocal3,
  SEQ,[maq5]], [CodiLocal5, SEQ, [maq6]]], [CodiLocal6,
  ALT_JOIN, []]]
```

Al aplicar aquest itinerari a la nostra classe extractora obtenim la Figura 5.5

```
Codi Local: CodiLocal1
Host Name:
Node Type: ALT_START
Next Agency List: [maq2, maq3]
Next Itinerari List:
- [[CodiLocal2, NOSE_SEQ, [maq4]], [CodiLocal4,
    NODE_SEQ, [maq6]], [CodiLocal6, ALT_JOIN, []]]
- [[CodiLocal3, NODE_SEQ, [maq5]], [CodiLocal5,
    NODE_SEQ, [maq6]], [CodiLocal6, ALT_JOIN, []]]
```

Figura 5.5: Dades obtingudes de l'itinerari2 amb l'arquitectura *simple*

A diferència d'amb l'itinerari 1, el tipus de node que detectem aquí és un *ALT_START*, que significa que estem en un node d'alternativa, les agències on podem migrar són `maq2` i `maq3`.
També es pot veure que l'extracció ha generat dos itineraris de sortida, un per a cada plataforma de destí.

5.1.2 Arquitectura *nested*

L'arquitectura *nested* va ser ideada al 2002 per en S. Robles, en J. Mir i en J. Borrell [RMB02].
Va més enllà de l'arquitectura *simple* i aplica encriptació a les dades dels nodes. Aquesta arquitectura també es caracteritza per que cadascun dels nodes està dins del node que el precedeix, això li dona el nom de *nested* (niuada) i per aquesta raó també és anomenada la Ceba (per les capes). Aquesta forma d'organitzar els nodes ajuda a que un node no pugui saber quants nodes s'han executat abans d'arribar l'execució al punt actual. Cadascun dels nodes d'aquesta arquitectura està xifrat asimètricament amb la clau pública de la plataforma on s'ha d'executar. Això permet que un node no pugui saber quants nodes manquen per executar-se i,

encara menys, el codi que s'hi ha d'executar.

La descripció en llenguatge IPL d'aquesta arquitectura és:

```
ITINERARI = [ NODE#i ] ;
NODE{SEQ} = aencrypt( ki, [ [ localCode#i : "seq" :
    hostName#j ] : NODE#sf ]) ;
NODE{ALT_START} = aencrypt( ki, [ [ localCode#i : "alt" :
    hostName#sf...sl ] : ITINERARI#sf...sl : NODE#j ] ) ;
NODE{CLONE_START} = aencrypt( ki, [ [ localCode#i :
    "clone" : hostName#sf...sl ] : ITINERARI#sf...sl :
    NODE#j ] ) ;
NODE{CLONE_JOIN} = aencrypt( ki, [ [ localCode#i :
    "join" : hostName#j ] : NODE#j ]);
```

L'IPL d'extracció que hem proposat és el que es pot veure a la Figura 5.6.

```
LOCALITINERARI(Return adecrypt(Self);) = [ [ LocalCode :
     NodeType : NextAgencyList ] | nextit ] ;
NEXTITINERARI = [ Null | @nextit ];
```

Figura 5.6: Arquitectura *nested* escrita en IPL d'extracció.

A continuació anem a provar els dos itineraris amb aquesta arquitectura:

Itinerari 1

Aquest itinerari, un cop compilat amb l'arquitectura *nested* queda com la següent llista:

```
[assym_enc, [[CodiLocal1, SEQ, [maq2]], [assym_enc,
  [[CodiLocal2, SEQ, [maq3]], [assym_enc, [[CodiLocal3,
  SEQ, [][][][][][][][]
```

Fixem-nos que una estructura del tipus `[assym_enc, LLISTA]` significa que aquella llista està encriptada amb criptografia asimètrica.

El resultat de l'extracció és el que es pot veure a la Figura 5.7

```
Codi Local: CodiLocal1
Host Name:
Node Type: SEQ
Next Agency List: [maq2]
Next Itinerari List:
- [assym_enc, [[CodiLocal2, SEQ, [maq3]], [assym_enc,
    [[CodiLocal3, SEQ, [][][][][][]
```

Figura 5.7: Dades obtingudes de l'itinerari1 amb l'arquitectura *nested*

Ens ha detectat quin codi local executar i, com ha passat amb l'arquitectura *simple*, ha detectat que ha de migrar de forma seqüencial a la plataforma `maq2`. També ens ha extret l'itinerari que s'haurà de passar a la següent plataforma.

Itinerari 2

L'itinerari 2 compilat amb l'arquitectura *nested* és:

```
[assym_enc, [[CodiLocal1, ALT_START, [maq2, maq3]],
  [assym_enc, [[CodiLocal2, SEQ, [maq4]], [assym_enc,
  [[CodiLocal4, SEQ, [maq6]], [assym_enc, [CodiLocal6,
  ALT_JOIN, [][][][][][][][]], [assym_enc, [[CodiLocal3, SEQ, [maq5]],
  [assym_enc, [[CodiLocal5, SEQ, [maq6]], [assym_enc,
```

```
[CodiLocal6, ALT_JOIN, [][][][][][][][]]
```

El resultat de l'extracció és el següent (Figura 5.8)

```
Codi Local: CodiLocal1
Host Name:
Node Type: ALT_START
Next Agency List: [maq2, maq3]
Next Itinerari List:
- [assym_enc, [[CodiLocal2, SEQ, [maq4]], [assym_enc,
    [[CodiLocal4, SEQ, [maq6]], [assym_enc, [CodiLocal6,
    ALT_JOIN, [][][][][][]]
- [assym_enc, [[CodiLocal3, SEQ, [maq5]], [assym_enc,
    [[CodiLocal5, SEQ, [maq6]], [assym_enc, [CodiLocal6,
    ALT_JOIN, [][][][][][]]
```

Figura 5.8: Dades obtingudes de l'itinerari2 amb l'arquitectura *nested*

El tipus de node que ha trobat és *ALT_START* i la generació del següents itineraris n'ha retornat dos, un per a cada plataforma on pot migrar (`maq2` i `maq3`).

5.2 Resum

En aquest capítol hem presentat dues implementacions de les arquitectures *simple* i *nested* en el nostre llenguatge d'extracció de tasques locals i les hem compilat amb el compilador que vam dissenyar i implementar als capítols anteriors. Sobre aquestes classes resultants hem aplicat un itinerari creat per nosaltres i hem observat que la nostra classe extractora ha extret bé les tasques locals de l'itinerari.

Capítol 6

Conclusió

En aquest projecte hem desenvolupat un compilador d'arquitectures criptogràfiques d'extracció que genera classes extractores de tasques locals per itineraris d'agents mòbils.

El primer que vam fer al començar el projecte va ser un estudi dels camps que hi havia oberts per a desenvolupar nous projectes. Aquesta feina ens va portar a un projecte que es va dur a terme l'any passat i que compilava agents mòbils amb itineraris protegits per arquitectures criptogràfiques.
Aquell projecte, un cop acabat, forma part d'un entorn de desenvolupament integrat per a poder dissenyar, crear, llançar i recollir agents mòbils, tot des d'un sol programa.

La creació d'aquest projecte deixava un buit en quant a poder extreure aquest itinerari protegit, aquest buit va ser el nostre objectiu general. Per aquesta raó vam pensar en implementar una ampliació del llenguatge existent de protecció d'itineraris (IPL).

Després d'aquest estudi vam començar amb el disseny dels requeriments del nostre nou llenguatge. Vam decidir que la seva sintaxi seria molt semblant l'original. A continuació vam fer la implementació del compilador, descrivint acuradament cadascuna de les seves quatre fases (anàlisi lexicogràfica, sintàctica, semàntica i generació de codi).

El projecte l'hem acabat provant el compilador amb itineraris i arquitectures

reals. Les proves les hem fet amb les arquitectures *simple* i *nested* i amb un itinerari seqüencial i un altre amb alternatives.

Grau d'assoliment d'objectius

La nostra ampliació del IPL, que nosaltres anomenem IPL d'extracció, compleix l'objectiu que ens havíem proposat, és capaç d'extreure tasques de qualsevol arquitectura escrita en llenguatge IPL. També compleix un objectiu que ens vam proposar durant la realització del projecte com és la semblança entre la sintaxi de definició i la d'extracció d'arquitectures.

Un altre dels objectius era el de crear un compilador d'aquesta ampliació del llenguatge per tal de poder passar d'una descripció d'extracció d'arquitectures criptogràfiques a una classe que fos capaç d'extreure les tasques locals d'un itinerari respecte de la plataforma on es trobés. Aquest objectiu també l'hem assolit satisfactòriament, començant per l'anàlisi lexicogràfic i seguint pel sintàctic, semàntic i finalment per la generació del codi de la classe.
Un dels objectius menors que també ens havíem proposat va ser integrar el nostre compilador a la IDE de desenvolupament d'agents mòbils, malauradament aquest objectiu l'hem hagut de deixar de banda per a dedicar més temps a la implementació i a les proves amb itineraris reals. Finalment, aquest compilador l'hem provat amb diversos jocs de proves i ha detectat tots els errors que li hem implementat i, amb les arquitectures *simple* i *nested*, hem obtingut els resultats esperats.

Línies de Continuïtat

Al finalitzar el nostre projecte, la IDE de desenvolupament d'agents ja te al seu abast un compilador d'arquitectures d'extracció fàcil d'usar degut a les seves similituds al llenguatge original de creació d'arquitectures. Aprofitant aquesta feina és podrien dur a terme ampliacions al projecte.

La primera ampliació que es podria fer seria implementar una llibreria de funcions estàndard per a usar en el codi de control i en la reconstrucció de l'itinerari

de les arquitectures d'extracció. Durant el projecte, n'hem proposat un parell com són *select_nhead* i *select_ntail* però, per tal de facilitar la feina del programadors d'agents, es podrien arribar a afegir moltes més funcions útils a aquesta llibreria.

Un altra tasca relacionada amb aquest projecte que és podria dur a terme és la implementació de la llibreria de funcions criptogràfiques. A la nostra implementació hem fet un prototip amb les funcions més usuals del que hauria de ser la llibreria final, però faltaria implementar aquestes funcions i afegir-ne de noves.

Finalment també seria interessant implementar l'arquitectura d'extracció del *scrambled* [MB03] per al nostre compilador, però aquesta feina haurà d'esperar a que hi hagi una bona implementació d'aquesta arquitectura per al llenguatge IPL.

Bibliografia

[RMB02] S. Robles, J. Mir and J. Borrell. MARISM-A: An Architecture for Mobile Agents with Recursive Itinerary and Secure Migration. In *2nd. IW on Security of Mobile Multiagent Systems*. Bologna, July 2002.

[MB03] J. Mir and J. Borrell. Protecting Mobile Agent Itineraries. In *Mobile Agents for Telecommunication Applications (MATA). Springer Verlag, vol. 2881* of Lecture Notes in Computer Science, 275-285. October 2003.

[Gar03] C. Garrigues. Disseny i implementació de dues arquitectures d'agents mòbils per a la plataforma JADE/MARISM-A. Projecte Final de Carrera de la Universitat Autònoma de Barcelona. 2003.

[Gar05a] J. Garcia. Compilador d'agents mòbils amb itineraris protegits per arquitectures criptogràfiques. Projecte Final de Carrera de la Universitat Autònoma de Barcelona 2005.

[Gar05b] C. Garrigues. Simplificació del desenvolupament d'agents mòbils basats en arquitectures criptogràfiques. Treball de Recerca de la Universitat Autònoma de Barcelona. 2005.

[SS05] X. Sánchez i G. Sánchez. Apunts de l'assignatura Compiladors I del curs 2005-2006.

[Rob05] S. Robles. Apunts de l'assignatura Xarxes de Computadors II del curs 2005-2006.

[Nor05] Theodore S. Norvell. The JavaCC FAQ and Tutorial. Faculty of Engineering and Applied Science. Memorial University, Newfoundland, Canada.
http://www.engr.mun.ca/∼theo/JavaCC-FAQ
http://www.engr.mun.ca/∼theo/JavaCC-Tutorial

Firmat: Xavier Franquet Conte

Bellaterra, Juny de 2006

Resum

En aquest projecte s'ha realitzat una ampliació del llenguatge de protecció d'itineraris d'agents mòbils i la seva implementació per tal d'extreure'n les tasques locals. Aquestes tasques es troben encriptades dintre de l'itinerari de l'agent i descriuen el comportament que tindrà, així que és necessària una classe que les sàpiga extreure. El nostre compilador crea aquesta classe a partir de l'arquitectura d'extracció. L'hem provat amb les arquitectures *simple* i *nested* i els resultats amb itineraris reals han estat els esperats. A la pràctica, aquest projecte completa una mica més l'entorn de desenvolupament integrat d'agents mòbils MARISM-A.

Resumen

En este proyecto se ha realizado una ampliación del lenguaje de protección de itinerarios de agentes móviles y su implementación para extraer las tareas locales de estos. Estas tareas se encuentran encriptadas dentro del itinerario del agente y describen el comportamiento que tendrá, por lo tanto es necesaria una clase que las sepa extraer. Nuestro compilador crea esta clase a partir de la arquitectura de extracción. Lo hemos probado con las arquitecturas *simple* y *nested* y los resultados con itinerarios reales han sido los esperados. En la práctica, este proyecto completa un poco más el entorno integrado de desarrollo de agentes móviles MARISM-A.

Abstract

In this project we have carried out an extension of the mobile agents itinerary protection language and it's implementation to extract the local tasks. These tasks, which describe the behaviour of the agent in a given platform, are encrypted within the agent itinerary. There needs to be a class capable of extracting them. Our compiler creates this class from the extraction architecture. We have tested it with simple and nested architectures and the results with real itineraries were the expected. In practice, this project completes a little more the integrated development environment of MARISM-A mobile agents.