Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Specialistica in Informatica

DESIGN E IMPLEMENTAZIONE DI UN INTEGRATED DEVELOPMENT ENVIRONMENT PER IL LINGUAGGIO JOLIE

Tesi di Laurea in Paradigmi di Programmazione

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Diego Castronuovo

Parole chiave: JOLIE, IDE, Xtext, EMF, Joliepse

Sessione III
2009-2010

Alla mia famiglia:
per il supporto in questi
anni di studio...

Indice

Elenco delle figure

Acronimi utilizzati

IDE *Integrated Development Environment*

SOA *Service Oriented Architecture*

SOC *Service Oriented Computing*

OASIS *Organization for the Advancement of Structured Information Standards*

WS-BPEL *Web Services Business Process Execution Language*

W3C *World Wide Web Consortium*

XML *eXtensible Markup Language*

SOAP *Simple Object Access Protocol*

HTTP *Hypertext Transfer Protocol*

SMTP *Simple Mail Transfer Protocol*

WS *Web Service*

WSDL *Web Service Description Language*

UDDI *Universal Description, Discovery and Integration*

AWT *Abstract Window Toolkit*

SWT *Standard Widget Toolkit*

AST *Abstract Syntax Tree*

JOLIE *Java Orchestration Language Interpreter Engine*

DSL *Domain-Specific Language*

EBNF *Extended Backus-Naur Form*

BNF *Backus-Naur Form*

EMF *Eclipse Modeling Framework*

OMG *Object Management Group*

RCP *Rich Client Platform*

VSIP *Visual Studio Industry Partners*

JDT *Java Development Tool*

CDT *C/C++ Development Tool*

IMP *IDE Meta-tooling Platform*

DLTK *Dynamic Language Toolkit for Eclipse*

GMF *Graphical Modeling Framework*

NBS *NetBeans Scripting File*

ANTLR *ANother Tool for Language Recognition*

MDA *Model Driven Architecture*

XMI *XML Metadata Interchange*

MWE2 *Modelling Workflow Engine 2*

Capitolo 1

Introduzione

Il successo dei moderni linguaggi di programmazione dipende da molti fattori, uno dei quali è la disponibilità di strumenti di supporto per gli sviluppatori. *Editor*, *debugger*, *profiler*, strumenti di versionamento e di documentazione sono ormai divenuti indispensabili nel settore dell'ingegneria del software.

Tutti questi elementi sono parte integrante di un IDE, un applicativo modulare in grado di accompagnare il programmatore durante la fase d'implementazione del software.

Già dalla fine degli anni settanta la necessità di tale strumento divenne indispensabile. Un primo esempio di IDE è rappresentato dal *Cornell Program Synthesizer* [TeiRep81]: un *syntax-directed programming environment* in grado di creare, editare, effettuare debug e lanciare programmi. I programmi non sono visti come semplici testi ma come strutture sintattiche derivate dalla definizione formale della grammatica di un linguaggio. In questo modo è possibile fornire una correzione interattiva durante la scrittura del codice in modo da liberare il programmatore da frustranti "dettagli sintattici".

L'aggettivo *integrated* compare più avanti nel tempo e indica la capacità di gestire le interdipendenze fra documenti in modo da mantenere uno sviluppo incrementale dei progetti software: le modifiche ad un file si ripercuotono automaticamente su tutti i documenti collegati.

Verso la fine degli anni ottanta compaiono i primi sistemi che forniscono le linee guida alla base dei moderni IDE, PSG System [BahSne86] e CENTAUR [BCD+88]. Le caratteristiche fondamentali che possiamo individuare in questi due sistemi sono:

- un'interfaccia grafica unica che permetta di integrare tutti gli strumenti di supporto per l'intera fase del processo di sviluppo software;
- un'immediata risposta a ogni input del programmatore (segnalazione degli errori in primis);
- un modello formale di alto livello in cui rappresentare tutti i documenti di lavoro del programmatore, comunemente sotto forma di abstract syntax tree;
- un sistema uniforme di manipolazione di questo modello utilizzabile da tutti gli strumenti dell'IDE, in grado di massimizzare il riuso del codice.

Un ulteriore importante aspetto di PSG e CENTAUR è la genericità: è possibile creare tutti gli strumenti che compongono un IDE per qualsiasi linguaggio di programmazione in maniera semi-automatica, o meglio assistita, partendo dalla definizione formale di tale linguaggio. Possiamo quindi vedere i due progetti come *IDE-generator*. Questo concetto è ripreso venti anni più tardi dal framework Xtext utilizzato nella fase implementativa di questa tesi, dedicata allo sviluppo di Joliepse, un IDE per il linguaggio di programmazione JOLIE.

Durante gli anni novanta sono stati sviluppati centinai di IDE per diversi linguaggi di programmazione e aggiunte nuove funzionalità come navigazione del codice, funzioni di auto completamento, suggerimenti, analisi semantiche più o meno avanzate, sistemi di *refactoring* e tanto altro ancora, ma le caratteristiche comuni restano quelle delineate dai due progetti citati in precedenza.

Verso la fine degli anni novanta sono nati due fra gli IDE più conosciuti, apprezzati e utilizzati nel mondo *opensource*: Eclipse [Ecl10] e NetBeans [Net10]. La popolarità è dovuta al linguaggio Java per cui sono stati creati e alla predisposizione all'estendibilità. Allo stato dell'arte esistono infatti plug-in per ogni aspetto dello sviluppo software.

Un IDE rappresenta quindi un importante strumento per aumentare la produttività di programmatori professionisti, ma è anche un ottimo punto di partenza per coloro che si avvicinano per la prima volta a un particolare linguaggio. Un grande ostacolo iniziale per i novizi è dato, infatti, da quei "frustranti dettagli sintattici" descritti in [TeiRep81].

E' quindi evidente che un linguaggio di programmazione per il quale non sia disponibile un IDE può essere svantaggiato dal punto di vista della diffusione e dell'utilizzo.

Una panoramica sugli approcci esistenti nell'implementazione di IDE per nuovi linguaggi di programmazione è prensentata nel terzo capitolo di questa tesi.

Questa tesi nasce nell'ambito del progetto JOLIE [MGLZ06], un linguaggio di programmazione open source per SOA sviluppato presso l'Università di Bologna. JOLIE è un linguaggio innovativo e ancora in fase di sviluppo ma già utilizzato per applicazioni commerciali.

Il contributo al progetto fornito dal seguente lavoro è quello di di implementare un IDE per JOLIE in grado di semplificare il lavoro degli sviluppatori e di contribuirne alla diffusione. Un parte del lavoro di tesi è inoltre dedicata ad alcune moditiche apportate all'interprete del linguaggio, in cui il parser attuale, scritto a mano senza l'ausilio di generatori, è sostituito dal parser utilizzato nell' IDE, generato con uno strumento automatico a partire da una grammatica in *Extended Backus-Naur Form* (EBNF) per JOLIE.

JOLIE è un potente e versatile linguaggio creato per l'emergente paradigma di programmazione *service oriented*. Gli elementi principali di questo paradigma sono appunto i servizi, in grado di essere invocati e fornire funzionalità all'esterno tramite interfacce di rete. I servizi giocano il ruolo degli

oggetti nel paradigma *object-oriented*.

Il paradigma service oriented consiste quindi nella manipolazione di questi servizi per costruire sistemi distribuiti. Quest'operazione prende il nome di *orchestration*, e JOLIE può essere quindi considerato un *orchestrator language*.

L'orchestrator language più conosciuto e utilzzato è sicuramente WS-BPEL [BahSne86], mentre la tecnologia più utilizzata attualmente per implementare il paradigma service oriented è quella dei WSs [W3C03].

JOLIE ha una sintassi di alto livello *Java/C-like* in grado di fornire un approccio più "umano" all'orchestration rispetto a una notazione XML utilizzata per WSs e WS-BPEL . Un altro punto di forza è il framework formale [GLG+06] su cui si basa il linguaggio.

Il secondo capitolo della tesi è interamente dedicato al linguaggio JOLIE e al paradigma SOA.

La fase iniziale di questo lavoro è incentrata sulla ricerca degli strumenti necessari per la creazione di IDE per un linguaggio di programmazione già esistente.

Dopo un'analisi sullo stato dell'arte la scelta ricade sul già citato framework Xtext [MVS10] per Eclipse, un *language development framework* pensato per *Domain-Specific Language* (DSL) e linguaggi di programmazione *general purpose*. Xtext permette di generare un editor per un linguaggio partendo da una grammatica in notazione EBNF. Il parser utilizzato internamente da Xtext è ottenuto dal *parser generator* ANTLR [ParQuo95] integrato nel framework.

Basato sul framework EMF, Xtext permette di creare un IDE altamente estendibile con funzionalità simili a quelle disponibili in Eclipse per il linguaggio Java. Xtext rappresenta quindi un'interpretazione moderna del PSG System presentato in [BahSne86].

Il framework è descritto dettagliatamente nei capitoli tre e quattro.

Xtext può essere utilizzato in due modalità: all'interno della piattaforma Eclipse e in modalità stand-alone, in qualsiasi applicazione Java. Il lavoro di

tesi può essere diviso in due parti principali:

- una prima parte in cui è stata definita la grammatica EBNF per Xtext di JOLIE analizzando il parser attuale. Si è cercato in seguito di sostiutire il parser attuale di Jolie con il parser generato da ANTLR, eseguendo Xtext in modalità stand-alone all'interno dell'interprete di JOLIE. Quest' operazione e i risultati ottenuti sono descritti dettagliatamente nel sesto capitolo;

- un seconda parte durante la quale è stato implementato Joliepse, un IDE *eclipse-based* per il linguaggio JOLIE. Le fasi di design e implementazione sono descritte nel quinto capitolo.

Nell'ultimo capitolo di questa tesi sono elencati i possibili sviluppi futuri per l'IDE Joliepse e una serie di considerazioni personali sul lavoro svolto.

Capitolo 2

JOLIE e SOA

In questo capitolo saranno descritti e analizzati l'emergente paradigma Service Oriented e il linguaggio JOLIE, che rappresentano il contesto in cui è stata sviluppata questa tesi.

2.1 SOA e SOC

Con il termine SOC s'intende un nuovo paradigma di programmazione che utilizza i servizi come elementi fondamentali per comporre sistemi distribuiti dinamici, ai quali ci si può riferire con il termine SOA.

Le SOA sono quindi un insieme di servizi che cooperano per realizzare le funzionalità richieste, ma questa definizione non è sufficientemente esauriente.

Secondo l'*Organization for the Advancement of Structured Information Standards* (OASIS) [MLM+06], SOA è un paradigma per organizzare e utilizzare *capabilitie* che possono essere sotto il controllo di differenti domini di proprietà.

Con il termine capabilitie intendiamo la capacità di fare qualcosa da parte di un'entità.

Nel mondo reale, le entità (persone e organizzazioni) creano capabilitie per sopperire ai problemi e i bisogni di ogni giorno. Queste sono esposte

all'esterno verso altre entità per creare collaborazioni. Allo stesso modo sistemi informativi possono creare funzionalità da esporre all'esterno ad altri sistemi tramite interfacce di rete.

Il nostro sistema economico può essere visto come un'enorme SOA in cui milioni di entità collaborano fra loro. Un lavoratore (entità) fornisce una capabilitie all'azienda presso la quale lavora (un'altra entità). Per raggiungere il posto di lavoro ha bisogno di un'automobile e si rivolge a venditore di auto (entità) per acquistarla. Stipulato il contratto, il venditore si rivolge alla casa produttrice per ordinare l'automobile da consegnare al cliente. La casa produttrice si rivolge a produttori di componenti e ai lavoratori per assemblare l'auto, e così via. Quest'analogia rende l'idea di quanto possa essere complessa la reale implementazione di una SOA.

Sempre secondo il *reference model* OASIS, i tre concetti chiave del paradigma SOA sono:

- **visibilità**: indica la capacità di chi possiede una particolare caipability e del soggetto avente il bisogno corrispondente di "vedersi reciprocamente. Questo meccanismo è realizzato fornendo in maniera accessibile descrizioni dettagliate (*service description*) delle funzionalità offerte da un lato, e delle funzionalità richieste dall'altro. Tornando all'analogia precedente con il sistema economico, il sistema pubblicitario serve a questo scopo;

- **interazione**: una volta che un'entità richiedente un servizio scopre un'enità fornitrice della caipability richesta, occorre che questi due attori comunichino fra di loro. L'interazione riguarda appunto questo scambio d'informazioni;

- **effetto**: è la conclusione della collaborazione fra le due entità. Il servizio è stato fornito al richiedente.

Per cercare di comprendere ulteriormente le SOA occorre definire l'aspetto centrale, e cioè cosa s'intenda con il termine "servizio".

Servizio letteralmente significa "lavorare per qualcun altro" e, nel contesto SOA, è inteso come il meccanismo con il quale bisogni e caipability vengono in contatto.

L'entità che fornisce una caipability è definita come *service provider*, mentre la controparte che utilizza il servizio *service consumer*.

Secondo [PapHeu07], i servizi coinvolti nelle SOA devono essere "trovabili, invocabili e componibili" dinamicamente.

La trovabilità è strettamente legata al concetto di visibiltà definito nel *reference model* OASIS e comprende quindi un sistema che garantisca questa caratteristica. Questo sistema può essere rappresentato da *service registry*, tassonomie o ontologie che i service consumer possono interrogare per conoscere quale sia il provider più adatto alle proprie esigenze.

L'invocaibilità dei servizi è garantita da un sistema di comunicazione standardizzato basato su XML che garantisce l'indipendenza da piattaforme o protocolli di comunicazione. Ogni servizio è in questo modo invocabile da chiunque (indipendenza da piattaforme) e in ogni luogo (indipendenza da protocollo comunicativo).

I servizi devono essere infine componibili: un servizio rappresentante Il *workflow* di un *business process* può infatti includere ricerca e invocazione di numerosi altri servizi.

2.2 Aspetti positivi delle SOA

Il paradigma SOA permette quindi riuso, interoperabilità e una crescita a costi contenuti di un sistema distribuito. E' possibile utilizzare caipability interne o private quando necessario e rivolgersi all'esterno quando questo risulta essere più conveniente. Sempre riferendosi all'analogia con il sistema economico, questo è lo stesso principio dell'*outsourcing* : le aziende (service consumer) possono impiegare imprese esterne specializzate (service provider) per realizzare parti del proprio workflow.

In [PapHeu07] sono evidenziati otto principali benefici dovuti al paradigma SOA:

1. **integrazione**: la standardizzazione di componenti e tecnologie permette una rapida integrazione con risorse interne. Un sistema che funziona internamente con il paradigma SOA è facilmente integrabile con i servizi esterni;

2. **riuso**: un'architettura orientata ai servizi permette di sfruttare le stesse soluzioni per problemi diversi;

3. **composizione**: ogni servizio è una composizione di altri servizi più piccoli, permettendo quindi una maggiore modularità;

4. **sfruttamento d'investimenti già esistenti**: tramite SOA è possibile l'utilizzo di sistemi legacy da parte di altri servizi. Sistemi che prima erano isolati possono contribuire alla fornitura di nuovi servizi;

5. **standard XML**: ogni dato nelle SOA è rappresentato mediante XML, uno standard affermato per l'interoperabilità fra piattaforme che permette di minimizzare i costi di conversione delle informazioni;

6. **diminuzione dei costi di comunicazione**: la standardizzazione di un architettura orientata ai servizi permette ad un organizzazione di investire in un unico sistema di comunicazione, interno ed esterno;

7. **possibilità di scelta**: tramite la descrizione standardizzata dei servizi offerti dai vari service provider, le SOA permettono ai service consumer di scegliere sempre il migliore fornitore disponibile;

8. **bassi switching cost**: le SOA permettono alle imprese di evolvere senza restare legate a particolari tecnologie. E' possibile cambiare fornitore di servizi agilmente, senza dover sostenere elevati costi di transizione.

Come abbiamo visto, le SOA permetto quindi di creare a bassi costi sistemi distribuiti agili e flessibili, modellati su misura per le funzionalità richieste e ampliamente riusabili.

Dopo questa breve introduzione teorica, vediamo quali sono le tecnologie che allo stato dell'arte permettono alle architetture orientate ai servizi di funzionare.

2.3 WS

Secondo [PapHeu07], il paradigma service oriented nasce grazie alla crescita della tecnologia dei *Web Service* (WS).

Un WS è definito dal *World Wide Web Consortium* (W3C) come un sistema software in grado di offrire funzionalità ad altri elaboratori tramite interfacce di rete [W3C03]. In base a quanto detto in precedenza, un WS rappresenta quindi l'implementazione di un servizio coinvolto in una SOA.

I WS nascono per integrare applicazioni risiedenti in piattaforme differenti principalmente tramite i protocolli HTTP, XML e SOAP.

Il protocollo *Simple Object Access Protocol* (SOAP) è lo standard di comunicazione dei WS e permette la formattazione di messaggi *XML-based*. Questi messaggi sono inviati ad altri WS tramite HTTP/S o SMTP. Il ricevente estrae le informazioni dal messaggio e le trasforma in un protocollo conosciuto dal sistema risiedente. Le informazioni sono elaborate e tipicamente viene spedito al mittente un messaggio SOAP di risposta.

La gestione delle richieste d'invocazione di un WS, la creazione e l'instradamento dei messaggi SOAP sono affidati ad un *WS container* che può essere integrato in un comune *application server*.

Nei WS la caratteristica di trovabilità del servizio è garantita dal linguaggio *Web Service Description Language* (WSDL) e dal protocollo *Universal Description, Discovery and Integration* (UDDI).

WSDL [W3C04] permette di descrivere con una sintassi *XML-based* i requisiti funzionali che un particolare servizio offre, rappresenta quindi l'interfaccia pubblica con la quale altri servizi possono comunicare.

Per ogni messaggio scambiato tra WS deve essere definito il tipo, e per definire i tipi WSDL usa l'elemento *types*. Gli elementi *message* definiscono i

messaggi, la base della costruzione di un WS con WSDL. I messaggi sono gli elementi che costituiscono input e output dei servizi, e possono contenere i tipi di dati complessi definiti nella sezione *Type* oppure semplici dati primitivi.

Gli elementi *operation* definiscono le operazioni, le funzionalità che saranno esposte dall'interfaccia del servizio. Gli elementi operation contengono elementi input, output e fault specificanti i messaggi scambiati durante l'operazione.

I collegamenti, definiti degli elementi binding, eseguono la mappatura tra il servizio ed il protocollo di comunicazione SOAP [W3C03a], e sono essenzialmente istanze degli elementi *portType*. L'ultimo elemento di un file WSDL è la definizione del servizio, l'elemento service, che consente di raccogliere tutte le operazioni sotto un unico nome.

UDDI è un sistema per standardizzare i registry dei WS, una sorta di elenco dei WS disponibili. L'interazione fra service consumer e service provider nel contesto dei WS può essere molto complessa a causa delle numerose componenti (pubblicazione/consumo WSDL, invio messaggi, trasporto) che entrano in gioco. In contesti reali inoltre i WS coinvolti possono essere numerosi e per ridurre la complessità è stato introdotto un componente dell'architettura che lavora ad un livello d'astrazione più alto, un *service aggregator*.

Un service aggregator permette di comporre servizi per crearne di nuovi mascherando la complessità interna di un sistema. Sono nati a questo scopo aggregator lenguage, come WS-BPEL e JOLIE, descritti nella prossima sezione.

2.4 WS-BPEL

L'integrazione di sistemi richiede qualcosa di più che utilizzare protocolli standard per le comunicazioni come SOAP. Il vero potenziale dei WS come piattaforma d'integrazione entra in gioco solo quando è disponibile un mo-

dello standardizzato che regola le interazioni tra i servizi offerti e *business process* interni.

Il modello offerto dai WS, o meglio dal WSDL, è un modello *stateless* mentre i modelli business sono tipicamente *statefull*, possono comprendere interazioni di lunga durata e con molti partecipanti. Per fondere l'idea di servizio e business process occorre quindi una descrizione formale del protocollo di comunicazione utilizzato per interagire fra le parti.

Tre aspetti fondamentali di cui questo protocollo di comunicazione deve tenere conto sono:

- le operazioni dipendono dai dati. Richeste con input più grandi possono richiedere tempistiche più lunghe. Sono necessarie quindi primitive che permettano limiti temporali di esecuzione;
- la gestione delle eccezioni e dei comportamenti anomali è tanto importante quanto l'esecuzione normale;
- lunghe interazioni possono coinvolgere numerose parti con processi annidati. Devono quindi esserci sistemi di coordinamento dei processi interni.

WS-BPEL [W3C03a] è l'esempio di un linguaggio in grado di "orchestrare" business process collegati fra loro mediante WS.

Fornendo una definizione formale di business process, chiamato *workflow*, con WS-BPEL è possibile esporre qualsiasi operazione sotto forma di servizio e programmare le interazioni fra le diverse parti.

WS-BPEL è dotato di una grammatica XML-based e opera secondo un paradigma definito come *orchestration* che si contrappone alla *choreography*.

L'orchestration si riferisce ad un business process eseguibile che può interagire con servizi interni ed esterni scambiando messaggi. E' evidente quindi che nell'orchestration esiste un processo (inteso come servizio) principale che comuinica con altri servizi per adempiere alla propria logica di business. Dal proprio punto di vista, questo processo è un orchestratore di altri servizi che sono quindi subordinati.

Nella choreography invece tutte le parti coinvolte in una SOA hanno la stessa importanza, ognuna con differenti funzioni nelle interazioni. Il workflow per realizzare una particolare operazione non è quindi custodito da un unico processo come nell'orchestration ma piuttosto condiviso fra tutti i processi che formano l'architettura. La choreography è quindi per sua natura una forma di composizione di servizi più collaborativa che l'orchestration. Un esempio di choreography language è WS-CDL [W3C05], mentre tra gli orchestrator language oltre a WS-BPEL possiamo indicare JOLIE.

2.5 JOLIE

JOLIE [MGLZ06] è un orchestrator language basato su SOCK [MGLZ06], un *process calculus* ideato appositamente per il contesto delle SOA considerando problematiche legate al *session management* e primitive di comunicazione di rete.

Il punto di forza di JOLIE rispetto a WS-BPEL, oltre al sistema formale sottostante, è la sintassi Java/C-like, senza dubbio più comprensibile rispetto a quella XML-based di WS-BPEL.

Il linguaggio JOLIE e il sistema formale sottostante SOCK formano un framework in grado di modellare e implementare architetture orientate ai servizi, rappresentando quindi un esempio dell'evoluzione di questo emergente paradigma di programmazione.

Questa tesi ha portato all'implementazione di Joliepse, un IDE per il linguaggio JOLIE, la sintassi e la grammatica saranno anilizzate e discusse nel capitolo quattro.

Cerchiamo ora di vedere come in JOLIE vengano interpretate le SOA e i concetti precedentemente introdotti in questo capitolo.

In JOLIE un servizio assume un significato meno astratto rispetto al reference model OASIS, e per comprenderlo occorre definire tre importanti concetti: Behaviour, Engine e Description.

2.5.1 *Behaviour*

Il behaviour di un servizio è definito come la descrizione delle attività composte in un workflow. Le attività rappresentano gli aspetti funzionali, la logica di business di un servizio, mentre la loro composizione in un workflow rappresenta la sequenza temporale di esecuzione di queste attività. Il workflow ha quindi lo stesso significato di WS-BPEL.

Le attività si dividono in funzionali, che riguardano la logica di business interna, attività per la gestione dei guasti (molto importante nelle SOA) e attività di comunicazione. Queste ultime sono ovviamente le più importanti e in JOLIE prendono il nome di *operation*, come in WSDL. Esistono due operazioni di input, e cioè *One-Way*, per ricevere un unico messaggio, e *Request-Response*, per ricevere un messaggio e inviare una risposta. Le operazioni di output, sono dualmente chiamate *Notification*, per inviare un solo messaggio e *Solicit-Response* per inviare un messaggio e ricevere una risposta.

2.5.2 *Engine*

In JOLIE un engine è un sistema in grado di gestire sessioni di servizi sfruttando creazione di sessione, supporto per gli stati, instradamento dei messaggi ed esecuzione di sessioni. Per comprendere meglio questa definizione occorre definire le quattro caratteristiche sopra citate.

La creazione di una sessione è la fase di inizializzazione di un servizio e può essere deifinita in maniera esplicita da parte dell'utente (*firing session*) o implicitamente durante la ricezione di un messaggio in un'operation opportunamente marcata come *session initiator*.

Come supporto per gli stati si intende la capacità di una sessione di accedere a dati non locali. A tale scopo esistono tre stati: uno stato locale che comprende i dati non visibili all'esterno della sessione, uno stato globale che permette di condividere dati fra le sessioni attive nell'engine e uno stato di memorizzazione (file o database) che garantisce la persistenza dei dati.

In JOLIE ogni sessione è identificata tramite *correlation set*, un concetto già presente in WS-BPEL che permette all'engine di instradare i messaggi verso la sessione corretta. E' possibile definire il correlation set per ogni sessione e influenzare quindi l'instradamento dei messaggi.

L'esecuzione invece riguarda il lancio di una sessione creata con il supporto degli stati corrispondente. L'esecuzione di sessioni può essere sequenziale o concorrente.

2.5.3 *Service Description*

La descrizione di un servizio riguarda l'interfaccia esterna, il modo in cui le funzionalità vengono esposte. In JOLIE la descrizione di un servizio si compone di due parti, interfaccia e *deployment*.

Il concetto d'interfaccia può essere identificato in tre differenti livelli. Un livello funzionale, che permette di definire quali siano le operazioni di un servizio in grado di ricevere e inviare messaggi (simile a un WSDL di un WS); un livello di workflow, che descrive il workflow del behviour, e un livello semantico contenente informazioni sulla semantica delle operazioni.

Le interfacce sono collegate al concetto di trovabilità dei servizi, altro aspetto cruciale delle SOA. Essendo JOLIE un linguaggio nuovo e tuttora in fase di sviluppo, la standardizzazione di un registro o *repositroy* di *JOLIE-service* non è ancora stata affrontata. E' anche possibile che in futuro saranno aggiunte alle interfacce funzionali informazioni maggiori relative all'engine rispetto a quelle attuali.

L'altro componente della descrizione di un servizio è rappresentato dal deployment, l'implementazione che collega un interfaccia con un protocollo di rete. In JOLIE il deployment di un'interfaccia avviene tramite la dichiarazione di porte di input o output. Una porta è, formalmente, un *endpoint* dotato di un indirizzo di rete e un protocollo di comunicazione collegato a un'interfaccia le cui operazioni saranno utilizzate per inviare e ricevere messaggi.

2.5.4 Composizione di servizi in JOLIE

Come sappiamo la composizione di servizi per crearne nuovi è una caratteristica fondamentale del paradigma SOA. In JOLIE i servizi possono essere composti tramite composizione semplice, *embedding*, *redirecting* e *aggregation*. Queste tecniche possono essere utilizzate insieme, impostando almeno un servizio contenente una firing session.

La **composizione semplice** avviene tramite l'esecuzione parallela di service container che comunicano in rete.

Nell'embedding più servizi sono inseriti nello stesso container e sono quindi in grado di comunicare fra loro senza l'utilizzo di una rete. L'embedding è l'ideale per servizi interni di manipolazione di dati che non richiedono visibilità esterna.

Nella composizione tramite **redirecting** un solo servizio, chiamato master, viene utilizzato per ricevere tutti i messaggi e per instradarli a servizi sottostanti, chiamate risorse. Il servizio master sarà quindi dotato di un unica input port e di tante output port quante le risorse utilizzate. Il client di una risorsa specifica al master il nome del servizio corrispondente senza doversi preoccupare di quali protocolli di rete saranno utilizzati dal master per raggiungere la risorsa.

Tramite **l'aggregation** più servizi possono essere composti mascherando completamente la struttura sottostante. Se nel redirecting un client conosce tutte le risorse subordinate al servizio master, in questo caso è visibile un unico servizio principale. Per realizzare le diverse funzionalità offerte ai client il master utilizza altri servizi, ma questi sono completamente invisibili al client.

2.5.5 *Design pattern* in JOLIE

I concetti di servizio, redirecting, aggregation e embedding permettono di individuare una serie di pattern architetturali che possono essere utilizzati in qualsiasi SOA.

Un primo pattern è rappresentato dalla classica architettura client-server. Con JOLIE è possibile realizzare un servizio HTTP che offre operation di tipo Request-Response come GET e POST. E' possibile in questo modo modellare tutte le funzionalità esistenti in un web server classico.

Altri pattern utilizzati sono lo *slave service mobility* e il *master service mobility*, che riguardano la mobilità di servizi. E' infatti possibile effettuare l'embedding di servizi dinamicamente, a tempo di esecuzione. In questo modo nello slave service mobility un servizio master può effettuare l'embedding di un servizio slave a tempo di esecuzione (non conoscendo staticamente le funzionalità che saranno richieste), mentre nel master service mobility un servizio può effettuare a runtime l'embedding di un servizio master per conoscere ed eseguire il workflow contenente le proprie operazioni.

Il *service of service pattern*, infine, permette di effettuare l'embedding dinamico di un servizio da parte di un master quando un client lo richiede. Il master scarica il servizio da un repository, effettua l'embedding e comunica il nome della risorsa al client. Il client per utilizzare il nuovo servizio utilizzerà il nome della risorsa sfruttando quindi il meccanismo di redirecting.

Riassunto capitolo

In questo capitolo è introdotto lo sfondo tecnologico che ha portato alla creazione del linguaggio JOLIE: l'emergente paradigma SOA e l'esigenza di un linguaggio maggiormente comprensibile e versatile rispetto a WS-BPEL.

Nel prossimo capitolo è presentato lo stato dell'arte nell'implementazione di IDE, che ha portato alla scelta del framework Xtext per l'implementazione di Joliepse, un IDE per JOLIE.

Capitolo 3

IDE: stato dell'arte

Durante la fase iniziale di questa tesi sono state analizzate le scelte disponibili allo stato dell'arte per arrivare all'implementazione di un IDE per il linguaggio JOLIE.

Lo sviluppo dell'ingegneria del software negli ultimi anni ha fatto crescere l'esigenza di IDE che permettessero agli sviluppatori di poter lavorare a progetti di varia natura su un'unica piattaforma. E' utile, infatti, poter utilizzare lo stesso strumento per creare componenti logiche, interfacce utente, gestire database e ogni aspetto richiesto dalle complesse architetture software moderne.

Creare un IDE per un nuovo linguaggio di programmazione fino a poco tempo fa rappresentava un lavoro abbastanza lungo e complesso. I moderni IDE, come MS Visual Studio [MVS10] nel mondo commerciale ed Eclipse [Ecl10] e NetBeans [Net10] nella sfera opensource facilitano questo compito fornendo piattaforme predisposte all'integrazione di qualsiasi tipo di componente.

Vediamo come prima cosa quali sono le principali funzioni che un IDE deve fornire per permettere a uno sviluppatore di aumentare la propria produttività. Queste funzioni rappresentano un'evoluzione di quelle già mostrate in [TeiRep81].

3.1 Principali funzionalità offerte da un IDE

Project Building

Normalmente un progetto software si compone di un'insieme di sorgenti e un insieme di artefatti. I sorgenti possono comprendere il codice sorgente del software, file di configurazione, librerie e molto altro. Gli artefatti tipicamente comprendono gli elementi tradotti nel linguaggio macchina del sistema su cui il software verrà eseguito. Un classico esempio di questa divisione è rappresentato dal codice sorgente Java e i file *class* prodotti che sono utilizzati dalla VM per l'esecuzione del programma.

Per quanto riguarda l'implementazione di questa funzionalità, dobbiamo considerare che un ambiente tipico è rappresentato da progetti con un grande numero di file. Il componente che si occupa della gestione del *building* tiene solitamente traccia delle modifiche effettuate ai sorgenti dall'ultimo *build*. Sono inoltre utilizzate le informazioni riguardanti le dipendenze tra file (per esempio interfacce e implementazioni in Java) per definire quali di questi vanno ricompilati.

Esecuzione progetto e *debugging*

Un altro servizio offerto tipicamente dagli IDE è la capacità di eseguire il codice prodotto per il progetto sia in modalità normale che in modalità *debug*. La modalità debug fornisce la possibilità di eseguire un programma passo-passo e permette quindi una facile individuazione degli errori da parte del programmatore, che è in grado ad esempio di visualizzare i valori delle variabili in ogni momento del flusso di esecuzione.

La fase di debugging può essere scomposta in parti tramite l'utilizzo di breakpoint e condizioni che permettono di rendere la fase d'individuazione degli errori meno frustrante. I *breakpoint* vengono ad esempio utilizzati per indicare in quale punto del codice avviene un errore durante la fase di compilazione.

Highlight del codice

La *feature* base e fondamentale di ogni IDE è ovviamente l'editing del codice. Uno dei primi strumenti comparsi nel tempo come supporto a questa fase è l'evidenziazione, o comunemente *highlight*, delle parole chiave e delle strutture sintattiche del codice sorgente. Possono essere utilizzati colori o *font* differenti per evidenziare commenti, stringhe, valori numerici, definizioni di metodi e quanto più possa essere utile a una rapida e semplice comprensione del codice.

L'evidenziazione del codice dipende fortemente dalla velocità con cui l'IDE risponde alle modifiche dei file. I costrutti sono evidenziati dopo un'analisi lessicale, sintattica e semantica e nel caso di un file di grande dimensioni questo processo può essere molto laborioso. Una soluzione adottata è quella di limitare il processo di evidenziazione soltanto all'area del testo appena modificata.

Indentazione e formattazione

Utilizzare uno stile di scrittura del codice uniforme permette di aumentare la comprensibilità del codice e quindi il mantenimento del software nel tempo. Uno "stile" di formattazione fornisce linee guida relative al modo di definire i nomi degli elementi, la lunghezza media delle linee di codice e altre caratteristiche stilistiche. In Java per esempio in nomi dei metodi dovrebbero iniziare con una lettera minuscola. A questo scopo sono state definite regole di stile per diversi linguaggi di programmazione [Ora10].

I modrerni IDE permettono di mantenere uno stile uniforme proponendo indentazione e formattazione automatica del codice e suggerimenti e proposte per i nomi di metodi, variabili e tipi.

L'indentazione può avvenire automaticamente durante la fase di input del codice o successivamente, su richiesta da parte dell'utente selezionando solo una parte del testo. Per non rendere quest'operazione troppo lunga sono utilizzati *parser* poco complessi che permettono una rapida analisi

dellla struttura, solitamente diversi dai parser utilizzati prima della fase di interpretazione o compilazione.

Rappresentazione visuale del codice

Per aiutare i programmatori a navigare in maniera semplice all'interno del codice, specialmente in programmi di grande dimensione, gli IDE utilizzando sistemi di visualizzazione della struttura del codice, tipicamente indicati con i termini di *outline* e *folding*. La tipica forma di questa rappresentazione è un albero gerarchico in cui gli elementi del codice rappresentano le foglie.

Un esempio è la visualizzazione classica di un file Java: al livello più alto dell'albero troviamo la definizione del package, le dichiarazioni di import e le classi definite nel file. E' possibile espandere le classi per visualizzare attributi e metodi definiti. In questo modo il programmatore visualizza la struttura del codice graficamente e può muoversi in maniera semplice fra le varie parti del programma.

Un'altra utile funzionalità è il folding, con il quale si intende la rappresentazione di un intero blocco in un unica riga di codice per mascherare ad esempio la definizione di un metodo in un sorgenta Java.

Per implementare queste funzioni di visualizzazione grafica l'IDE utilizza un modello, comunemente un AST associato ad ogni file. Da questo modello sono estrapolate le informazioni riguardanti la struttura del file e il modulo relativo ricompone queste informazioni graficamente sotto forma di albero.

Ricerca di elementi nel codice

Un sistema software tipicamente include numerosi file che possono facilmente raggiungere le migliaia nei progetti più complessi. E' indispensabile quindi un sistema di ricerca che possa permettere sia di recuperare dove ad esempio si trovi la definizione di un metodo e, ugualmente importante, in quali punti del progetto questo metodo sia richiamato. I moderni IDE spesso includono l'utilizzo di pattern ed espressioni regolari per raffinare le ricerche.

Le funzioni di ricerca devono essere rapide. La ricerca è da sempre uno dei grandi argomenti studiati dall'informatica e gli IDE utilizzano i progressi ottenuti a livello algoritmico in quest'ambito per rendere efficiente questo processo. Per questo motivo si combinano meccanismi di *lazy loading*, *caching* e indicizzazioni.

Autocompletamento

L'evoluzione dell'editing dei moderni IDE ha portato a una delle funzionalità più utili e in grado di aumentare la produttività degli sviluppatori, l'autocompletamento sensibile al contesto. L'IDE può, infatti, consigliare e proporre parole chiave e referenze che possono essere utilizzate in un particolare punto del programma.

Anche l'autocompletamento utilizza funzioni di ricerca. Per fornire questa *feature*, legata al contesto, occorre tenere traccia delle informazioni sullo scope attuale e questo è realizzato tramite una ricerca all'interno dei modelli rappresentati i file. Quando l'utente richiede la funzione di autocompletamento, il modulo corrispondente dell'IDE determina il contesto e fornisce una lista di scope. Questa lista è raffinata eliminando le proposte semanticamente scorrette e offrendo i risultati in ordine alfabetico o per frequenza di utilizzo.

Wizard

Con il termine *wizard* si intende la capacità offerta da un IDE di creare un template per un progetto o per certi elementi di un progetto. Per esempio è possibile definire graficamente la struttura di una classe Java (interfacce implementate, campi, *getter* e *setter*) e far generare all'IDE il codice sorgente relativo.

Per implementare il wizard sono utilizzate tecniche di template engine o executable pattern come Apache Velocity [Vel10], FreeMarker [Fre10] o Xpand, utilizzato da Xtext e descritto nel prossimo capitolo. Un template

engine è uno strumento in grado di generare codice a partire da un file di configurazione.

Anche questo modulo utilizza la funzionalità di ricerca, per esempio volendo generare una classe Java che implementa una certa interfaccia, occorre ricercare le signature dei metodi che saranno implementati.

Refactoring

Un importante strumento che sta assumendo sempre più importanza è il *refactoring*, cioè la possibilità di modificare la struttura di un progetto mantenendo intatta la sua correttezza semantica [FMR99]. Utilizzando il refactoring è possibile ad esempio modificare il nome di un metodo e lasciare che l'IDE automaticamente modifichi il nome in tutte le occorrenze del metodo stesso all'interno del progetto.

Il refactoring è implementato combinando tecniche di ricerca con modifiche ai modelli dei file contenenti le referenze, che vengono poi riconvertiti nei file sorgente corrispondenti.

Queste sono le funzionalità che la maggior parte degli IDE forniscono per i linguaggio di programmazione più diffusi come Java e C. Vediamo nella prossima sezione quali sono gli IDE più diffusi e quali sono loro caratteristiche.

3.2 IDE come piattaforma d'integrazione

Negli ultimi anni si può osservare una tendenza all'utilizzo degli IDE open surce come piattaforme d'integrazione da parte di sviluppatori di terze parti. I moderni IDE come Eclipse e NetBeans hanno una struttura aperta basata sull'altissima estendibilità tramite lo sviluppo di plug-in ed estensioni. Gli IDE forniscono l'accesso alle funzionalità base descritte sopra a questi plug-in e questa architettura ha aperto alla possibilità di utilizzare gli IDE come base per implementare piattaforme per qualsiasi linguaggi di programmazione oltre che a strumenti di integrazione in ambienti enterprise [YanJian07].

Partendo dal presupposto che creare un IDE per un nuovo linguaggio di programmazione da zero rappresenta un lavoro molto dispendioso e contro i moderni principi dell'ingegneria del software, IDE già esistenti visti come piattaforme universali d'integrazione rappresentano un ottimo punto di partenza. Vediamo quindi una breve rassegna degli IDE esistenti più diffusi.

3.2.1 Eclipse, NetBeans e MS Visual Studio

Tra i software commerciali il mercato è dominato da Microsoft Visual Studio [MVS10] per la piattaforma .NET, mentre nel mondo open source le scelte degli sviluppatori si dividono principalmente fra Eclipse [Ecl10] e NetBeans [Net10].

Entrambi gli IDE sono nati come alternative a Visual Studio e in concomitanza con la diffusione esponenziale del linguaggio Java e la piattaforma Java EE, la soluzione open source per ambienti enterprise che si contrappone a .NET.

NetBeans rappresenta da questo punto di vista la scelta effettuata da Sun, l'azienda sviluppatrice di Java. L'IDE è stato acquistato alla fine degli anni novanta da un progetto di uno studente ceco [Vau03] e il codice sorgente è stato rilasciato al pubblico, permettendo agli sviluppatori di modellare il software secondo le proprie esigenze, creando ad esempio estensioni per altri comuni linguaggi come C e Cobol.

Eclipse nasce per iniziativa di IBM, Borland, RedHat e altre società riunite nell'Eclipse Foundation e s'impone da subito come concorrente di NetBeans nell'ambiente Java. Entrambi gli IDE forniscono un *core* formato da un editor, compilatore, debugger e una GUI *user frendly*.

NetBeans

NetBeans utilizza due componenti principali: la piattaforma, che comprende una serie di librerie per fornire gli elementi base dell'IDE come presentazione dei dati e interfaccia utente, e l'IDE vero e proprio, che permette di gestire il controllo e le funzionalità offerte dalla piattaforma.

NetBeans utilizza *Abstract Window Toolkit* (AWT), un insieme di API realizzate da Sun che permettono agli sviluppatori di modellare le interfacce grafiche delle finestre, pulsanti e altri elementi visuali. AWT fornisce gli elementi grafici base che dipendono dalla piattaforma utilizzata, mentre per gli aspetti di alto livello come gestione di colori e interazione con l'utente è usata la libreria Swing.

Eclipse

Prima della nascita dell'Eclipse Foundation IBM utilizzava internamente l'IDE con il linguaggio SmallTalk. A differenza di NetBeans Eclipse nasce come IDE generico, non legato soltanto al linguaggio Java. Come nell'IDE di Sun, una serie di API collega fra loro l'editor, il debugger e tutti i moduli e i plug-ins che lavorano in sincronia offrendo un'unica interfaccia utente. Per gli elementi grafici Eclipse utilizza *Standard Widget Toolkit* (SWT) che permette di utilizzare la stessa interfaccia grafica del sistema operativo che si sta utilizzando.

Una delle grosse differenze tra i due IDE opensource è proprio nel motore grafico utilizzato. Sfruttando le API grafiche del sistema operativo ospitante, Eclipse (tramite SWT) permette una performance maggiore rispetto all'utilizzo di AWT e Swing da parte di NetBeans. Questo trade-off permette comunque una maggiore libertà nelle possibilità di estensione a livello grafico dell'IDE di Sun.

Dalla sua nascita l'Eclipse Foundation ha superato le 100 società, e tra i partecipanti troviamo realtà come Cisco, Google, IBM, Intel, Oracle e Sap.

Eclipse è stato creato tenendo conto degli standard proposti dall'*Object Management Group* (OMG) [OMG10], che produce e mantiene specifiche per l'interoperabilità fra applicazioni enterprise. Il successo di Eclipse deriva proprio da questo. L'IDE è infatti stato predisposto fin dall'inizio all'estendibilità. Ogni componente è un plug-in e qualsiasi sviluppatore può scrivere un nuovo plug-in in grado di comunicare con tutte le altre componenti.

Questa caratteristica ha permesso una diffusione capillare dell'IDE. Esistono infatti plug-in per praticamente ogni settore dell'ingegneria del software (1600 contro i 500 per NetBeans) [Gee05], e queste estensioni si integrano perfettamente in un IDE che, oltretutto, è gratuito e opensource.

I plug-in sviluppati e le versioni *Rich Client Platform* (RCP) possono essere rilasciati sotto licenza, e l'esempio "commerciale" di Eclipse è rappresentato da MyEclipse, disponibile comunque a un costo molto più basso rispetto alla media del settore. La natura open source del progetto permette una rapida innovazione e sviluppo che indirettamente conviene a tutte le società facenti parte dell'Eclipse Foundation.

MS Visual Studio

La serie di software che nel nome commerciale utilizzano il prefisso Visual si riferisce a prodotti di sviluppo per Microsoft. Alcuni esempio sono Visual C++, Visual Basic e Visual J++, linguaggi creati appositamente con lo scopo di rendere semplice e rapido lo sviluppo di applicazioni per il proprio sistema operativo Windows.

La prima versione di uno strumento integrato comprendente tutti questi linguaggi compare nel 1997 con il nome di Visual Studio. Nel 2003 dopo una fase di ristrutturazione Visual Studio apre agli sviluppatori di estensioni di terze parti, creando il *Visual Studio Industry Partners* (VSIP) e rilasciando VS SDK.

3.3 Esistenti approcci nell'implementazione di IDE

Ripetiamo ancora una volta che l'utilizzo di piattaforme d'integrazione come Eclipse, NetBeans e Visual Studio rappresenta un punto di partenza obbligato per gli sviluppatori di IDE per nuovi linguaggi, permettendo un grande risparmio di tempo e risorse.

3.3.1 Il ruolo del compilatore

I moderni compilatori hanno una struttura modulare organizzata per componenti. Ognuno di questi componenti si occupa di una fase specifica del processo di compilazione, che solitamente sono:

- analisi lessicale
- analisi sintattica
- analisi semantica
- generazione codice intermedio
- ottimizzazione codice intermedio
- generazione codice oggetto

Le prime tre fasi sono indipendenti dalla piattaforma utilizzata per compilare il programma ma dipendenti dal linguaggio di programmazione che si sta considerando. Le ultime tre, dualmente, sono indipendenti dal linguaggio considerato ma dipendenti dalla piattaforma target.

Come abbiamo visto nella prima parte di questo capitolo le principali funzioni fornite dai moderni IDE manipolano il codice sorgente o, per meglio dire, il modello che lo rappresenta. E' naturale quindi che l'implementazione di un IDE per un determinato linguaggio si appoggi a un compilatore (o un interprete) esistente per ottenere queste funzionalità.

Ad esempio per fornire suggerimenti di autocompletamento al programmatore l'IDE chiede al parser del compilatore di creare l'AST corrispondente al programma sorgente, visita l'albero ed è in grado di determinare il contesto e quindi le eventuali proposte.

Questa operazione comunque non è immediata, esistono infatti vincoli che non permettono di implementare le funzionalità dell'IDE semplicemente sfruttando il compilatore. Un semplice esempio è rappresentato dalla segnalazione degli errori: nel normale processo di compilazione questa causa lo

stop immediato dell'operazione mentre nel caso di un IDE occorre comunque produrre un AST completo anche in caso di errori multipli.

3.3.2 *Implementation Cloning*

Un primo approccio nella creazione di IDE per nuovi linguaggi è rappresentato dall'*implementation cloning*, descritto in [Gom08].

Nel corso della sua evoluzione Eclipse, pur mantenendo centrale Java, ha iniziato un processo di apertura a nuovi linguaggi. La creazione di *Java Development Tool* (JDT) [Ecl06] rappresenta il punto d'inizio di questo processo, trasformando Eclipse in una piattaforma di sviluppo in grado di fornire l'accesso alle funzionalità fornite dall'IDE e di fornire quindi grande libertà agli sviluppatori. Il primo approccio nello sviluppo di IDE per nuovi linguaggi è stato quindi quello di "clonare" le funzionalità offerte da Eclipse JDT, processo comunque non semplice a causa della bassa riusabilità del codice di JDT.

Gli sviluppatori di *C/C++ Development Tool* (CDT) utilizzano a volte l'acronimo come *Cloned Development Tool* per riferirsi all'approccio utilizzato per implementare l'IDE.

L'implementation cloning è utilizzato anche da MS Visual Studio che nell'SDK fornisce esempi di codice che possono essere liberamente copiati in nuovi progetti e modificati a piacere.

Il grande vantaggio di questo meccanismo d'implementazione è la rapidità con cui è possibile produrre un prototipo, anche se perfezionare questo prototipo è comunque molto laborioso.

3.3.3 DLTK

La clonazione di JDT per l'implementazione di un nuovo linguaggio è, come abbiamo visto, un buon punto di partenza, ma comunque richiede un grande lavoro di raffinamento successivo, dovendo sostituire il supporto per Java con quello per il nuovo linguaggio. L'esigenza di una piattaforma

di partenza non dipendente da un linguaggio ha portato alla creazione di DLTK [Ecl06], nato per fornire supporto a linguaggi interpretati come Perl, Ruby e PHP.

Grazie all'esperienza ottenuta durante la creazione di JDT gli sviluppatori hanno aggiunto interfacce predisposte al riuso e quindi perfette per l'adattamento ad altri linguaggi, fornendo quindi un modello generico per la creazione di IDE che riprende quindi i concetti di [BahSne86] e [BCD+88] visti nell'introduzione.

Il vantaggio rispetto al cloning è quindi la possibilità di poter estendere direttamente le funzionalità offerte dall'IDE, permettendo ad esempio di usare un compilatore preesistente per la fase di analisi sintattica e lessicale e di sfruttare il motore di ricerca interno. Esiste comunque un considerevole aspetto negativo dell'utilizzo del DLTK: il modello generico è stato creato per i linguaggi dinamici simili fra loro, con costrutti simili e quindi con AST strutturalmente analoghi. Un esempi di IDE implementato con DLTK è rappresentato da [YRR+09].

3.3.4 Babel Package in MS Visual Studio

La proposta di Microsoft per un framework generalizzato è rappresentata da Babel, un package per Visual Studio SDK contenente le funzionalità base di un IDE come evidenziazione, outline, folding, autocompletamento e formattazione. Queste informazioni sono prese dal compilatore, che deve essere opportunamente modificato per interagire con Babel. Il costo di quest'operazione dipende ovviamente dalla struttura del compilatore.

Babel permette quindi di fornire le funzioni sopra citate senza intervento da parte degli sviluppatori, che devono invece occuparsi dell'interazione con il compilatore. Con Babel non è inoltre possibile implementare funzioni come ricerca, refactoring e wizard.

Un esempio di IDE creato con Visual Studio e Babel è Visual Haskell [AngMar05].

3.3.5 Proiezione del *code model*

Per ottenere i maggiori vantaggi da entrambi gli approcci visti in precedenza occorre considerati tre importanti punti:

- deve essere possibile utilizzare il compilatore o l'interprete già esistente;
- deve essere possibile utilizzare diversi modelli (o AST) all'interno dell'IDE;
- devono essere fornite le funzioni base di un IDE altamente estendibili.

L'utilizzo di DLTK è utile quando il modello utilizzato dal compilatore o dall'interprete è simile al modello generico del framework. Poichè i modelli di diversi linguaggi possono essere molto diversi, questo rappresenta una grande limitazione di DLTK che, come già detto, è indicato per linguaggi di scripting.

Per evitare questi problemi è da notare l'approccio proposto da Babel: per ogni funzionalità offerta dall'IDE sono definiti i dati non dipendenti dal linguaggio necessari al funzionamento, che rappresentano l'interfaccia con il modello generico. Il processo di estrazione di questi dati da un compilatore preesistente è chiamato proiezione del modello del linguaggio [Gom08]. Utilizzando quest'approccio è possibile eliminare la limitazione a famiglie simili di linguaggi.

Un pratico sistema per rappresentare questi dati è XML, che permette di offrire flessibilità, estendibilità e la possibilità di usare database XML per semplificare la ricerca tramite strumenti come XQuery. La natura language-independet di XML permette di utilizzare lo schema dei dati richiesti dall'IDE per implementare le funzionalità con qualsiasi linguaggio di programmazione.

3.3.6 *IDE Meta-tooling Platform*

IDE Meta-tooling Platform (IMP) [CFS07] è una piattaforma per la creazione di IDE specifici per nuovi linguaggi. IMP combina un framework indipendente dal linguggio, generatori per le classiche funzionalità dell'IDE e un supporto per l'estensione di questi servizi che garantisce ampia flessibilità,

permettendo di programmare a differenti livelli d'astrazione. Diversamente da DLTK permette quindi un elevato livello di personalizzazione dell'IDE creato e uno sviluppo incrementale delle funzionalità.

La prima operazione da effettuare nello sviluppo di un IDE tramite IMP consiste nella definizione del linguaggio target tramite informazioni come nome del linguaggio, estensioni utilizzate e altri aspetti. La seconda operazione è quella della configurazione del lexer e del parser che l'IDE utilizzerà durante l'esecuzione.

IMP è progettato per funzionare con qualsiasi parser, sia auto generati che scritti ad hoc, e a questo scopo fornisce API per quanto riguarda l'AST che verrà successivamente utilizzato. Quest'approccio ricorda quindi molto da vicino la proiezione del code model vista precedentemente. E' inoltre possibile utilizzare il generatore di parser incluso LPG, uno strumento opensource in caso di generare parser LALR(k) semplicemente definendo una grammatica *Backus-Naur Form* (BNF) per il linguaggio. Un'importante caratteristica fornita da LPG è l'auto generazione di classi visitor per le successive analisi dell'AST. A questo punto è possibile utilizzare l'editor per la grammatica che viene fornita con alcuni costrutti esempio tipici dei linguaggi imperativi. Ad ogni modifica della grammatica viene richiamato il builder dell'IDE che ricostruisce lexer e parser.

Una volta che il servizio di parsing è stato implementato lo sviluppatore può aggiungere i servizi richiesti secondo un qualsiasi ordine, e l'implementazione di ogni servizio è assistita da un wizard. I wizard si concludono con la generazione di classi skeleton che forniscono le caratteristiche base del servizio richiesto, già pronte per le modifiche e le estensioni.

Per quanto riguarda il linking, utile per la navigazione all'interno del codice, se è disponibile un compilatore già esistente in grado di fornire questo servizio l'implementazione risulta immediata, mentre in caso contrario occorre servirsi di una classe rappresentante una tabella dei simboli già predisposta per linking relativo a scope/definizione, entità/definizione e entità/referenza.

Nel framework IMP si possono distinguere due tipi di componenti: meta-

strumenti, utilizzati durante l'implementazione dell'IDE, e componenti di runtime che invece sono utilizzati durante l'esecuzione dell'IDE. I componenti utilizzati a runtime estendono le classi fornite dal *core* di Eclipse, come ad esempio la classe *UniversalEditor* che estende il *TextEditor* di Eclipse.

L'UniversalEditor implementa le funzionalità dell'IDE creato utilizzando quelle già esistenti in Eclipse, incapsulate in modo da permettere allo sviluppatore di concentrarsi soltanto sulle caratteristiche di questi servizi che dipendono dal linguaggio.

3.3.7 Xtext

Diversamente dai progetti precedenti, Xtext [Eff06] è un framework che permette l'implementazione completa di linguaggi, di cui la generazione di un IDE è soltanto un aspetto. E' disegnato per DSL ma può benissimo essere utilizzato per linguaggi *general purpose*.

Un DSL è un linguaggio di programmazione di piccole dimensioni (in termini di costrutti utilizzati) che si concentra su un particolare dominio.

Per l'impelemntazione dell'IDE di supporto al linguaggio JOLIE è stato utilizzato questo framework per tutta una serie di motivi che saranno descritti nel prossimo capitolo.

Xtext è sviluppato da *Itemis* [Ite10], una società specializzata in model-based programming, rilasciato in modalità opensource e in collaborazione con l'Eclipse Foundation. Il framework è utilizzato nel settore della telefonia mobile, automobilistico, sistemi embedded e nell'industria dei videogiochi. Xtext si compone di circa 1.500 Klines di codice, delle quali solo il dieci per cento sono scritte a mano. La documentazione nella versione attuale è di circa 100 pagine, molto dettagliata rispetto a quella degli altri progetti. Il newsgroup inoltre è molto aggiornato e completo.

Xtext fornisce una serie di API e DSL interni che sono utilizzati per descrivere il linguaggio di programmazione che si desidera implementare, permettendo di ottenere un'implementazione completa basata su componenti che si appoggiano semplicemente alla Java VM e non a Eclipse. I componen-

ti prodotti con Xtext come il parser, l'AST, lo scoping framework, il validator e l'interpeter sono quindi indipendenti da Eclipse e possono essere utilizzati anche in modalità satand-alone.

Gli strumenti generati da Xtext sfruttano EMF e quindi permettono l'integrazione con strumenti collegati a EMF come ad esempio *Graphical Modeling Framework* (GMF), un framework per l'editing grafico di modelli molto versatile.

Con Xtext è possibile inoltre ottenere un editor Eclipse-based per il linguaggio che offre tutte le funzionalità classiche di un IDE, estendibili in modo semplice tramite le API fornite o inserendo componenti esterni sfruttando *Google Guice*, un framework per *dependecy injection*. La dependecy injection è una tecnica avanzata della programmazione ad oggetti che permette di disaccoppiare classi dipendenti fra loro, lasciando questo compito ad un container esterno. In termini pratici, Guice permette di evitare l'utilizzo dell'operatore *new* per istanziare le classi, utilizzando invece l'annotation *@inject* per richiedere al container di occuparsi della gestione della dipendenza.

Xtext utilizza Guice internamente per collegare tutti i componenti del linguaggio fra loro e con l'infrastruttura di Eclipse. E' possibilie configurare il container se le API e i DSL utilizzati per la descrizione del linguaggio risultano troppo deboli, garantendo quindi un elevato livello di estendibilità.

Poichè Xtext è il framework utilizzato nell'implementazione di Joliepse (l'IDE per JOLIE creato durante lo sviluppo di questa tesi) il funzionamento del framework è descritto dettagliatamente a partire dal prossimo capitolo.

3.3.8 NetBeans: Progetto Schliemann

Il progetto Schliemann [janPru08] propone un framework generico per l'integrazione e il supporto a un nuovo linguaggio nell'IDE NetBeans. A partire dalla versione 6.0 il framework è integrato in NetBeans. Il nome deriva dall'esploratore del secolo XIX Heinrich Schilemann, famoso per aver ideato un metodo di rapido apprendimento dei linguaggi locali nel corso dei suoi viaggi.

Il progetto Schliemann si compone di un engine per la definizione del linguaggio e un modulo per l'integrazione nell'IDE, che comprende principalmente elementi di visualizzazione e di editing. Il framework fornisce elementi base e la possibilità di modificare l'output prodotto dall'engine per creare nuove funzionalità.

Il progetto è ispirato alle funzionalità che editor come Emacs o JEdit forniscono, che tipicamente comprendono la colorazione delle parole chiave e l'indentazione automatica, ma permette di andare oltre queste funzioni senza comunque creare un supporto completo al linguaggio. Non è infatti possibile implementare moduli per compilazione e l'esecuzione, il che lo rende indicato soltanto per linguaggi di scripting.

Per integrare un nuovo linguaggio in NetBeans tramite l'engine Schliemann occorre descrivere il linguaggio in un *NetBeans Scripting File* (NBS). Il file si compone di tre sezioni principali: lessicale, che comprende la definizione dei token riconosciuti dal linguaggio, sintattica, contenente le produzioni della grammatica del linguaggio, e di visualizzazione.

Le produzioni richiedono una descrizione della grammatica EBNF, permettendo la definizione di grammatiche LL(K). Oltre all'utilizo dell'EBNF all'interno del file NBS è possibile inserire porzioni di codice Java per gestire l'analisi lessicale e sintattica quando non sono sufficienti le espressioni regolari.

La grammatica EBNF utilizzata è simile a quelle per generatori di parser come JavaCC e AntLR, ma non è possibile inserire questo tipo di grammatiche senza opportune modifiche sintattiche.

Tramite la definizione di questo documento di dimensioni ridotte, l'engine del framework è in grado di fornire automaticamente evidenziazione sintassi, folding, bilanciamento parentesi e indentazione.

Il progetto Schliemann sembra molto promettente ma con una carente documentazione e codice sorgente non pubblico non sembra offrire grandi margini di miglioramento nel breve periodo.

Capitolo 4

Il framework Xtext

Dopo una panoramica sugli approcci esistenti nell'implementazione di IDE, in questo capitolo è descritto il funzionamento del framework Xtext, utilizzato per la realizzazione di Joliepse, un IDE per il linguaggio JOLIE. Possiamo indicare in cinque punti principali le motivazioni di questa scelta:

- **eclipse-based**: escluso MS Visual Studio dal principio per la natura non opensource, Eclipse è stato preferito a NetBeans per la sua maggiore diffusione [Gee05], la disponibilità di un maggior numero di plug-in e la community di supporto. Non è da escludere comunque tra gli sviluppi futuri la possibilità di implementare una versione di Joliepse per NetBeans tramite il progetto Schliemann;

- **funzionalità offerte**: Xtext permette di specificare referenze incrociate durante la definizione della grammatica per il linguaggio target, che oltre le funzionalità classiche di un IDE permette di rendere il codice navigabile, collegando ad esempio la chiamata di una funzione alla sua definizione. Questa caratteristica con DLTK e IMP non è implementabile in maniera semplice se non si dispone di un compilatore già esistente. Xtext permette inoltre di implementare facilmente quick fix, validator e proposal provider, descritti in questo capitolo;

- **riuso degli strumenti generati**: come descritto nell'introduzione, il parser attuale di JOLIE è stato scritto mano senza l'ausilio di generatori. Xtext permette di utilizzare esternamente gli strumenti generati per il supporto all'IDE. Una parte di questa tesi è dedicata al tentativo di sostituire il parser attuale di JOLIE con il parser generato da Xtext, utilizzato in Joliepse. Il sesto capitolo è dedicato a questa fase e ai risultati ottenuti;

- **integrazone con EMF e GMF**: Xtext utilizza il framework EMF e questo permette di rendere l'IDE Joliepse e gli sturmenti generati da Xtext estendibili e riusabili. Un possibile sviluppo futuro è rappresentato dall'integrazione con GMF [Ecl04], un framework per il design grafico di modelli e generazione automatica di codice;

- **comunità di supporto**: è disponibile una comunità di supporto molto attiva e una documentazione dettagliata, non paragonabile a quelle offerte da IMP e DLTK, giudicate scarse.

Xtext utilizza internamente tre strumenti:

- **EMF** [BSE+03]: un framework per la modellazione che permette di costruire e manipolare modelli strutturati, fornendo supporto per la generazione automatica di codice. In Xtext EMF è utilizzato a più livelli d'astrazione, ma l'utilizzo principale è per la rappresentazione della grammatica e dell'albero di sintassi astratta dei file sorgente del linguaggio target;

- **ANTLR Parser Generator** [ParQuo95]: utilizzato per la generazione del parser del linguaggio target. ANTLR permette di generare un parser LL(k) a discesa ricorsiva con un ottimo supporto per l'*error recovery*, che permette ad esempio la creazione dello AST anche in presenza di errori nel sorgente. All'interno di Xtext ha il compito di generare il parser che sarà utilizzato per fornire all'IDE tutte le funzionalità classiche. Poichè una parte della tesi è dedicata all'analisi del

parser attuale di JOLIE e alla sua sostituzione con il parser generato da Xtext, ANTLR sarà descritto in maniera più approfondita nel sesto capitolo.

- **Google Guice** [Van08]: un framework per *dependency injection*, un principio introdotto dal framework Spring [JHA+08], utilizzato per associare dati di configurazione a porzioni di codice Java tramite annotazioni, permettendo riuso del codice e supporto non invasivo al testing. Nella terza sezione di questo capitolo è descritto l'utilizzo all'interno di Xtext.

Il framework più importante utilizzato da Xtext è sicuramente EMF, vediamo quindi una breve descrizione di questo versatile framework di modellazione.

4.1 EMF

EMF è un framework per la modellazione in Java utilizzato principalmente all'interno della piattaforma Eclipse. Il termine modellazione definisce un'architettura progettuale proposta da OMG, alla quale ci si può riferire con l'acronimo *Model Driven Architecture* (MDA). Senza entrare troppo nel dettaglio di questo pattern architetturale, possiamo definire un modello come una descrizione astratta di un sistema che nasconde informazioni e alcuni aspetti dell'implementazione del sistema stesso [BezGer01].

EMF nasce dalla necessità di unificare tre differenti modelli di uno stesso sistema:

- il codice Java;
- il class diagram UML corrispondente;
- il file XML corrispondente.

EMF permette di definire un'architettura in uno dei tre modelli e di generare automaticamente i restanti due. Volendo definire una certa entità di

un sistema software, tramite EMF è possibile ad esempio descrivere le caratteristiche dell'entità in XML e lasciare al framework il compito di generare il modello UML e il codice Java corrispondente. Lo schema in figura riassume questo concetto.

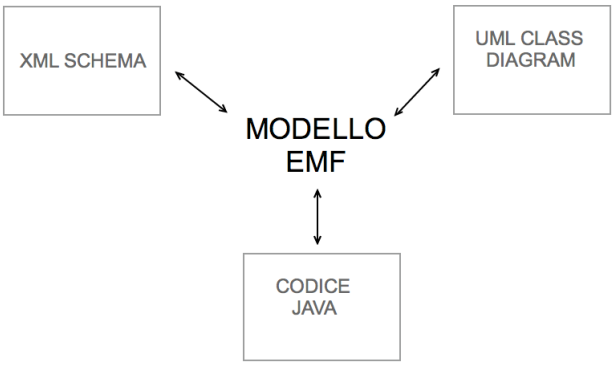

Figura 4.1: Schema concettuale di *Eclipse Modeling Framework*

EMF non può essere definito un approccio alla MDA, ma si posiziona esattamente a metà tra la programmazione e la modellazione, conservando di ognuna gli aspetti positivi. Per utilizzare un linguaggio di modellazione di alto livello come UML, infatti, occorre studiare sintassi e semantica del linguaggio e comprendere il rapporto esistente tra il modello e il codice corrispondente. Con EMF questo non è necessario poiché è possibile modellare utilizzando gli stessi concetti del linguaggio Java, come classi, metodi e attributi.

Un modello EMF è sostanzialmente un *class diagram* di UML, descritto utilizzando lo stesso tipo di astrazione delle classi e dei metodi. Gli attributi di un'entità, per esempio, sono rappresentati da una coppia di metodi (i *getter* e i *setter*) del modello EMF corrispondente e all'interno del framework, oltre alla normale funzione, sono utilizzati per le notifiche ad altri elementi dell'architettura e per funzioni di persistenza.

Un meta-modello è invece la specifica di un'astrazione, che identifica un insieme di aspetti rilevanti e un serie di relazioni fra questi.

Un meta-modello permette quindi di estrarre un modello da un sistema, ed è possibile creare diversi modelli dello stesso sistema utilizzando differenti meta-modelli.

4.1.1 EMF: il meta-modello Ecore

Il modello utilizzato per rappresentare modelli EMF è chiamato meta-modello Ecore. Ecore è definito usando lo stesso meta-modello Ecore, per cui Ecore è allo stesso tempo meta-modello e meta-meta-modello.

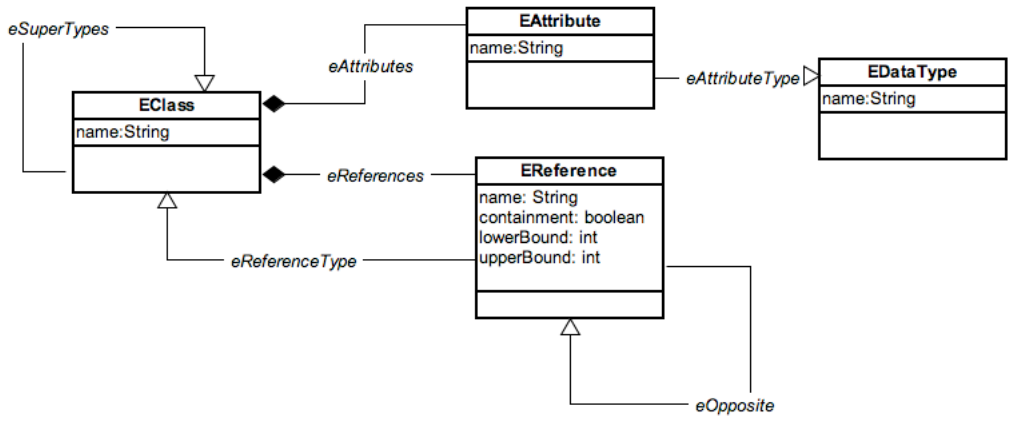

Figura 4.2: Una semplificazione del meta-modello Ecore

Il modello in figura 4.2 è una parte semplificata del meta-modello Ecore e rappresenta quattro *Ecore Class*:

- **EClass**: è utilizzata per rappresentare le classi del modello. Ogni classe ha un nome, zero o più attributi e zero o più referenze. In Java e UML questo elemento è rappresentato dalle classi, mentre in XML tramite definizione di *complex type*;

- **EAttribute**: è utilizzata per rappresentare gli attributi delle EClass;
- **EReference**: rappresenta un lato di un'associazione fra EClass. Contiene un attributo *flag* che rappresenta il contenimento e un *reference target type* che rappresenta la EClass a cui è legata
- **EDataType**: rappresenta il tipo di un attributo, un Eint o uno EString ad esempio.

I nomi delle classi corrispondono agli stessi nomi utilizzati in UML per il diagramma delle classi, poiché EMF è sostanzialmente un sottoinsieme semplificato di UML, che è lo standard *de facto* dei linguaggi di modellazione.

Un modello EMF è quindi un'istanza del modello Ecore, ed è chiamato *core model*.

Ritornando a quanto detto in precedenza, EMF permette di definire la stessa architettura con tre approcci differenti. Partendo dalla definizione di classi e interfacce Java, il framework è in grado di analizzarle e creare il modello corrispondente, a patto di utilizzare opportune Java annotation,*@model* in primis.

Allo stesso modo il modello può essere creato partendo da un XML *Schema*.

L'altro approccio possibile è quello di lavorare direttamente sul core model, utilizzando strumenti di design diretto di modelli EMF come *EclipseUML Graphical Editor* o l'editor base per Eclipse contenuto nel framework. E' inoltre possibile importare modelli UML ed esportarli allo stesso modo.

I core model utilizzano *XML Metadata Interchange* (XMI) come rappresentazione canonica, un file Ecore XMI rappresenta quindi lo standard di serializzazione dei metadati utilizzati da EMF.

4.1.2 EMF: Generazione del codice

Un'importante caratteristica di EMF (utilizzata ampiamente in Xtext) è la generazione di codice. Partendo da un core model tramite il wizard integrato nel framework è possibile far generare le classi Java corrispondenti.

Un EClass corrisponde ad un'interfaccia Java e a una sua implementazione. Ogni interfaccia estende l'interfaccia EObject, che corrisponde alla classe Object di Java e rappresenta la base di ogni modello, e ogni implementazione generata eredita da EObject una serie di metodi:

- **eClass()**: restituisce la EClass dell'oggetto, implementa quindi il concetto di *reflection* in EMF;
- **eContainer()** e **eResource()**: restituisce il container dell'oggetto e la EResource associata;
- **eGet()**,**eSet()**,**eIsSet()** e **eUnset()**: permettono l'accesso e la modifica dell'oggetto.

Lo EContainer definisce il collegamento con la EClass padre all'interno del modello EMF. Ogni modello è memorizzato all'interno di una EResource, un'interfaccia che rappresenta la *location* fisica in cui risiede il file XMI del modello, che come abbiamo visto rappresenta l'interfaccia di persistenza.

Un ResourceSet è un insieme di risorse e serve a gestire collegamenti fra modelli. In Xtext ad esempio è utilizzata per gestire referenze incrociate e linking fra diversi file sorgente del linguaggio target.

L'interfaccia EObject inoltre estende Notifier, che si occupa di gestire funzioni di notifica implementate tramite il design pattern *observer*. Gli oggetti che si occupano di osservare altri oggetti in EMF sono chiamati *Adapter*. Ogni EObject possiede una AdapterList contenente tutti gli Adapter che sono in attesa di ricevere notifiche: quando un attributo dell'EObject viene motificato tramite il metodo set() corrispondente, la notifica del cambiamento arriva ad ogni componente dell'AdapterList. Gli adapter possono essere utilizzati anche per estendere le funzionalità degli oggetti che stanno osservando.

Il framework genera inoltre due importanti classi, un Efactory e un Epackage, utilizzate come pattern creazionali per istanziare nuovi oggetti. Xtext utilizza lo Epackage per creare i costrutti della grammatica del linguaggio

target durante la fase di costruzione dello AST di un file sorgente. Il framework inoltre produce automaticamente *switch class*, che implementando il design pattern visitor permettono di navigare il modello. Xtext ad esempio utilizza switch class per navigare all'interno dell'AST di un file sorgente.

4.2 Configurazione di un progetto Xtext

Il framework Xtext è distribuito come plug-in per Eclipse ed è dotato di un wizard in grado di guidare il processo d'implementazione di un linguaggio e la generazione dell'IDE.

Dopo aver definito nome del linguaggio ed estensioni associate, *.ol* e *.iol* nel caso di JOLIE, il framework produce due progetti: uno contenente la definizione della grammatica per il linguaggio e componenti utilizzati a runtime come *linking provider*, *scope provider* e *validation provider*, e un altro contenente gli aspetti legati all'interfaccia dell'IDE come l'editor, *label provider* (per gestire lo outline) e *navigator*. Xtext può essere, infatti, utilizzato anche in modalità stand-alone, ed in questo caso il secondo progetto non è necessario.

Entrambi i progetti sono coposti in parte da componenti generati a partire dalla grammatica e dal file di configurazione, ed in parte da codice scritto a mano, come la grammatica stessa e le classi utilizzate per adattare il codice generato per ottenere comportamento desiderato. Tutte le librerie richieste sono richiamate utilizzando il *Manifest file* secondo OSGI.

4.2.1 La grammatica Xtext per JOLIE

Il primo passo nell'implementazione di un linguaggio tramite Xtext è la definizione della grammatica del linguaggio. Xtext non è semplicemente uno strumento per l'integrazione di un linguaggio in un IDE, ma un framework per la creazione di linguaggi in grado di produrre componenti come parser e interprete utilizzabili anche in modalità stand-alone, senza l'ausilio di Eclip-

se. L'editor per il linguaggio è quindi solo un componente ottenibile ad un bassissimo costo.

La grammatica per JOLIE è stata scritta analizzando il parser attuale e semplici test, utilizzando quindi *reverse engineering* per costruire tutte le produzioni o regole di derivazione. L'appendice di questa tesi contiene la grammatica per Xtext completa del linguaggio JOLIE e una breve descrizione della semantica di ogni istruzione. Il sesto capitolo inoltre descrive meglio il processo di definizione della grammatica e l'analisi del parser attuale di JOLIE.

Una grammatica nel contesto di Xtext ha una duplice funzione: serve a descrivere la sintassi concreta e permette di definire la struttura del modello (o AST) che verrà creato dal parser.

```
grammar jolie.Xtext.Jolie
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
generate jolie "http://www.Xtext.org/Jolie"
```

Il codice sopra rappresenta l'intestazione del file della grammatica. La prima riga definisce il nome (Jolie), che deve corrispondere al file .Xtext che si sta editando, così come in Java il nome della classe deve corrispondere al nome del file. La dichiarazione di import serve per poter utilizzare il metamodello Ecore, che come abbiamo visto rappresenta la matrice di un modello EMF.

L'istruzione successiva, *generate*, identifica infatti il core model che sarà generato dalla grammatica, Jolie.ecore, che è utilizzabile tramite l'Ecore package corrispondente, JoliePackage.

Un Ecore package, come abbiamo visto, è una factory utilizzata per creare oggetti in EMF, e in Xtext viene utilizzata per definire quali saranno le EClass che comporranno l'AST che rappresenta un file sorgente del linguaggio target. In altre parole, l'AST di un file sorgente è un modello EMF, definito come un'istanza del core model della grammatica, a sua volta definito come un'istanza dell'Ecore model.

In Xtext e possibile definire all'interno della grammatica il tipo dell'elemento che sarà aggiunto all'albero tramite l'istruzione *returns*. Se l'istruzione

returns e omessa, il tipo è lo stesso definito dal nome della regola.

Per collegare diversi oggetti insieme occorre assegnare gli oggetti restituiti dalla regola ad una *feature* della regola corrente. Una feature, nel contesto di EMF e Xtext, è un attributo o una reference della EClass che rappresenta l'oggetto.

Vediamo un esempio concreto per comprendere meglio come la grammatica Xtext e il framework EMF sono collegati fra loro, considerando un piccolo frammento della grammatica di JOLIE e un semplice file sorgente composto da due istruzioni di assegnamento.

```
....
Program:
        ...
        main=Main?;

Main:
        name='main' mainrocess=MainProcess;

MainProcess:
         '{' parallelStatement=ParallelStatement '}';

ParallelStatement:
        {ParallelStatement} (children+=SequenceStatement
           (VERT children+=SequenceStatement)*);

SequenceStatement:
        {SequenceStatement} (children+=BasicStatement
           (SEMICOLON children+=BasicStatement)*);

BasicStatement:
        ...
        assignStOrPstIncDecrement=
           AssignStOrPstIncDecrement |
          ...;

AssignStOrPstIncDecrement:
           variablePath=VariablePath rightSide=RightSide;

RightSide:
        { ASSIGN expression=Expression
           | CHOICE | DECREMENT |
              POINTSTO variablePath=VariablePath
                   | DEEPCOPYLEFT variablePath=VariablePath;
```

```
Expression:
        TerminalExpression (...)?;

TerminalExpression returns Expression:
        '(' Expression ')'
           | {IntLiteral} MINUS? value=INT | ... ;
```

Queste sono solo alcune delle produzioni della grammatica, ma sono le uniche necessarie per riconoscere il seguente programma in JOLIE:

```
main {

  foo = 3;
  fee = 2

}
```

Xtext utilizza EMF in due momenti separati. In un primo momento, durante la fase di configurazione e generazione del linguaggio, è generato il core model della grammatica, creando un'istanza dell'Ecore model. Una versione semplificata del core model della grammatica è presentata in figura 4.3.

Il mata-modello è definito tramite Ecore, quindi con una serie di EClass, EReference, EAttribute e EDataType. Gli EAttribute sono i campi all'interno dei rettangoli (le EClass), mentre le ERefernece sono rappresentate dalle linee di collegamento fra le classi.

Il secondo momento in cui entra in gioco EMF è durante la fase di parsing. Tramite le istruzioni returns e gli assegnamenti alle feature delle regole, all'interno della grammatica è possibile definire la struttura del modello che forma l'AST, l'output del processo di parsing. L'AST di un file JOLIE quindi è un modello EMF, un'istanza del core model della grammatica, ed è la struttura che sarà utilizzata dai componenti che operano dopo il processo di parsing come validator o interprete. In figura 4.4 si può osservare l'AST corrispondente al codice JOLIE visto precedentemente.

Le istruzioni di assegnamento e returns servono per modellare l'AST, permettendo di definire nomi e struttura delle feature che saranno utilizzati

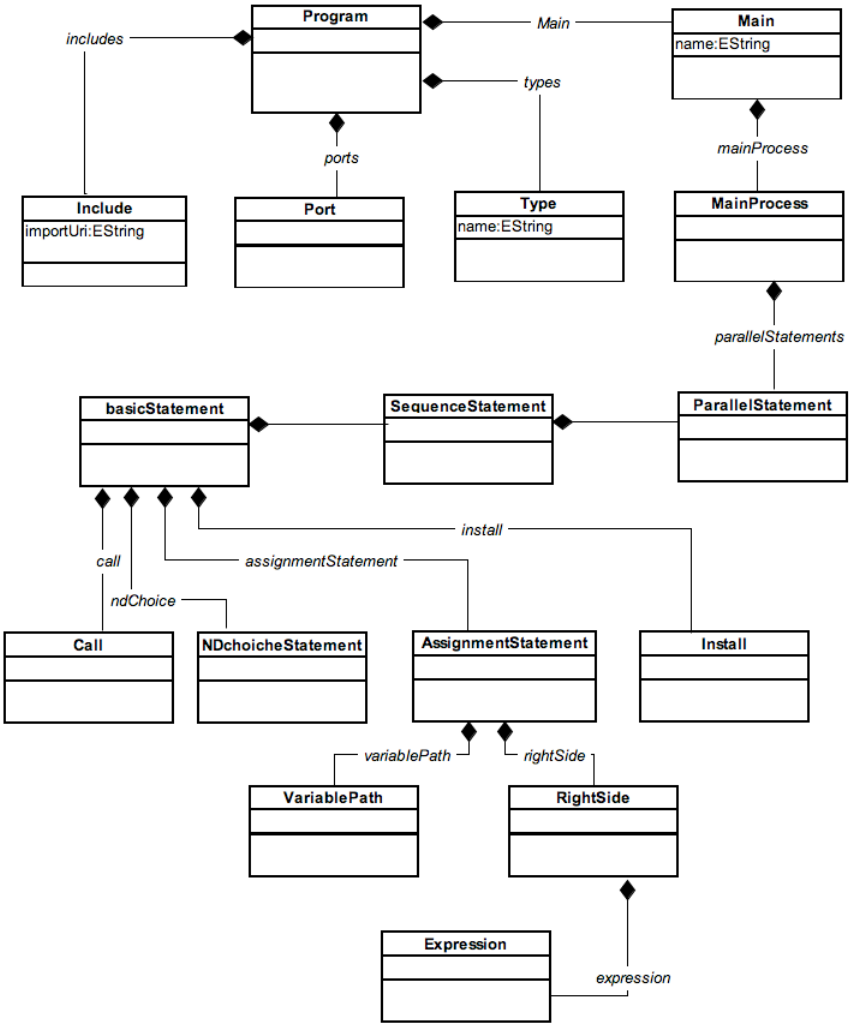

Figura 4.3: Il core model EMF semplificato della grammatica di JOLIE

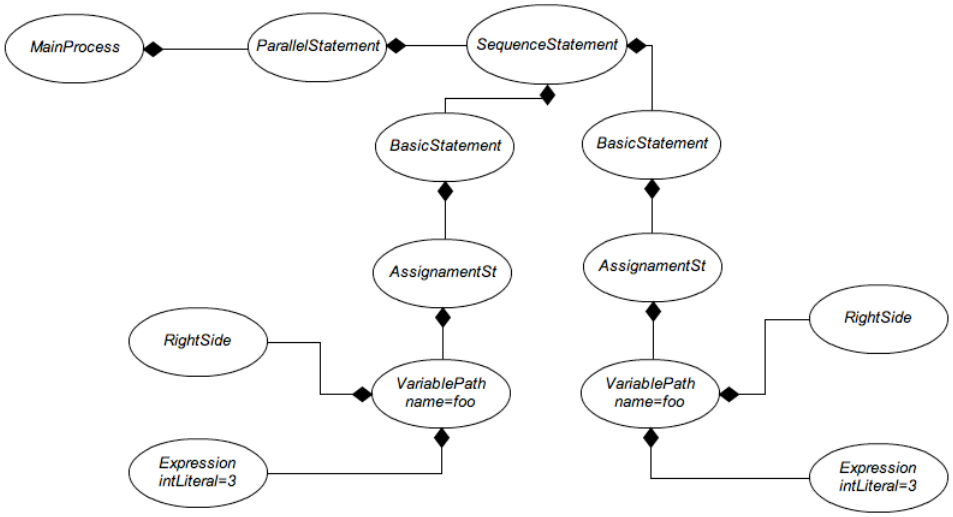

Figura 4.4: Il modello EMF di un AST di JOLIE

successivamente. Esistono in realtà altre tecniche per modellare lo AST, come le azioni semplici e le azioni con assegnamento, descritte nella prossima sezione.

L'utilizzo di EMF all'interno di Xtext può essere quindi riassunto in figura 4.5.

4.2.2 La sintassi della grammatica

Durante la fase di analisi lessicale del processo di parsing, una sequenza di caratteri (il file sorgente) è trasformata in una sequenza di token. A questo livello d'astrazione un token può essere visto come un dato fortemente tipato della sequenza di input, poichè ad ogni token è associato un simbolo terminale o una parola chiave. In Xtext i simboli terminali sono identificati da produzioni con l'elemento a sinistra in maiuscolo, precedute dalla parola chiave *terminal*. Per esempio:

```
terminal SEMICOLON:
        ';';
```

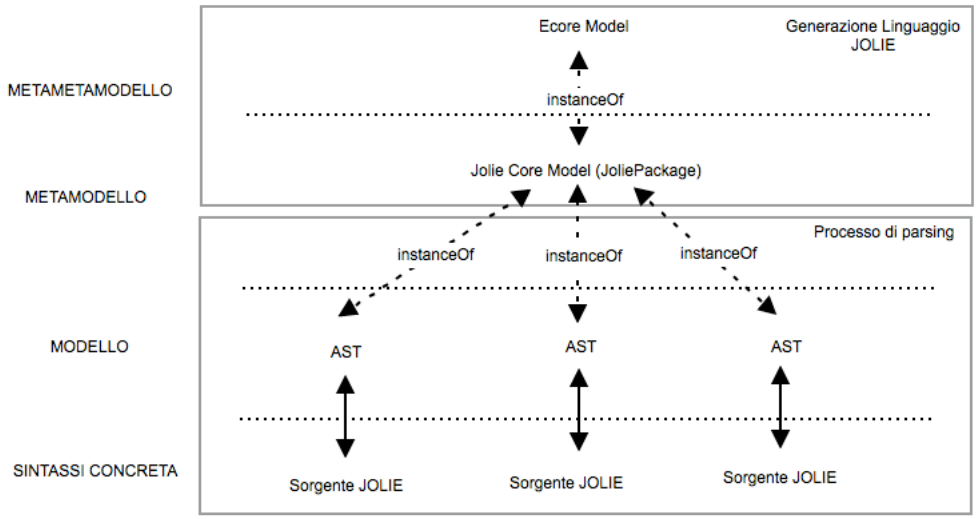

Figura 4.5: EMF in Xtext

associa ad ogni carattere ';' la produzione terminale denominata SEMICOLON. La definizione formale di regola terminale in Xtext, contenuta nel file della grammatica di Xtext, è la seguente:

```
TerminalRule :
  'terminal' name=ID ('returns' type=TypeRef)? ':'
    alternatives=TerminalAlternatives ';'
;
```

Abbiamo in precedenza parlato dell'istruzione return, utilizzata insieme agli assegnamenti per modellare la struttura dal core model della grammatica, lo EPackage. Le produzioni terminali hanno come valore di restituzione una stringa, ma è possibile modificare lo EDataType utilizzato nel core model nel modo seguente:

```
terminal INT returns ecore::EInt:
        ('0'..'9')+;
```

Nel core model ogni regola di tipo INT sarà quindi identificata con un EDataType Eint.

Le produzioni terminali sono descritte utilizzando una notazione simile alla EBNF, che perette l'utilizzo di operatori che rendono la scrittura della grammatica più semplice e il lavoro ultimato maggiormente leggibile.

Gli operatori utilizzabili all'interno delle regole terminali sono:

- **operatori di cardinalità**: è possibile tramite "?", "*" e "+", modificare la cardinalità di una regola. Nella produzione terminale INT precedente, il "+" sta ad indicare "almeno uno","*" identifica "zero o più", "?" presenza o assenza mentre l'operatore di default, vuoto, "esattamente un";

- **range di caratteri**: sempre nella regola INT, tramite l'operatore ".." è possibile definire un range, in questo caso da zero a nove;

- **wildcard**: è possibile utilizzare l'operatore "." per esprimere "qualsiasi carattere". Per esempio:

```
FOO : 'f' . 'o';
```

Permette le stringhe del tipo fao, fbo, ecc;

- **until token**: permette di identificare una sequenza di caratteri delimitata, ad esempio:

```
terminal ML_COMMENT : '/*' -> '*/';
```

identifica i commenti su più linee;

- negazione: il carattere "!" inverte la semantica di ogni comando precedente;

- **invocazione di un'altra regola**: è possibile utilizzare altre produzioni terminali per definirne una nuova:

```
    terminal DOUBLE : INT '.' INT;
```

- **alternative**: è possibile utilizzare l'operatore "|" per definire più alternative per la stessa produzione terminale. Per esempio:

```
terminal WS : (' '|'\t'|'\r'|'\n')+;
```

- **gruppi**: tramite le parentesi tonde è possibile definire gruppi di caratteri:

```
terminal ASCII : '0x' ('0'..'7') ('0'..'9'|'A'..'F');
```

L'altro tipo di regole è rappresentato dalle regole di parsing o produzioni, che permettono di costruire un albero di simboli terminali e non terminali (chiamato *parse tree* o *node model* in Xtext) partendo da una sequenza di simboli terminali. Il node model rappresenta quindi l'albero di sintassi concreta, e contiene più informazioni sintattiche rispetto allo AST. Anche i node model sono rappresentati da opportuni modelli EMF.

Le istruzioni di return, le azioni e gli assegnamenti sono utilizzati per definire la struttura degli EObject che comporranno lo AST.

Le produzioni hanno in comune con le regole terminali i gruppi, le alternative e l'invocazione di altre regole, oltre ad una serie di operazioni utilizzate per modellare la struttura dell'AST, qui descritte brevemente.

Assegnamento

Gli assegnamenti sono utilizzati per assegnare informazioni alle feature (attribuiti o referenze) della EClass di una produzione. Il nome della EClass è definito dal valore di return, se l'istruzione return è omessa la EClass prende il nome della produzione. Il tipo di una feature della EClass invece è derivato dal lato destro dell'istruzione di assegnamento. Esistono tre operatori di assegnamento: "=" per l'assegnamento singolo di una feature con un unico valore, "+=" per l'assegnamento di una feature che contiene una lista di valori, e "?=" per una feature di tipo EBoolean. Vediamo un esempio concreto:

```
Interface:
        'interface' name=ID '{'
        (requestResponseOperation+=RequestResponseOperation? &
           oneWayOperation+=OneWayOperation?)
        '}';
```

La seguente produzione corrisponde a una EClass di nome Interface nel core model. Interface avrà tre feature:

- *nome*, un EAttribute di tipo EString (ID è un terminale, tutti i terminali hanno tipo EString);
- *requestResponseOperation*, di tipo Elist<RequestResponseOperation>, una lista di referenze alle EClass che rappresentano le operazioni di tipo Request Response;
- *oneWayOperation*, un EReference di tipo Elist <OneWayOperation>, una lista di referenze alle EClass che rappresentano le operazioni di tipo One Way.

La EClass Interface è creata durante la generazione del linguaggio, mentre durante la fase di parsing, quando una regola di questo tipo sarà riconsciuta un EObject istanza di questa classe sarà aggiunto all'AST che rappresenta il file.

Referenze incrociate

Xtext permette referenze incrociate all'interno della grammatica.

Nei compilatori la fase di linking avviene normalmente dopo il processo di parsing, Xtext funziona allo stesso modo ma è possibile comunicare con il linker, il componente che serve a questo scopo, direttamente al livello della grammatica. Le referenze incrociate permettono di implementare una funzionalità molto utile nell'IDE del linguaggio target, la navigabilità del codice. Vediamo un esempio di utilizzo in JOLIE:

```
Type:
        'type' name=ID COLON native_type_sub+=Native_type typedef+=Typedef?;

Typedef:
        {Typedef} '{'
        subtypes+=Subtypes
        '}';

Subtypes:
        {Subtypes} (DOT name+=ID Cardinality? COLON
           native_type_sub+=Native_type typedef+=Typedef*)* | {Subtypes}
             QUESTION;

Cardinality:
        QUESTION | \ac{AST}ERISK | ('[' INT COMMA (INT | ASTERISK) ']');

Native_type:
        {Native_type_sub} ("any" | "int" | "real" | "string" | "void" |
           'undefined' | 'double' | 'raw' | type=[Type]);
```

Il frammento di grammatica si riferisce alla definizione dei tipi in JOLIE. E' possibile creare tipi composti, come nel seguente caso:

```
...
type JavaExceptionType:string {
        .stackTrace:string
}
...
type IOExceptionType:JavaExceptionType
```

All'utlima riga del frammento di grammatica, si può notare che una delle possibili alternative della produzione NativeType è type=[Type]. Questa è

un esempio di referenza incrociata, utilizzata per definire collegamenti interni alla grammatica. La feature type della EClass NativeType sarà una EReference di tipo Type. Il valore tra parentesi graffe non rappresenta una regola terminale o una produzione, ma il tipo restituito da una produzione (definito dal'istruzione return dopo la regola o dal nome della regola se return non è specificato).

Le referenze incrociate sono utilizzate dal linker del linguaggio per implementare la gestione dello scope, l'effetto visibile durante l'utilizzo dell'IDE è invece la navigabilità del codice: nel frammento sopra ad esempio, tenendo premuta una sequenza di tasti opportuna (tipiamente CTRL o CMD), al passaggio del cursore sul tipo di IOExceptionType è attivato un collegamento che permette di raggiungere la definizione del tipo JavaExceptionType con un semplice click. Questa funzione risulta particolarmente utile perchè è attiva su tutti i documenti aperti nel workspace, che in JOLIE possono essere collegati tramite l'istruzione *include*.

Azioni semplici

Come abbiamo già detto il tipo dell'EObject restituito da una produzione è definito dall'istruzione return o dal nome della produzione. L'oggetto viene creato con una politica lazy al momento del primo assegnamento su una feature. Tramite le azioni è comunque possibile esplicitare la creazione del tipo restituito, come nel seguente caso:

```
BasicStatement:
        {Process} process=Process |
        {AssignStatement} assignStatement=AssignStatement | ...
```

Prima dell'assegnamento assignStatement=AssignStatement le parentesi graffe sono utilizzate per definire l'azione che identifica il tipo restituito dalla regola. In questo caso quando sarà riconosciuta una regola di tipo AssignStatement sarà creato un EObject di tipo AssignStatement che estende BasicStatement, senza l'utilizzo dell'azione invece il risultato sarebbe stato

la creazione di un EObject di tipo BasicStatement contenente una referenza ad un EObject di tipo AssignStatement.

Il tipo dell'azione tra parentesi graffe può essere un tipo qualsiasi, anche non definito da altre produzioni della grammatica. Un'ulteriore possibilità per forzare il tipo restituito è quella di utilizzare la chiamata ad un'altra produzione senza assegnamento, in questo caso il tipo dell'EObject risultato della produzione è il tipo della produzione invocata.

Azioni con assegnamento

Il parsing di tipo LL(k) ha come pregio rispetta all'algoritmo LR(k) una migliore gestione degli errori che permette di costruire AST anche nel caso in cui questi siano presenti nel file sorgente. Il difetto maggiore di una grammatica LL(k) è che non permette la cosiddetta ricorsione sinistra:

```
Expression :
  Expression '+' Expression |
  '(' Expression ')' |
  INT
;
```

La produzione sopra presenta appunto questo problema, il primo termine della produzione è la produzione stessa. Il pattern utilizzato per eliminare questa situazione è conosciuto come fattorizzazione sinistra, ad esempio il codice precedente può essere trasformato in maniera abbastanza semplice in:

```
Expression :
  TerminalExpression ('+' TerminalExpression)?
;

TerminalExpression :
  '(' Expression ')' |
  INT
;
```

Utilizzando assegnamenti e azioni la grammatica Xtext risultante per le espressioni è la seguente:

```
Expression :
  {Operation} left=TerminalExpression (op='+' right=TerminalExpression)?
```

```
;

TerminalExpression returns Expression:
  '(' Expression ')' |
  {IntLiteral} value=INT
;
```

ma il risultato che si ottiene ad esempio dopo il parsing dell'espressione "5" è la costruzione di un EObject di tipo Operation, contenente una EReference ad un altro EObject di tipo Operation che a sua volta contiene una EReference ad un EObject di tipo IntLiteral. L'AST risultante è quindi molto complesso rispetto al contenuto semantico, ma utilizzando chiamate a produzioni senza assegnamento e azioni con assegnamento è possibile risolvere il problema:

```
Expression :
  TerminalExpression ({Operation.left=current}
    op='+' right=Expression)?
;

TerminalExpression returns Expression:
  '(' Expression ')' |
  {IntLiteral} value=INT
;
```

L'azione con assegnamento Operation.left=current è chiamata azione di riscrittura dell'albero e assegna alla feature left dell'EObject Operation creato dalla regola, l'Expression restituita dalla TerminalExpression di sinistra. In questo modo se un'espressione contiene un unico valore intero, l'EObject risultante sarà un istanza di IntLiteral, mentre nel caso di un espressione come una somma di due numeri avremo un istanza di Operation con una feature contenente l'operatore "+" e due feature (left e right) con le referenze a due istanze di IntLiteral.

4.3 Il generatore di Xtext

Il passo successivo alla definizione della grammatica Xtext del linguaggio target consiste nella generazione dei componenti del linguaggio.

Il generatore di Xtext utilizza *Modelling Workflow Engine 2* per la configurazione.

4.3.1 MWE2

MWE è un framework per l'orchestration di componenti di modellazione all'interno della piattaforma Eclipse, e MWE2 è una evoluzione retrocompatibile del framework implementata appositamente per Xtext.

MWE2 è un engine per la generazione di componenti configurabile in maniera dichiarativa, creato per gestire workflow. Un workflow consiste in una serie di componenti che interagiscono fra di loro, come ad esempio componenti per leggere risorse EMF, componenti per eseguire trasformazioni su questi modelli e componenti per generare qualsiasi tipo di artefatto derivato.

Il framework mappa un componente definito nel file .MWE2 con una classe java opportuna, ed è possibile configurare e comporre istanze di queste classi senza dover modificare il codice Java. Il meccanismo di funzionamento è quindi molto simile al concetto di dependency injection utilizzato dal framework Spring, che utilizza file XML per configurare oggetti.

Il principale aspetto positivo di questo sistema è il riuso del codice, che in un framework di grandi dimensioni come Xtext è molte importante. Utilizzando MWE2 è infatti possibile configurare e generare in poche righe di codice tutte i componenti che saranno poi utilizzati dall'IDE del linguaggio target, nel nostro caso **Joliepse!** (**Joliepse!**).

Cosi come per il file *.xtext* che contiene la grammatica di un linguaggio, anche il linguaggio utilizzato nei documenti MWE2 è implementato tramite lo stesso Xtext, sono quindi disponibili funzioni come il controllo sintattico e proposal provider.

Un documento MWE2 è quindi utilizzato nel contesto di Xtext come configurazione del linguaggio, e contiene tutte le informazioni necessarie a generare i componenti.

Vediamo il codice del file GenerateJolie.MWE2:

```
1 module jolie.Xtext.Jolie
```

```
2
3  import org.eclipse.emf.mwe.utils.*
4  import org.eclipse.Xtext.generator.*
5  import org.eclipse.Xtext.ui.generator.*
6
7  var grammarURI = "classpath:/jolie/Xtext/Jolie.Xtext"
8  var file.extensions = "iol,ol"
9  var projectName = "jolie.Xtext"
10 var runtimeProject = "../${projectName}"
11
12 Workflow {
13             bean = StandaloneSetup {
14     platformUri = "${runtimeProject}/.."
15   }
16   component = DirectoryCleaner {
17     directory = "${runtimeProject}/src-gen"
18   }
19   component = DirectoryCleaner {
20     directory = "${runtimeProject}.ui/src-gen"
21   }
22   component = Generator {
23     pathRtProject = runtimeProject
24     pathUiProject = "${runtimeProject}.ui"
25     projectNameRt = projectName
26     projectNameUi = "${projectName}.ui"
27     language = {
28       uri = grammarURI
29       fileExtensions = file.extensions
30
31     // Java API to access grammar elements
32     //(required by several other fragments)
33       fragment = grammarAccess.GrammarAccessFragment {}
34
```

```
35      // generates Java API for the generated EPackages
36      fragment = ecore.EcoreGeneratorFragment {
37      // referencedGenModels = "uri to genmodel, uri to next genmodel"
38      }
39
40      // the serialization component
41      fragment = parseTreeConstructor.ParseTreeConstructorFragment {}
42
43      // a custom ResourceFactory for use with \ac{EMF}
44      fragment = resourceFactory.ResourceFactoryFragment {
45        fileExtensions = file.extensions
46      }
47
48      // The antlr parser generator fragment.
49      fragment = parser.antlr.XtextAntlrGeneratorFragment {
50        options = {
51          backtrack = true
52        }
53
54      }
55
56      // java-based API for validation
57      fragment = validation.JavaValidatorFragment {
58        composedCheck = "org.eclipse.Xtext.validation.NamesAreUniqueValidator"
59      }
60
61
62      // scoping and exporting API
63      fragment = scoping.ImportURIScopingFragment {}
64      fragment = exporting.SimpleNamesFragment {}
65                // formatter API
66      fragment = formatting.FormatterFragment {}
67
```

```
68      // labeling API
69      fragment = labeling.LabelProviderFragment {}
70
71      // outline API
72      fragment = outline.TransformerFragment {}
73
74                                  // quickfix API
75      fragment = quickfix.QuickfixProviderFragment {}
76
77      // content assist API
78      fragment = contentAssist.JavaBasedContentAssistFragment {}
79
80      // generates a more lightweight Antlr parser
81      // and lexer tailored for content assist
82      fragment = parser.antlr.XtextAntlrUiGeneratorFragment {}
83
84      // project wizard (optional)
85       fragment = projectWizard.SimpleProjectWizardFragment {
86          generatorProjectName = "${projectName}.generator"
87          modelFileExtension = file.extensions
88       }
89    }
90  }
91 }
```

L'interfaccia IGeneratorFragment che ogni elemento implementa estende la classe AbstractGeneratorFragment, il cui compito principale è quello di delegare la generazione del codice a un template di XPand.

XPand è un linguaggio per eseguire trasformazioni *Model to Text* (M2T). Partendo da un qualunque modello, con XPand è possibile definire in un template che tipo di operazioni eseguire su ogni elemento del modello. L'uso principale all'interno di Xtext è la generazione del codice degli elementi del linguaggio.

I *component* sono annidati all'interno di una radice (Workflow). Le variabili nella prima parte del MWE2 servono a definire l'URI del file della grammatica, le estensioni associate al linguaggio e le cartelle in cui le saranno generati i vari moduli o componenti del linguaggio.

Il component DirectoryCleaner, alle righe 16 e 19, è utilizzato per ripulire le directory corrispondenti prima di una generazione. A questo component è quindi associata una classe DirectoryCleaner che viene istanziata due volte con parametri diversi. Il più importante è il componente Generator, che è formato da una serie di IGeneratorFragment. Ogni fragment corrisponde ad un elemento del linguaggio:

- **GrammarAccessFragment**: il file della grammatica (jolie.Xtext) è trasformato in un modello EMF, da questo modello è generato JolieGrammarAccess.java che offre API per l'accesso alla grammatica utilizzate dai fragment successivi;

- **EcoreGeneratorFragment**: è il fragment più importante, permette di costruire il meta-modello della grammatica, utilizzato per produrre gli AST dei file del linguaggio. Nelle sezioni precedenti è stato discusso questo processo, che come risultato finale ha la creazione del JoliePackage che contiene il modello della grammatica;

- **ParseTreeConstructorFragment**: questo componente è utilizzata durante il processo di serializzazione, che consiste nel salvataggio della versione testuale di un AST relativo a un sorgente JOLIE. Questa operazione è quindi l'inversa del processo di parsing. JolieParseTreeContructor, generato da questo fragment, si occupa di ricostruire lo stream di token utilizzando JolieGrammarAccess per recuperare le informazioni sintattiche che non erano necessarie nello AST. Questo modulo è utilizzato ad esempio per implementare la funzione di auto formattazione del testo all'interno dell'IDE;

- **ResourceFactoryFragment**: Xtext utilizza lo stesso concetto di Resource e ResourceSet di EMF. Una XtextResource è un oggetto che na-

sconde all'esterno il processo di parsing e di serializzazione, utilizzabile anche senza il supporto di Eclipse. Il componente generato da questo fragment crea un Factory di XtextResource che fornisce API per l'utilizzo di tutte le funzionalità di Xtext, utilizzabile in altre applicazioni o durante la fase di adattamento delle funzionalità dell'IDE;

- **XtextAntlrGeneratorFragment**: questo fragment è utilizzato per produrre il parser del linguaggio con ANTLR. La grammatica Xtext è modificata e adattata a quella richiesta dal parser generator (InternalJolie.g) la cui esecuzione produce InternaJolieParser.java e InternalJolieLexer.java. Tramite le estensioni di queste classi, i servizi JolieParser.java e JolieAntlrTokenFileProvider sono resi disponibili a tutti gli altri elementi del linguaggio;

- **JavaValidatorFragment**: permette di generare AbstractJolieValidator. La validazione automatica dei file sorgenti avviene in tre livelli diversi e sarà discussa successivamente. La classe corrispondente a questo fragment permette invece di configurare e generare un validator in grado di rilevare errori e warning. Nell'IDE per il linguaggio JOLIE è utilizzato ad esempio per il costrutto with. Nelle prossime sezioni è descritto il funzionamento del validator.

- **ImportURIScopingFragment e SimpleNamesFragment**: generano i componenti che si occupano della gestione delle politiche di scoping all'interno del linguaggio. La gestione dello scoping sarà affrontata nel prossimo capitolo, anche se per quanto riguarda JOLIE lo Scope Provider è stato modificato unicamente per la gestione del costrutto *include*, poichè tutte le variabili hanno scope globale;

- **FormatterFragment**: la classe corrispondente genera JolieFormatter che permette di gestire le politiche di formattazione del codice sorgente che come abbiamo detto sono implementate tramite serializzazione di modelli;

- **LabelProviderFragment**: JolieLabelProvider permette di associare ad ogni elemento della grammatica un immagine ed un'etichetta. Questi elementi grafici sono utilizzati nell'interfaccia utente dell'IDE nell'outline e nella visualizzazione delle proposte d'inserimento e dei collegamenti;

- **TransformerFragment**: gestisce la visualizzazione dell'outline, normalmente inferita dalla struttura gerarchica della grammatica. Adattando JolieTransformer.java è comunque possibile influenzare la visualizzazione, per esempio nascondendo o evidenziando alcuni elementi;

- **QuickfixProviderFragment** : genera JolieQuickfixProvider che fornisce il supporto per i quick fix, che permettono di offrire proposte per la correzione di errori e warning rilevati dal validator.

- **JavaBasedContentAssistFragment e XtextAntlrUiGeneratorFragment**: creano il supporto per il proposal provider, altra caratteristica dell'IDE generato molto utile per la produttività degli sviluppatori del linguaggio. Il proposal provider si occupa di fornire suggerimenti conformi alla grammatica e sensibili al contesto durante la scrittura del codice. Quando ad esempio è definita una RequestResponse operation in JOLIE, al momento dell'inserimento dei tipi per la memorizzazione dei valori di input e output premendo la sequenza di tasti opportuna è possibile visualizzare la lista dei tipi base e dei tipi definiti nel documento corrente e nei documenti importati tramite l'istruzione *include*;

- **SimpleProjectWizardFragment**: permette di creare JolieProjectInfo.java, utilizzata per aggiungere la voce "New Jolie Project" tra quelle disponibili al momento della creazione di un nuovo progetto nell'IDE generato. E' possibile utilizzare Xpand per configurare questo componente, permettendo ad esempio di creare file di interfacce o servizi JOLIE con un contenuto prestabilito.

Eseguendo il workflow definito in GeneratoJolie.MWE2 è possibile generare il codice dei componenti che formeranno l'implementazione del linguaggio e l'IDE Eclipse-based di supporto.

Il già citato Google Guice, un framework per la dependency injection, è utilizzato per assemblare i componenti generati.

4.3.2 Google Guice in Xtext

In Xtext, Google Guice è lo strumento che permette di comporre i vari componenti del linguaggio utilizzando dependency injection, una tecnica di programmazione che consente un elevato riuso del codice e migliore supporto al mantenimento dell'architettura.

Indicativamente, Guice è un container che può essere utilizzato per richiedere le funzionalità offerte da un componente. Questa operazione in Java è esprimibile istanziando tramite l'operatore *new* l'oggetto dalla classe opportuna, mentre con Guice è sufficiente dichiarare al container che si vuole utilizzare le funzionalità fornite da quella determinata classe tramite l'annotazione Java *@Inject*.

La dipenza sarà poi gestita a tempo d'esecuzione dal container. Guice deve comunque essere configurato per conoscere come istanziare gli oggetti a tempo d'esecuzione, e la configurazione che mappa un tipo utilizzabile tramite @Inject con la classe concreta o le istanze già attive avviene all'interno dei cosiddetti module.

Xtext genera due module per il linguaggio:

- **JolieRuntimeModule**, utilizzato quando il linguaggio è esguito in modalità stand-alone,
- **JolieUiModule**, utlizzato da Eclipse per l'implementazione dell'IDE per il linguaggio target.

I module sono creati durante l'esecuzione del workflow. Ecco una parte del codice di AbstractJolieRuntimeModule, la classe astratta che JolieRuntimeModule estende:

```
1  public Class<? extends org.eclipse.Xtext.IGrammarAccess>
2         bindIGrammarAccess() {
3      return jolie.Xtext.services.JolieGrammarAccess.class;
4  }
5
6  public Class<? extends org.eclipse.Xtext.parsetree.reconstr.
7         IParseTreeConstructor>
8           bindIParseTreeConstructor() {
9      return jolie.Xtext.parseTreeConstruction.
10                JolieParsetreeConstructor.class;
11 }
12 ...
```

Il module serve per definire il collegamento tra i componenti utilizzati nel'implementazione del linguaggio e le classi concrete che sono state generate dal workflow.

I componenti possono quindi essere utilizzat internamente tramite le annotation @Inject da tutti gli elementi del linguaggio, riferendosi sempre all'interfaccia più generale, come IGrammarAccess e IParseTreeConstructor, permettendo quindi un grande riutilizzo del codice.

Riassunto capitolo

In questo capitolo è presentato il framework Xtext. Nella prima sezione è analizzato EMF, un framework per la modellazione utilizzato da Xtext per la rappresentazione della grammatica e degli alberi di sintassi astratta dei file sorgente.

La seconda sezione descrive la sintassi della grammatica di un linguaggio di Xtext analizzando parte delle produzioni della grammatica di JOLIE.

La terza sezione infine descrive il workflow di generazione dei componenti del linguaggio e come questi siano collegate fra loro tramite Google Guice.

Terminata la parte relativa alla configurazione del linguaggio, nel prossimo capitolo è descritto il modo in cui all'interno dell'IDE sono utilizzati

i componenti generati e come questi sono stati adattati per aumentare le funzionalità disponibili in Joliepse.

Capitolo 5

Joliepse IDE

In questo capitolo è discusso il funzionamento a tempo d'esecuzione dei componenti di Xtext e le modifiche apportate a questi elementi per realizzare l'IDE Joliepse.

Sia Xtext che i linguaggi generati con questo framework fanno largo uso del concetto di dependency injection di Google Guice per il funzionamento a tempo d'esecuzione. Il linguaggio generato può essere utilizzato sia all'interno di Eclipse, creando quindi un IDE come nel caso di Joliepse, oppure esternamente, all'interno di qualsiasi applicazione Java.

L'esecuzione del workflow MWE2 per la generazione del linguaggio produce un'implementazione dell'interfaccia ISetup, che ha la funzione di inizializzare i componenti del linguaggio e restituire un injector utilizzato per accedere ai vari elementi come parser e lexer quando non si utilizza la piattaforma Eclipse.

Un esempio di quest'approccio è analizzato nel prossimo capitolo, dove si descrive il processo di sostituzione del parser attuale di Jolie con quello generato da ANTLR in Xtext

Per la generazione dell'IDE invece, il framework ha bisogno di comunicare con la piattaforma target, e a questo scopo è generato un *activator* che ha il compito di informare Eclipse su come utilizzare il plug-in contenente l'IDE per il nuovo linguaggio. Anche l'activator utilizza Guice e fornisce alla

piattaforma tutti gli injector per utilizzare i componenti del linguaggio.

Vediamo quindi come tutti gli elementi generati da Xtext a partire dalla grammatica sono configurati per implementare Joliepse, un IDE per il linguaggio JOLIE.

5.1 Validazione

Nel capitolo precedente è citato JolieValidator, un validator configurabile generato dal fragment JavaValidatorFragment del workflow MWE2 di configurazione. Xtext offe tre tipi di validazione automatica:

- **validazione sintattica**: permette di visualizzare gli errori contenuti in ogni file sorgente durante la scrittura del codice. Questa funzione, forse la più utile dal punto di vista del programmatore, è implementata in maniera automatica sfruttando il parser e il lexer generati da ANTLR;

- **validazione delle referenze incrociate**: le referenze incrociate, descritte nel capitolo precedente, permettono di definire collegamenti all'interno di un file sorgente o tra più file. Il controllo della correttezza dei collegamenti è implementato automaticamente dall'editor tramite il linker, navigando fra i modelli EMF utilizzati per rappresentare gli AST dei file di input. Una referenza incrociata è memorizzata all'interno dell'AST come URI di un EObject all'interno dello stesso modello o fra i modelli appartenenti allo stesso ResourceSet;

- **validazione della sintassi concreta**: questo tipo di validazione avviene prima di ogni processo di serializzazione e dopo la modifica di un modello. Quando ad esempio si utilizza Xtext in combinazione con uno strumento di editing grafico di modelli o nel caso in cui un modello sia modificato manualmente come nell'implementazione di quick fix. In questa situazione, per controllare la conformità dello input alla grammatica non è utilizzato il parser ma un'implementazione dell'interfaccia IConcreteSyntaxValidator.

E' inoltre possibile configurare il processo di validazione tramite la classe opportuna generata dal worklow MWE2. Durante la fase di configurazione in realtà le classi generate relative alla validazione sono due: nel caso di JOLIE, abbiamo la classe astratta AbstractJolieJavaValidator che semplicemente mantiene una lista degli EPackage su cui dovrà operare, e JolieJavaValidator, che estende la precedente e contiene le definizioni dei vincoli che l'utente può imporre al proprio linguaggio.

*A*bstractJolieJavaValidator estende inoltre AbstractDeclarativeValidator, che permette di definire i vincoli in maniera dichiarativa utilizzando l'annotazione *@Check* sui vari metodi. I metodi definiscono come parametro formale un elemento della grammatica del linguaggio, e il validator invocherà i metodi opportuni a tempo d'esecuzione.

Xtext fornisce inoltre classi di test, implementate tramite JUnit, che permettono di verificare la correttezza del validator dichiarativo creato dall'utente, generando ad esempio modelli che appositamente violano i vincoli definiti nel validator.

In Joliepse, la classe JolieJavaValidator che funge da validator è configurata per verificare alcuni vincoli non definibili in maniera semplice tramite la grammatica xtext per JOLIE, come nel caso del costrutto *with*. Ecco un frammento del codice:

```
1  public class JolieJavaValidator extends AbstractJolieJavaValidator {
2
3    public static final String PREFIXED_WITH_BLOCK =
4            "jolie.xtext.jolie.PrefixedWithoutWithBlock";
5
6    @Check
7    public void checkPrefixedVariableinWith(VariablePath variablePath) {
8      boolean flag = false;
9
10     for (int i = 0; i < variablePath.getName().size(); i++) {
11       if (variablePath.getDot().size() == 1) {
12
```

```
13      EObject container = variablePath.eContainer();
14        while (!(container instanceof jolie.xtext.jolie.impl.ProgramImp
15        &&flag = false)) {
16
17           container = container.eContainer();
18           if (container instanceof jolie.xtext.jolie.impl.WithImpl)
19            flag = true;
20           if (container instanceof jolie.xtext.jolie.impl.ProConfImpl)
21            flag = true;
22
23        }
24
25        if (flag == false)
26          error("Prefixed␣variable␣paths␣must␣be␣inside␣a␣with␣block",
27              JoliePackage.VARIABLE_PATH__NAME,
28              PREFIXED_WITH_BLOCK, "name");
29   ...
30 }
```

Il costrutto with in JOLIE fornisce una scorciatoia per l'accesso a variabili contenute nella stessa struttura dati.

```
main{
      animals.pet[0].name = "cat";
        animals.pet[1].name = "dog";
        animals.wild[0].name = "tiger";
        animals.wild[1].name = "lion"
}
```

Il codice sopra può essere riscritto utilizzando with:

```
with (animals) {
        .pet[0].name = "cat";
        .pet[1].name = "dog";
        .wild[0].name = "tiger";
        .wild[1].name = "lion"
}
```

Il problema sorge dal fatto che all'interno di un blocco definito dal with è possibile inserire qualsiasi tipo di istruzione, con la possibilità di utilizzare il punto prefisso all'identificatore di una variabile.

Permettere questa istruzione al livello della grammatica non è semplice e occorrerebbe raddoppiare il numero di produzioni complessive.

Inserendo la possibilità del punto iniziale nella produzione riguardante il percorso della variabile invece, è possibile spostare questo tipo di controllo al livello del validator. A ogni occorrenza di una regola del tipo VariablePath all'interno del modello che rappresenta l'albero di sintassi astratta, il validator richiama il metodo *checkPrefixedVariableinWith()*, alla riga 7 del listato precedente. Se l'identificatore della variabile inizia con un punto, è possibile risalire l'AST sfruttando il modello EMF corrispondente. Ogni elemento della grammatica è, infatti, un EObject e come tale possiede il medoto eContainer() che restituisce l'EObject padre del costrutto corrente. Se, risalendo l'albero, il validator non incontra un costrutto di tipo with, è rilevato un errore inviando un messaggio all'utente tramite l'interfaccia grafica dell'editor.

È possibile allo stesso modo rilevare altri tipi di errore o generare warning, controllando ad esempio che i nomi delle interfacce inizino per lettera maiuscola o che i nomi delle variabili non siano più lunghi di un certo valore.

Oltre alla comunicazione dell'errore all'interfaccia grafica, il validator può opzionalmente comunicare con il gestore dei quick fix che si occuperà di proporre all'utente un modo per correggere l'errore.

5.2 *Scoping*

Nel contesto di Xtext, le API per lo scoping permettono di definire la visibilità delle referenze incrociate. Nel caso di Joliepse, le referenze incrociate hanno come unico scopo agevolare il lavoro del programmatore rendendo il codice navigabile. Nella grammatica sono presenti referenze incrociate che permettono di collegare:

- tipo di dato e sua definizione;
- chiamata di funzione e sua definizione (istruzione *define*);
- interfaccia utilizzata in una porta e sua definizione;
- invocazione di un'operazione su una porta e definizione nell'interfaccia.

L'interfaccia che si occupa di gestire lo scope di una referenza incrociata è IScopeProvider. Xtext permette di definire la politica generale di scoping del linguaggio tramite il workflow MWE2, permettendo in seguito di modificare l'implementazione dello scope provider per adattarlo alle proprie esigenze.

In JOLIE lo scope delle referenze incrociate descritte sopra e delle variabili è globale all'interno di tutto il file sorgente. Utilizzando la direttiva *include* all'inizio del file sono inoltre visibili tutti gli elementi del codice importato.

Xtext permette di definire questa politica tramite il meccanismo di import delle URI. Attivando nel workflow il fragment ImportURIScopingFragment, la classe JolieScopeProvider generata estenderà ImportUriGlobalScopeProvider, ed è sufficiente rispettare una convenzione rispetto al nome della feature della produzione interessata, che deve essere importUri, per attivare questo comportamento. Nella grammatica è quindi sufficiente inserire:

```
Include:
        'include' importURI=STRING;
```

per implementare in Joliepse la corretta politica di scoping del linguaggio JOLIE.

In questo modo, oltre ad esempio alla navigabilità delle referenze incrociate tra un file in cui è invocata un'operazione e il file contenente la definizione, sono disponibili suggerimenti per l'auto completamento (grazie al proposal provider, descritto nelle prossime sezioni) che includono gli elementi presenti nei file dichiarati con la direttiva *include*.

In JOLIE, oltre a poter includere in un file altri documenti prodotti all'interno di un progetto, sono disponibili servizi library, che al momento dell'installazione dell'interprete sono copiati all'interno della cartella */include* del percorso d'installazione del linguaggio.

Ad esempio, tramite l'istruzione

```
include "console.iol"
```

è possibile utilizzare le operazioni rese disponibili dal servizio di console, che fungono da interfaccia di input e output con l'utente.

Per implementare questa funzione è necessaria una modifica alla classe JolieScopeProvider:

```
1  public class JolieScopeProvider extends ImportUriGlobalScopeProvider {
2
3    private EList<Include> includeList;
4
5    @Override
6    public IScope getScope(EObject context, EReference reference) {
7
8      EObject container = context.eContainer();
9      while (!(container instanceof ProgramImpl)) {
10
11       container = container.eContainer();
12
13     }
14     ProgramImpl program = (ProgramImpl) container;
15     includeList = program.getInclude();
16     final LinkedHashSet<URI> uniqueImportURIs = getImportedUris(context);
17     IResourceDescriptions descriptions = getResourceDescriptions(context,
18         uniqueImportURIs);
19     ArrayList<URI> newArrayList = Lists.newArrayList(uniqueImportURIs);
20     Collections.reverse(newArrayList);
21     IScope scope = IScope.NULLSCOPE;
22     for (URI u : newArrayList) {
23
24       scope = createLazyResourceScope(scope, u, descriptions, reference);
25
26     }
```

```
27      return scope;
28    }
29
30    @Override
31    protected LinkedHashSet<URI> getImportedUris(EObject context) {
32
33      LinkedHashSet<URI> uris = super.getImportedUris(context);
34      for (Include include : includeList) {
35
36        uris.add(URI.createURI("platform:/resource/JolieIncludedLibraries/"
37            + include.getImportURI()));
38
39      }
40
41      uris.add(context.eResource().getURI());
42      return uris;
43    }
44  }
```

La scelta progettuale effettuata è quella di utilizzare all'interno di Joliepse un progetto che funge da library, contenente una copia dei servizi presenti nella cartella */include* del percorso di installazione di JOLIE. Il progetto è automaticamente creato dall'IDE e inserito nel workspace alla prima creazione di un servizio o di un'interfaccia JOLIE

La modifica apportata allo scope provider permette di recuperare l'URI dei servizi e delle interfacce contenuti nel progetto library, chiamato JolieIncludedLibraries, rendendo possibile includerli all'interno dei file creati con Joliepse.

L'aspetto positivo di questa soluzione è che il programmatore durante la scrittura del codice JOLIE del proprio progetto ha a disposizione anche le operation dei servizi base tramite il proposal provider.

E' da notare comunque come all'interno della grammatica alla feature importURI non sia assegnata una referenza incrociata bensì una semplice

stringa. Questo non rende quindi navigabile il nome del file incluso, e per implementare questa funzionalità Joliepse utilizza il componente del linguaggio chiamata hyperlink helper, descritto successivamente.

5.3 Serializzazione e formattazione

In Xtext la serializzazione permette di trasformare un modello EMF nella sua rappresentazione testuale. Possiamo schematizzare in cinque punti questo processo:

1. il validator di sintassi concreta controlla che il modello sia conforme alle regole della grammatica;
2. il *parse tree constructor* costruisce uno stream di token conforme al modello e alla grammatica, che comprende tutti gli elementi non necessari nel modello semantico come parole chiave e simboli sintattici;
3. il *comment associator* aggiunge i commenti al modello semantico;
4. ai nodi del modello semantico sono aggiunte le stringhe corrispondenti nello stream di token;
5. sono aggiunti gli spazi e le linee bianche utilizzando un formatter.

La serializzazione è utilizzata in xtext quando i modelli sono modificati da altri strumenti come ad esempio un tool grafico o durante l'utilizzo di un quick fix, quando cioè il modello semantico è modificato forzatamente e non è possibile utilizzare il parser.

Durante la fase di serializzazione è utilizzato un componente, il formatter, che hai il compito di inserire spazi e linee vuote tra i token per ricomporre il codice corrispondente ad un modello.

In Xtext il formatter può essere modificato a piacimento per rispettare le regole stilistiche di scrittura del codice, e può essere richiamato all'interno dell'IDE in qualsiasi momento tramite la sequenza di tasti opportuna o una voce di menu.

Xtext permette di utilizzare due tipi di formattatori, OneWhitespaceFormatter, che semplicemente inserisce uno spazio bianco tra ogni token, e AbstractDeclarativeFormatter che permette di impostare le regole di formattazione del codice del proprio linguaggio in maniera dichiarativa.

Joliepse utilizza il secondo approccio tramite JolieFormatter, generato dal workflow MWE2 dal fragment FormatterFragment. La configurazione del formatter è veramente molto semplice, come si può notare dal seguente frammento di codice:

```
1  public class JolieFormatter extends AbstractDeclarativeFormatter {
2
3    @Override
4    protected void configureFormatting(FormattingConfig c) {
5      JolieGrammarAccess f = (JolieGrammarAccess) getGrammarAccess();
6
7      c.setLinewrap(1,2,3).around(f.getBasicStatementRule());
8
9      List<Pair<Keyword,Keyword>> pairs = f.findKeywordPairs("{", "}");
10     for (Pair<Keyword, Keyword> pair : pairs) {
11       c.setIndentation(pair.getFirst(), pair.getSecond());
12     }
13     ...
14 }
```

Le regole sono richiamate tramite l'elemento GrammarAccess ed è possibile con metodi come *setLineWrap()* e *setIndentation()* formattare il codice ponendo un'istruzione su ogni riga e indentare il codice all'interno di parentesi graffe.

5.4 *Label Provider*

All'interno dell'IDE gli elementi contenuti in un modello sono presentati graficamente all'utente nell'outline e nei suggerimenti di completamento

automatico del proposal provider.

L'interfaccia ILabelProvider permette di definire quali immagini e stringhe utilizzare rispettivamente come icone ed etichette per gli elementi della grammatica.

L'implementazione base di questa interfaccia è data da DefaultLabelProvider, e LabelProviderFragment nel workflow MWE2 permette di generare JolieLabelProvider che estende questa classe.

Anche in questo caso la configurazione è molto semplice: è necessario solamente definire un metodo *image()* e un metodo *text()* con il parametro formale corrispondente alla regola della grammatica per la quale si vuole fornire un icona o un etichetta.

DefaultLabelProvider implementa un visitor che andrà ad associare icone ed etichette agli elementi che compongono il file sorgente ad ogni momento necessario. Nel caso di JOLIE:

```
1  public class JolieLabelProvider extends DefaultEObjectLabelProvider {
2    ...
3    @Inject
4    public JolieLabelProvider(AdapterFactoryLabelProvider delegate) {
5      super(delegate);
6    }
7
8    @Override
9    protected Object doGetImage(Object element) {
10     if(element instanceof EObject) {
11       return ((EObject)element).eClass().getName() + ".gif";
12     }
13     return super.doGetImage(element);
14   }
15   ...
16 }
```

E' sufficiente creare un'immagine con il nome corrispondente e inserirla nella cartella icons del plug-in riguardante l'interfaccia grafica e il label

provider si occuperà di eseguire le associazioni a tempo d'esecuzione.

5.5 *Proposal Provider*

Xtext fornisce il supporto all'IDE per una funzionalità molto interessate, il cosiddetto proposal provider, che elabora suggerimenti sensibili al contesto durante la scrittura del codice. Questa feature è implementata sfruttando la natura EMF dei modelli e il componente GrammarAcces: nel caso di JOLIE ad esempio, volendo definire un nuovo tipo di dato strutturato, è possibile richiamare il proposal provider con la sequenza di tasti opportuna subito dopo la digitazione dei due punti, come è possibile osservare in figura 5.1.

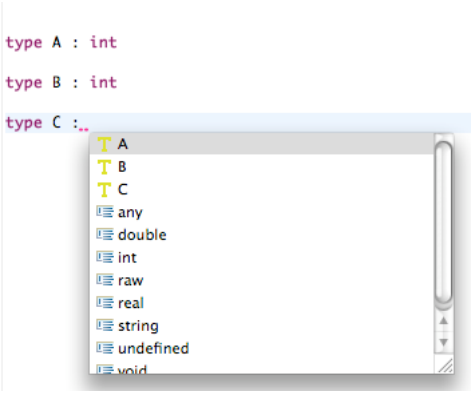

Figura 5.1: Esempio dell'utilizzo del proposal provider

Il proposal provider tramite GrammarAccess recupera le scelte disponibili direttamente dalla produzione denominata Type (già discussa nel capitolo precedente nella sezione riguardante le referenze incrociate), che comprendono i tipi nativi come int, real e string; poiché fra le possibili scelte di Type nella grammatica è presente la referenza incrociata verso un Type, il proposal provider cerca all'interno del modello corrente e fra tutti i corrispondenti ai file collegati tramite la direttiva *include* tutte le definizioni di tipo, che vengono inserite tra le proposte disponibili.

Anche questo componente è altamente configurabile, ed in Joliepse è utilizzata per presentare all'utente le proposte relative alla direttiva *include*,

che come nel caso dello scope provider non è implementata automaticamente durante la generazione del linguaggio.

Come spiegato nella sezione sulle politiche di scope, in Joliepse esiste un progetto library, denominato JolieIncludedLibraries, che è generato alla creazione del primo progetto JOLIE, e ha lo scopo di presentare il codice dei servizi base che risiedono nella cartella *include* di JOLIE, come la console per la stampa a video.

La modifica al proposal provider riguarda sempre la direttiva *include*, che non essendo implementata tramite referenza incrociata non è gestita automaticamente.

Il proposal provider, quando è richiamato dopo un'istruzione *include*, recupera i nomi dei servizi e delle interfacce presenti nel progetto JolieIncluded-Libraries e nel progetto corrente tramite un oggetto IResourceDescriptions, che restituisce i riferimenti a tutti i file aperti nel workspace:

```
1  private IResourceDescriptions descriptions;
2
3  @Override
4  public void completeInclude_ImportURI(EObject model,
5  Assignment assignment, ContentAssistContext context,
6     ICompletionProposalAcceptor acceptor) {
7
8        Iterable<IResourceDescription> allResourceDescriptions =
9          descriptions.getAllResourceDescriptions();
10         for (IResourceDescription description :
11               allResourceDescriptions) {
12
13                     if (description.getURI().toString()
14         .contains("JolieIncludedLibraries")) {
15
16       proposal = '"' + description.getURI().lastSegment() ;
17       completionProposal =
18         createCompletionProposal(proposal, context);
```

```
19          acceptor.accept(completionProposal);
20
21        }
22
23      ...
```

Ogni nome di file è infine aggiunto alla lista delle proposte tramite il metodo *accept()* del proposal provider.

5.6 *Quick Fix*

JolieQuickfixProvider, generato da QuickfixProviderFragment, fornisce una funzionalità molto pratica all'IDE target: le correzioni veloci o comunemente quick fix.

Xtext fornisce quick fix automatici quando possibile, ad esempio in Joliepse nel caso di piccoli errori sintattici relativi agli identificatori di tipi o operation l'IDE fornisce le proposte corrette in base al contesto. E' possibile osservare questo comportamento in figura 5.2.

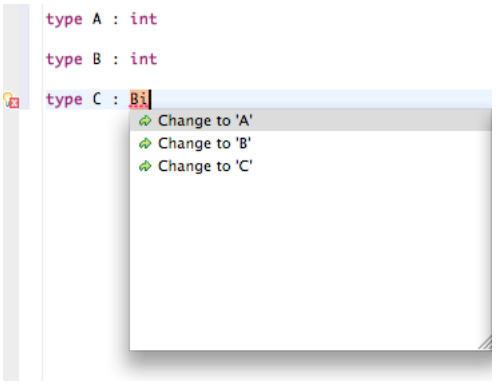

Figura 5.2: Un esempio di quick fix generato automaticamente

Per i warning o gli errori rilevati dal validator esiste inoltre la possibilità di fornire all'utente proposte ad hoc per risolvere immediatamente il problema.

Nella sezione sul validator si afferma che questo è in grado di comunicare con il quick fix provider per attivare i suggerimenti di correzione per un

particolare errore. Questo avviene tramite un'opportuna stringa definita nel validator, passata al metodo *error()* o *warning()*, che si occupa di segnalare la presenza di un problema all'interfaccia grafica dell'editor.

Quando ad esempio il validator riscontra all'interno dell'AST l'utilizzo di una variabile con un punto prefisso non contenuta in un blocco with, comunica l'errore all'editor passando la costante PREFIXED_WITHOUT_WITH_BLOCK al metodo error(). Il quick fix provider, tramite l'annotazione *@Fix*, associa la proposta di correzione alla segnalazione opportuna come nell'esempio:

```
1  @Fix(JolieJavaValidator.PREFIXED_WITHOUT_WITH_BLOCK)
2    public void fixFeatureName(final Issue issue,
3        IssueResolutionAcceptor acceptor) {
4      acceptor.accept(issue, "Delete␣the␣dot",
5          "Delete␣the␣dot␣'" + issue.getData()[0] + "'", ,
6          new IModification() {
7            public void apply(IModificationContext context) {
8
9              IXtextDocument xtextDocument = context
10                 .getXtextDocument();
11             try {
12               xtextDocument.replace(issue.getOffset() - 1, 1, "");
13             } catch (BadLocationException e) {
14               e.printStackTrace();
15             }
16           }
17         });
18 ...
```

Xtext permette di correggere l'errore con due possibilità:

- una modifica puramente testuale, basata sulle interfacce IDocument di Eclipse e IModification di Xtext, che permettono di recuperare e modificare le informazioni contenute nel documento con dati come numero di linea e di carattere;

- una modifica basata su modello semantico, tramite l'interfaccia ISemanticModificcation, che permetta di agire direttamente sugli EObject che compongono il modello EMF che rappresenta l'AST. Questa soluzione utilizza il validator di sintassi concreta per verificare che le modifiche apportate al modello siano conformi alla grammatica del linguaggio.

Nell'esempio sopra è utilizzata una modifica testuale del documento, in cui si cancella il punto iniziale di una variabile non contenuta in un costrutto with sfruttando offset della posizione in cui è riscontrato l'errore.

5.7 *Code Template*

Eclipse fornisce il supporto per la creazione di qualsiasi tipo di template di codice, che può essere richiamato con la stessa sequenza di tasti del proposal provider, permettendo di diminuire il tempo di scrittura di istruzioni sintatticamente complesse. I template sono definiti in un documento XML, e tramite il pannello preferenze del proprio IDE l'utente può importare template e crearne di nuovi.

Xtext permette di definire una serie di template che saranno automaticamente importati nell'IDE target, semplicemente inserendo il documento XML corrispondente nella cartella templates del progetto sorgente. La sintassi dei template è molto semplice e immediata, ad esempio il seguente frammento di codice in Joliepse:

```
1 <template autoinsert="true"
2 context="jolie.xtext.Jolie.InputPortStatement"
3 deleted="false" description="template␣for␣an␣InputPortStatement"
4 enabled="true" id="jolie.xtext.jolie.InputPortStatement"
5 name="InputPortStatement">inputPort ${name} {
6
7     ${Location} : ${Uri}
8     ${Protocol} : ${Name}
```

```
9
10 }</template>
```

permette di generare il codice corrispondente alla definizione di una input port di JOLIE e di modificarne successivamente il contenuto.

5.8 *Outline*

Xtext fornisce una visualizzazione grafica della struttura del codice, chiamata comunemente outline, basta sulla gerarchia della grammatica del linguaggio. E' possibile modificare l'outline tramite l'interfaccia ISemanticModelTransformer. Un'implementazione di questa interfaccia è fornita dalla classe AbstractDeclarativeSemanticModelTransformer, che permette di influenzare l'outline in maniera dichiarativa, creando novi nodi dell'albero che rappresenta il codice o modificando quelli attuali.

In Joliepse l'outline non è modificato, poichè il comportamento di default è giudicato sufficiente.

5.9 Collegamenti

Xtext implementa automaticamente il supporto per i collegamenti all'interno del documento corrente e fra tutti quelli inclusi utilizzando le referenze incrociate definite nella grammatica tramite il componente *hyperlink helper*.

Per raggiungere la corretta posizione all'interno del documento indicata dal collegamento, l'hyperlink helper utilizza l'interfaccia ILocationInFileProvider, che sfrutta la feature *name* dalla produzione a cui punta la referenza incrociata.

Così come per gli altri elementi dell'IDE, è possibile modificare il comportamento standard dell'hyperlink helper, e in Joliepse questo avviene ancora una volta per il comando *include*, permettendo di raggiungere il codice del file incluso.

Per ogni istruzione *include*, si ricerca all'interno del progetto corrente e del progetto library l'URI che rappresenta l'indirizzo simbolico del file all'interno della piattaforma Eclipse, e questo indirizzo è impostato come collegamento per il nome del file incluso, attivabile con il tasto corrispondente.

Riassunto capitolo

In questo capitolo sono indicate tutte le modifiche apportate agli elementi base di un IDE generato con Xtext per implementare Joliepse.

E' analizzato il processo di validazione, gestito automaticamente da Xtext in tre differenti livelli, e come Joliepse utilizzi un validator dichiarativo per gestire il costrutto with.

Per quanto riguarda le politiche di scope, Joliepse utilizza scope globale basato sulla direttiva *include*, e tramite una modifica allo scope provider sono inoltre visibili i servizi base del linguaggio Jolie tramite la scelta progettuale del progetto library.

L'implementazione dell'istruzione *include* richiede inoltre una modifica del comportamento di default del proposal provider e della gestione degli hyperlink fra i file.

Sono inoltre descritti i funzionamenti del processo di serializzazione, formattazione del codice, gestione del'outline, quick fix e template di codice.

Il prossimo capitolo descrive come sia possibile utilizzare Xtext in modalità stand-alone, ad esempio sostituendo il parser attuale di Jolie con il parser di Xtext generato da ANTLR.

Capitolo 6

Sotituzione del parser di JOLIE

Nei capitoli precedenti è descritto come Xtext permetta sia di generare un IDE Eclipse-based per il linguaggio target, sia di utilizzare soltanto alcuni componenti all'interno di qualsiasi applicazione Java, in modalità stand-alone.

Questo capitolo espone una modifica dell'interprete del linguaggio JOLIE, in cui il parser attuale, scritto senza l'ausilio di generatori, è sostituito dal parser generato da Xtext tramite ANTLR.

Nella prima sezione è presentata una breve panoramica sulle tecniche di parsing disponibili e nella seconda la struttura del parser attuale di JOLIE. La terza sezione descrive ANTLR, mentre l'ultima parte espone la sostituizione del parser attuale di JOLIE con quello utilizzato da Xtext e i problemi riscontrati.

6.1 Parsing

La fase di parsing o analisi sintattica consiste, formalmente, nel verificare che una qualunque sequenza di caratteri appartenga all'insieme di tutte le stringhe di un particolare linguaggio di programmazione, ed in caso affermativo produrre l'albero di sintassi astratta corrispondente.

L'insieme di tutte le stringhe di un linguaggio è definito da una grammatica libera dal contesto, solitamente scritta in Backus-Naur Form o in Extended Backus-Naur Form come nel caso di Xtext.

Nel processo di compilazione e interpretazione la fase di parsing è preceduta dall'analisi lessicale, che ha lo scopo di raggruppare la sequenza di caratteri dello stream di input in token utilizzando espressioni regolari.

Un parser quindi può essere definito come una macchina astratta che raggruppa i token contenuti nell'input in base alle produzioni definite nella grammatica del linguaggio.

La grammatica libera dal contesto ha lo scopo di definire in modo semplice e non ambiguo la sintassi di un linguaggio di programmazione, permettendo l'implementazione di algoritmi di complessità lineare per l'analisi dell'input.

L'output prodotto dall'analisi lessicale può essere quindi o una lista di errori contenuti nell'input o l'albero di sintassi astratta utilizzato successivamente per l'analisi semantica, che porta all'interpretazione e quindi all'esecuzione del programma o alla traduzione verso un altro linguaggio.

Un aspetto cruciale dell'analisi sintattica è l'algoritmo utilizzato per il riconoscimento dell'input. Possiamo suddividere questi algoritmi in due categorie principali [GruJac90]:

- **algoritmi top-down:** partendo dal simbolo iniziale della grammatica si costruisce l'albero sintattico che ha come foglie la sequenza di token dell'input. I parser a discesa ricorsiva e i parser LL utilizzano quest'approccio;

- **algoritmi bottom-up:** l'albero sintattico è costruito partendo dalle foglie, risalendo fino a raggiungere il simbolo iniziale, come nel caso di parser LR.

6.1.1 Parser top-down

Parser a discesa ricorsiva

Un parser a discesa ricorsiva è solitamente la scelta effettuata in caso d'implementazione manuale dell'analizzatore, come nel caso di JOLIE.

Ad ogni simbolo non terminale della grammatica è associata una procedura, e la prima eseguita è quella corrispondente al simbolo iniziale della grammatica. Normalmente ogni procedura contiene un ciclo in grado di analizzare quali delle possibili produzioni corrisponda all'input, utilizzando quindi *backtracking* per risalire l'albero sintattico che si sta costruendo.

Per implementare un parser a discesa ricorsiva la grammatica non deve presentare ricorsione sinistra, non deve cioè contenere produzioni in cui il primo termine della derivazione è lo stesso simbolo iniziale della produzione. La ricorsione sinistra potrebbe infatti causare cicli infiniti all'interno del parser.

Parser predittivi e LL(k)

Una sottoclasse dei parser a discesa ricorsiva è rappresentata dai cosiddetti parser predittivi, che permettono di rendere più efficiente l'analisi lessicale non utilizzando backtracking.

La diminuzione dell'efficienza di un parser a discesa ricorsiva è data proprio dal dover eseguire numerosi backtracking per riuscire a riconoscere la giusta produzione per un particolare input. Per risolvere questo problema, i parser predittivi utilizzano il concetto di *lookahead*, che consiste nell'analizzare un numero arbitrario di token che seguono quello corrente per decidere che produzione selezionare. Per poter utilizzare un solo simbolo di lookahead senza backtracking occorre che la grammatica, oltre a non presentare ricorsione sinistra, sia caratterizzata da fattorizzazione sinistra, che permette di eliminare un eventuale prefisso comune a due parti destre della stessa produzione. Una grammatica così ottenuta è detta LL(1). Riassumendo, una grammatica LL(1):

- utilizza un solo simbolo di lookahead;
- analizza l'input da sinistra verso destra;
- risolve le derivazioni *left most*.

I parser associati a questo tipo di grammatica cono detti parser LL(1). Un parser è invece LL(k) se utilizza k simboli di lookahead per associare ad una sequenza di token la giusta produzione. Esistono inoltre parser indicati come LL(*) come ANTLR che possono utilizzare un numero di simboli di lookahead variabile.

L'implementazione classica di un parser LL(1) avviene mediante una tabella di parsing, che permette di indicare che produzione utilizzare per costruire l'albero in base all'input e una pila di supporto. Per costruire la tabella di parsing vengono utilizzati gli insiemi *first* e *follow* definiti nel seguente modo:

- **first(α)**: è l'insieme dei simboli terminali che occupano la prima posizione delle stringhe ottenute da una qualche derivazione di α, dove α è una stringa composta da simboli terminali e non terminali;
- **follow(A)**: è l'insieme dei terminali che seguono il non terminale A nelle stringhe derviate da tutte le produzioni della grammatica.

L'algoritmo classico per costruire insieme first, follow e tabella di parsing a partire da una grammatica BNF che non presenti ricorsione sinistra e fattorizzazione sinistra è relativamente semplice e non verrà indicato, perchè non interessante ai fini di questo capitolo ed in particolar modo di questa tesi.

In termini pratici infatti il compito della costruzione di questi elementi è lasciato a generatori di parser LL(k) come ANTLR [ParQuo95], discusso successivamente, o JavaCC [Coo07].

6.1.2 Parsing bottom-up

Gli algoritmi di parsing bottom-up costruiscono l'albero sintattico dalle foglie fino ad arrivare alla radice, rappresentata dal simbolo iniziale della grammatica. In linea di principio, si cerca di individuare all'interno dello input le sequenze che rappresentano la parte destra di una produzione, sostituendo poi la sequenza con il non terminale corrispondente fino a raggiungere il simbolo iniziale.

La più comune strategia di parsing bottom-up è lo *shift-reuce parsing*, che utilizza un automa a pila implementato mediante due tabelle, una *action table* e una *goto table*. Ogni stato dell'automa è rappresentato da una riga di questa tabella.

Ad ogni passo il parser, in base all'input e allo stato dell'automa controlla nella action table il tipo di operazione da eseguire, che può essere *shift* o *reduce*.

Se nella tabella è indicata un'operazione shift associata a un particolare stato, questo è inserito nella pila e diventa lo stato corrente mentre il simbolo è rimosso dall'input.

Se invece è indicata un'operazione reduce associata a un numero che corrisponde a una produzione, la regola è inserita nell'output e sarà utilizzata per costruire l'abero, mentre dalla pila sono rimossi un numero di stati pari al numero di simboli contenuti nella parte destra della produzione corrispondente; in base al simbolo non terminale a sinistra della produzione inserita nell'output e allo stato corrente si consulta la goto table che identifica il passo successivo.

La tabella inoltre contiene un'azione accept che corrisponde all'accettazione della stringa e azioni error che permettono di identificare gli errori.

Poichè l'analisi dell'input avviene tramite un automa a pila, questo tipo di parsing produce derivazioni *right most*, e viene indicato con la sigla LR.

Anche in questo caso è possibile utilizzare simboli di lookahead. In base al numero di simboli di lookahead e all'algoritmo di costruzione della tabella

di parsing, è possibile suddividere parser e grammatiche bottom up in LR(0), LR(1), LR(n), SLR(1) e LALR(1).

A differenza del parsing LL(k), in questo caso la grammatica libera dal contesto non deve avere particolari caratteristiche, ed in particolar modo può presentare ricorsione e fattorizzazione sinistra. E' possibile quindi implementare un parser LR per qualsiasi liguaggio di cui sia disponibile una grammatica BNF. La classe di linguaggi definibili con grammatiche LR è quindi maggiore della classe LL.

Implementare manualmente un parser LR, specialmente nel caso dell'utilizzo di simboli di loockhead, è comunque molto complicato per via del grande numero di stati che possono essere richiesti dall'automa. Sono quindi necessari generatori di parser come Yacc [Joh78] per il linguaggio C o LPG [Lpg10] per Java.

6.2 Il parser di Xtext generato da ANTLR

Come abbiamo visto nei capitoli precedenti, Xtext utilizza ANTLR come generatore di parser. Durante la fase di generazione del linguaggio la grammatica di Xtext è infatti tradotta in una grammatica per ANTLR e questo viene eseguito per produrre il lexer e il parser utilizzati all'interno del framework. Vediamo quindi una breve descrizione di ANTLR confrontato con le tecniche di parsing viste nella prima sezione di questo capitolo.

6.2.1 ANTLR

ANTLR [ParQuo95] è un generatore di parser per grammatiche LL(k), che produce parser predittivi a discesa ricorsiva strutturalmente simili ai parser scritti a mano. Inizialmente implementato per i linguaggi C e C++, è disponibile anche nella sua versione Java.

ANTLR nasce dall'esigenza di un parser generator di facile comprensione sia dal punto di vista della sintassi utilizzata per la definizione della grammatica e la fase di configurazione, sia dal punto di vista della struttura dell'out-

put, cioè il lexer e il parser per un determinato linguaggio. Sia parser LL(k) implementati tramite tabelle che parser LR infatti spesso risultano difficili da comprendere da parte del programmatore, complicando la fase di debug. Un'altra importante considerazione è che ANTLR, a differenza di altri tool simili, non si concentra soltanto sulla tecnica di analisi lessicale utilizzata, ma permette di integrare e considerare altri aspetti che comprendono analisi sintattica, gestione degli errori e costruzione dell'albero di sintassi astratta.

Possiamo indicare i principali punti di forza di ANTLR nel seguente elenco:

- ANLTR integra la definizione di lessico e grammatica di un linguaggio in un unico documento. Altri parser generator richiedono che lo stram di input sia precedentemente suddiviso in token da uno scanner. ANTLR invece produce il lexer e il parser direttamente dalle regole e le produzioni della grammatica;

- ANTLR utilizza grammatiche in EBNF, che permettono l'utilizzo di operatori di cardinalità nella definizione di regole e produzioni; la grammatica prodotta risulta quindi di dimensioni più ridotte della corrispondente in BNF;

- ANTLR permette di influenzare la costruzione dell'AST a livello della grammatica tramite le azioni e valori di restituzione delle produzioni, e Xtext sfrutta questa caratteristica per lasciare al programmatore la libertà di definire quali EObject comporranno il modello EMF che rappresenta l'albero di sintassi astratta;

- Il parser generato è un parser a discesa ricorsiva, che utilizza una procedura per ogni produzione o regola della grammatica. Questa caratteristica rende il codice prodotto maggiormente comprensibile rispetto a un parser *table driven*;

- Antlr permette una gestione degli errori sia automatica che manuale. La grande potenza e libertà da questo punto di vista è il motivo principale per cui questo parser generator è utilizzato in Xtext;

Xtext utilizza ANTLR internamente, e la grammatica xtext è sintatticamente diversa da quella per ANTLR. Durante la fase di generazione del linguaggio, la grammatica per Xtext è tradotta nella corrispondente per ANTLR in cui sono aggiunte tutte le informazioni per costruire l'albero di sintassi astratta con EMF.

Ad ogni produzione viene assognato come tipo restituito un EObject definto nel componente GrammarAccess. L'overhead di informazioni che Xtext aggiunge alla grammatica per ANTLR la rende circa dieci volte più grande di una grammatica base per un linguaggio di programmazione come Java.

6.3 Il parser attuale di JOLIE

Il parser attuale di JOLIE è stato scritto senza l'utilizzo di generatori da Fabrizio Montesi [MGLZ06].

Così come la maggior parte dei parser scritti a mano, il parser di JOLIE è un parser a discesa ricorsiva che utilizza backtracking, ed è presente quindi una funzione per ogni produzione della grammatica. L'albero di sintassi astratta è rappresentato tramite composizione: esiste quindi un oggetto principale, *Program*, che contiene referenze a nodi sintattici che possono essere nodi composti (che definiscono ad esempio le istruzioni *parallel* e *sequence*) e nodi semplici o *basic statement*. Gli elementi sono quindi simili a quelli utilizzati da Xtext e EMF in Joliepse.

All'interno dell interprete di JOLIE, l'algoritmo di parsing descritto precedentemente è implementato dal seguente codice:

```
1  throws InterpreterException {
2        ...
3  OLParser olParser = new OLParser(new Scanner(
4    cmdParser.programStream(),
```

```
5       cmdParser.programFilepath()),
6        includePaths, classLoader);
7   olParser.putConstants(cmdParser.definedConstants());
8   program = olParser.parse();
9   OLParseTreeOptimizer optimizer = new OLParseTreeOptimizer(program);
10  program = optimizer.optimize();
11        ....
12  SemanticVerifier semanticVerifier = new SemanticVerifier(program);
13  if (!semanticVerifier.validate()) {
14                throw new InterpreterException("Exiting");
15            }
16  return (new OOITBuilder(this, program,
17  semanticVerifier.isConstantMap())).build();
18   ...
19  }
```

Possiamo schematizzare l'algoritmo nel modo seguente:

1. E' istanziato un oggetto della classe *OLParser*, che rappresenta il parser a discesa ricorsiva, passando al metodo costruttore il percorso assoluto del file da analizzare e un'istanza della classe *Scanner*. L'oggetto Scanner è utilizzato come analizzatore lessicale e permette di ottenere la rappresentazione in token del file di input (riga 3);

2. Al parser vengono aggiunte le costanti, che in JOLIE possono essere definite da linea di comando (riga 7);

3. E' invocato il metodo *parse()* del parser. Il metodo ha come tipo restituito un oggetto Program, che rappresenta la radice dell'AST (riga 8);

4. Il metodo richiama a sua volta sequenzialmente altri metodi che si occupano di analizzare i costrutti più esterni di JOLIE, che riguardano aspetti di configurazione come le porte, le istruzioni include e le defini-

zioni di tipo. Gli oggetti così creati sono aggiunti al Program, che ha visibilità globale;

5. Il parsing delle istruzioni contenute nel main avviene richiamando il metodo *parseCode()*; all'interno del metodo *parseCode()* è istanziato un oggetto di tipo Main che sarà successivamente aggiunto a Program. Sono richiamati, uno all'interno dell'altro e fino all consumo di tutti i token dello Scanner, *parseProcess()*, *parseParallelStatement()*, *parseSequenceStatement()* e *parseBasicStatement()*. All'interno di quest'ultimo, tramite una serie di *if* annidati sono aggiunti all'albero di sintassi astratta tutti i costrutti semplici come assegnamenti o invocazione di servizi, terminando l'operazione in caso di errori;

6. Quando la sequenza di token è terminata, l'oggetto Program che rappresenta l'AST è ottimizzato tramite *OLParseTreeOptimizer*, che esamina la struttura implementando il pattern visitor (riga 10);

7. Program viene poi analizzato allo stesso modo da un oggetto *SemanticVerifier*, che esegue una validazione semantica comprendente tutti i controlli non effettuabili a livello sintattico (riga 12);

8. E' istanziato un oggetto *OOITBuilder*, che ha il compito di trasformare l'albero di sintassi astratta in un *Object Oriented Interpretation Tree*, pronto per essere interpretato (riga 16).

6.4 La sostituzione del parser di JOLIE

E' già stato più volte spiegato come Xtext possa essere utilizzato sia all'interno della piattaforma Eclipse che in modalità stand-alone.

L'argomento principale di questa tesi è l'implementazione di un IDE per il linguaggi JOLIE, e Joliepse è il risultato ottenuto.

Un ulteriore area di analisi è rappresentata dalla possibilità di sostituire il parser attuale di JOLIE con un parser generato da un tool automatico a

partire da una grammatica libera dal contesto. Un risultato positivo permetterebbe infatti di rendere il linguaggio maggiormente estendibile e modulare, per esempio nel caso di voler aggiungere una nuova istruzione sarebbe sufficiente modificare pochi caratteri della grammatica, eseguire il generatore e dedicarsi soltanto alla modifica dell'interprete.

La struttura modulare di Xtext lo rende particolarmente indicato a questa soluzione, è infatti relativamente semplice integrare il framework all'interno di una qualsiasi applicazione Java come ad esempio l'interprete di JOLIE.

Dopo aver inserito i JAR necessari a Xtext alle library di JOLIE, è possibile copiare la directory */SRC-GEN*, che contiene tutte le classi generate dal workflow MWE2 di Xtext, all'interno della directory */Libjolie* di JOLIE.

JOLIE in fase di installazione utilizza il tool Apache ANT [BBB03] per compilare i file sorgente e produrre i class utilizzati a tempo d'esecuzione. E' possibile quindi impostare nel file di configurazione di ANT l'esecuzione del workflow MWE2 per generare le classi utilizzate da Xtext durante l'installazione del linguaggio.

A questo punto l'interprete di JOLIE può essere modificato nel modo seguente:

```
1  ...
2  Injector injector = new JolieStandaloneSetup().
3    createInjectorAndDoEMFRegistration();
4  XtextResourceSet resourceSet = injector.
5    getInstance(XtextResourceSet.class);
6  resourceSet.addLoadOption(XtextResource.OPTION_RESOLVE_ALL,
7    Boolean.TRUE);
8  Resource resource = resourceSet.createResource(
9    URI.createURI("dummy:/example.ol"));
10 resource.load(cmdParser.programStream(),
11   resourceSet.getLoadOptions());
12 EList<Resource.Diagnostic> errors = resource.getErrors();
13 ...
14 EObject model = resource.getContents().get(0);
```

```
15   ...
16 }
```

L'interfaccia *XtextResourceSet* permette di incapsulare il parser e il lexer di un linguaggio realizzato con Xtext. Alla riga 4 del listato, è creata un istanza di *JolieStandaloneSetup* che fornisce l'accesso a tutti i componenti del linguaggio tramite un injector quando non si utilizza la piattaforma Eclipse.

Dallo XtextResourceSet, che rappresenta il container di tutte le risorse, viene creata una risorsa fittizia. Come già descritto nel quarto capitolo, una risorsa è, nel contesto di EMF, una locazione fisica in cui risiede un modello EMF. Una resource in Xtext è quindi la locazione in cui risiede un file per il linguaggio target o la sua versione serializzata XMI. Invocando il metodo load di una risorsa (riga 11) con uno stream di input come parametro attuale, Xtext richiama internamente il lexer e il parser che, tramite il componente GrammarAccess si occupano di costruire il modello EMF che rappresenta l'AST.

Gli errori sintattici, se presenti, sono memorizzati all'interno di una Elist (riga 13), mentre lo EObject ottenuto alla riga 14 rappresenta il modello del file sorgente, il suo albero di sintassi astratta.

6.5 Problemi riscontrati

Abbiamo visto come sia possibile utilizzare Xtext all'interno di JOLIE in maniera relativamente semplice. E' comunque presente un importante trad-off tra versatilità e performance del framework: con la modifica apportata all'interprete di JOLIE descritta nella sezione precedente, si riscontra un aumento del tempo di parsing di un file decisamente non tollerabile.

Utilizzando lo strumento di profiling del codice integrato nell'IDE Net Beans è possibile confrontare il tempo di esecuzione dell'interprete nelle due situazioni, e con l'utilizzo di Xtext il tempo complessivo è circa dieci volte maggiore.

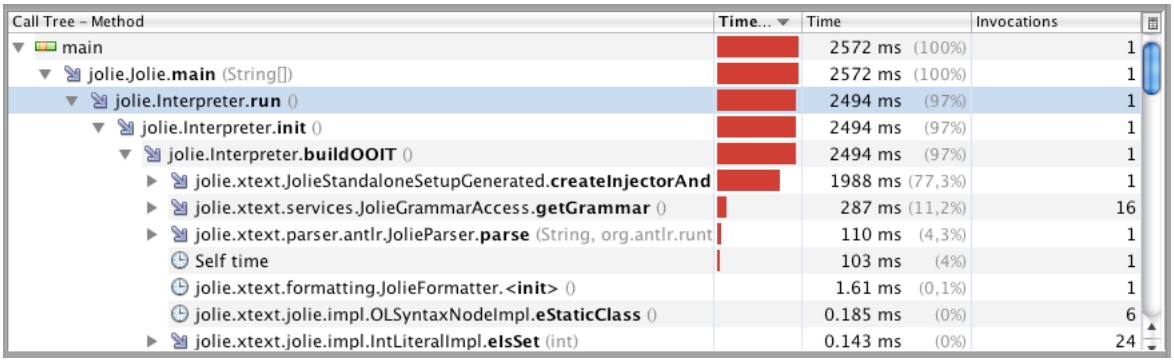

Figura 6.1: *Profile* dell'interprete con Xtext

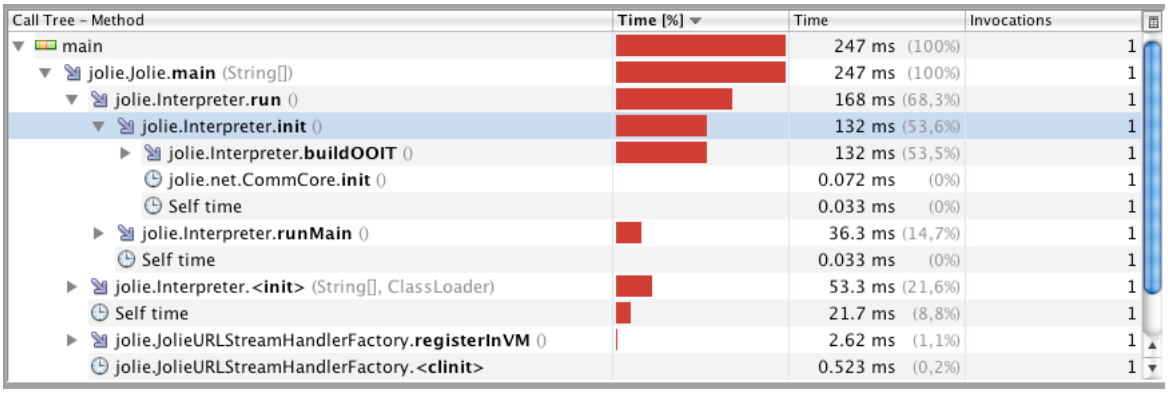

Figura 6.2: *Profile* dell'interprete senza Xtext

Analizzando nel dettaglio la fig. 6.1 si può notare come in realtà questo aumento sia da attribuire alla fase di inizializzazione del framework, in cui avvengono i binding fra classi concrete e identificatori di Google Guice.

Come descritto nella terza sezione quarto capitolo infatti Xtext utilizza il framework Google Guice per collegare a tempo d'esecuzioni tutte i componenti del linguaggio, e in questo caso per coordinare il parser interno generato con ANTLR e il componente GrammarAccess, che contiene le informazioni relative agli EObject che formeranno lo AST.

Una volta effettuato questo bootstrap, il tempo dedicato al parsing resta comunque maggiore rispetto al parser attuale ma comunqe tollerabile, introno ai 200 ms per un file sorgente di dimensioni medie.

Le possibilità per porre rimedio a questo inconveniente sono due:

- **Modificare il framework XText**: eliminando dal workflow MWE2 di generazione tutte i componenti non necessari si può ottenere una diminuzione del tempo di esecuzione, ma comunque non sufficiente. Sarebbe invece tecnicamente possibile eliminare l'utilizzo di Google Guice, configurando preventivamente GrammarAccess in modo da non dover più ricorrere alla fase di bootstrap iniziale;

- **Traduzione della grammatica**: un'altra possibilità è rappresentata dalla traduzione della grammatica di Xtext per JOLIE in una equivalente per ANTLR. Questo processo non è particolarmente costoso data la similitudine dei due formalismi.

Riassunto capitolo

In questo capitolo è discusso come sia possibile modificare la struttura dell'interprete attuale di JOLIE, sostituendo l'attuale parser scritto a mano con uno equivalente prodotto da un parser generator. Dopo una breve introduzione sulle tecniche di parsing esistenti, è introdotto il parser generator ANTLR e come questo tool è utilizzato nel contesto di Xtext. Sostituento direttamente il parser attuale di JOLIE con quello utilizzato da Xtext si

ottiene un notevole aumento del tempo parsing, dovuto principalemente a Google Guice, il framework per dependency injection utilizzato da Xtext, e nell'ultima sezione sono indicate due possibilità per eliminare questo problema. Nel prossimo capitolo sono indicate le conclusioni e gli sviluppi futuri di questa tesi.

Capitolo 7

Conclusioni e sviluppi futuri

Questo lavoro di tesi è nato nel contesto dell'emergente linguaggio di programmazione per SOA JOLIE, e ha avuto come obiettivo principale l'implementazione di Joliepse, un IDE per tale linguaggio, mentre come obiettivo secondario la sostituzione del parser attuale con uno equivalente generato da un tool automatico.

Il punto comune a entrambi gli aspetti è stata la definizione della grammatica EBNF del linguaggio, che ha richiesto un'attenta analisi del parser attuale di JOLIE, scritto a mano senza l'utilizzo di generatori.

Piattaforme come Eclipse e Net Beans forniscono tool per l'implementazione di IDE per nuovi linguaggi, e il primo passo nell'utilizzo di questi strumenti è la definizione di una grammatica formale per il linguaggio in questione, dalla quale produrre un parser utilizzando un parser generator. Quasi ogni funzionalità dell'IDE, dal controllo sintattico al proposal provider, è infatti implementata sfruttando il parser e gli AST generati da questo.

La considerazione fatta è quindi stata quella di poter utilizzare lo stesso strumento per realizzare entrambi gli obiettivi.

Il framework Xtext è apparso quindi, fra tutti quelli disponibili allo stato dell'arte, quello maggiormente indicato a tale scopo, poiché è in grado di fornire un IDE ricco di funzionalità per il linguaggio target, integrabile nella piattaforma Eclipse ad un costo poco elevato.

Xtext non è pensato per la creazione di IDE come gli altri strumenti che sono stati presentati nel secondo capitolo, ma piuttosto per l'implementazione di ogni aspetto di un linguaggio, e la generazione di un ambiente di sviluppo è soltanto una sfaccettatura di questo processo.

Gli strumenti generati da Xtext, come lexer, parser e validator possono essere infatti utilizzati in modalità stand-alone, all'interno di qualsiasi applicazione Java e quindi anche all'interno dell'interprete di JOLIE, aprendo quindi la strada alla sostituzione del parser attuale.

Per quanto riguarda l'IDE Joliepse, abbiamo visto nel quinto capitolo come sia stato relativamente semplice adattare i componenti generati da Xtext durante la fase di configurazione: sfruttando principalmente il framework di modellazione EMF e Google Guice per il meccanismo di dependency injection, componenti come scope provider, validator e linking provider sono stati modificati per gestire elementi del linguaggio non definibili facilmente a livello della grammatica come il costrutto *with*, o per aumentare la produttività degli sviluppatori come nel caso della direttiva *include*.

Nell'ultimo capitolo abbiamo analizzato il parser attuale di JOLIE, un parser a discesa ricorsiva che utilizza backtracking.

La necessità di sostituzione del parser attuale con uno generato da uno strumento automatico è dovuta ad un maggior controllo che si arriverebbe ad avere sul linguaggio, permettendo di poter modificare ed estendere con nuovi costrutti JOLIE concentrandosi soltanto sulla grammatica e sull'interprete, senza più dover considerare l'analisi sintattica e lessicale.

Come abbiamo visto le API di Xtext permettono un uso agevole dei componenti del linguaggio in modalità stand-alone. In termini pratici, richiedere al framework l'AST corrispondente ad un file sorgente richiede due linee di codice, mascherando completamente l'utilizzo del lexer, del parser e del validator.

Questa versatilità non è comunque gratuita: come è stato evidenziato dai test eseguiti tramite strumenti di profiling del codice, il framework Xtext risulta particolarmente lento. I circa tre secondi necessari al caricamento com-

pleto di un AST, ampiamente tollerabili all'interno dell'editor di un IDE, non permettono l'utilizzo di tale soluzione all'interno dell'interprete di JOLIE.

Da questa fase di test di performance è emerso che in realtà il 90 per cento del tempo richiesto da Xtext è da attribuire alla fase di inizializzazione di tutti i componenti del linguaggio effettuata tramite il framework Google Guice. Già eliminando i binding di tutte i componenti non necessari all'utilizzo in modalità stand-alone, si ottiene un dimezzamento del tempo d'esecuzione, che comunque non è ancora tollerabile.

A questo punto, possiamo individuare due possibilità di proseguimento delle indagini:

- una modifica al codice del framework Xtext per eliminare l'uso di Google Guice e abbattere i tempi di inizializzazione;
- una traduzione della grammatica dalla sintassi di Xtext alla sintassi di ANTLR, per ottenere un parser piu leggero di quello utilizzato da Xtext.

Un test effettuato inserendo un parser per il linguaggio Java generato con ANTLR all'interno dell'interprete di JOLIE, ha infatti dimostrato come il tempo richiesto sia sostanzialmente lo stesso necessario al parser attuale. Ricordiamo inoltre che ANTLR non produce un parser LL(*) table driven, bensì un parser a discesa ricorsiva, quindi strutturalmente simile a quello attuale.

Un ulteriore tentativo che andrebbe effettuato sarebbe quello di utilizzare IMP, esaminato nel secondo capitolo, abbandonando completamente l'utilizzo di Xtext e modificando notevolmente la grammatica.

Questo framework per l'implementazione di IDE infatti utilizza il parser generator per grammatiche LALR(1) LPG, e la grammatica utilizzata non e distinta da quella del parser generator come nel caso di Xtext e ANTLR, permettendo un utilizzo diretto del parser generato all'interno di JOLIE.

Come è stato evidenziato sempre nel secondo capitolo comunque, un IDE generato con IMP non fornisce le stesse funzionalità allo stesso costo di Xtext.

Sarebbe quindi da valutare il trade-off tra lo sforzo necessario al raggiungimento di un livello qualitativo dell'IDE generato paragonabile a Joliepse e la possibilità di ottenere senza ulteriori modifiche un nuovo parser per JOLIE.

In ultima analisi, la strada maggiormente sensata sarebbe quella di riscrivere la grammatica dalla sintassi di Xtext alla sintassi di ANTLR, molto simili fra loro, e ottenere un parser a discesa ricorsiva strutturalmente simile a quello attuale ma generato automaticamente.

Per quanto riguarda Joliepse, questa tesi ha fornito un punto di partenza per la creazione di un progetto che potrà crescere in numerose direzioni.

Il codice sorgente, disponibile sul sito di JOLIE, è ricco di commenti e questa tesi vuole inoltre fornire una documentazione utile a tutti coloro che volessero continuare a sviluppare Joliepse, oltre al sottoscritto.

Tra le principali feature che potrebbero essere implementate nel breve periodo possiamo indicare:

- aumento dei quick fix disponibili;
- migliormento del proposal provider sensibile al contesto;
- sistemi di refactoring;
- type checking;
- integrazione dell'interprete di JOLIE in Joliepse;
- integrazione con GMF per l'editing grafico e la generazione automatica di codice.

Le possibilità realizzative, dovute principalmente all'elevato grado di estensibilità di Eclipse e alla versatilità di Xtext e EMF, sono veramente molte e rendono Jolipese un ottimo punto di partenza per la realizzazione di uno strumento completo in grado di accompagnare il progettista di SOA in tutta la fase del processo di sviluppo.

Appendice A

La grammatica di Xtext per JOLIE

In questa appendice è riportata per esteso la grammatica di Xtext per JOLIE. Per una descrizione della sintassi della grammatica si rimanda alla seconda sezione del quarto capitolo.

```
/*************************************************************
  Author: Diego Castronuovo
  An Xtext grammar for the programming language Jolie
 *************************************************************/
grammar jolie.xtext.Jolie hidden(WS, ML_COMMENT, SL_COMMENT)

generate jolie "http://www.xtext.org/Jolie"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

Program:
  (constants+=Constant |
  include+=Include |
  ports+=Port |
  interface+=Interface |
  types+=Type |
  init+=Init |
```

```
17   execution+=Execution |
18   define+=Define |
19   embedded+=Embedded)*
20   main=Main?;
21
22 /*************************TYPES*********************************/
23
24 Constant:
25   'constants' '{' name+=ID ASSIGN (INT | STRING | REAL | ID)
26   (COMMA name+=ID ASSIGN (INT | STRING | REAL | ID))* '}';
27
28 Execution:
29   'execution' '{' (CONCURRENT | SEQUENTIAL) '}';
30
31 Include:
32   'include' importURI=STRING;
33
34 FileName:
35   name=STRING;
36
37 Type:
38   'type' name=ID COLON native_type_sub+=Native_type
39   typedef+=Typedef?;
40
41 Typedef:
42   {Typedef} '{'
43   subtypes+=Subtypes
44   '}';
45
46 Subtypes:
47   {Subtypes} (DOT name+=ID Cardinality? COLON
48   native_type+=Native_type typedef+=Typedef*)* | {Subtypes}
49   QUESTION;
```

```
50
51 Cardinality:
52   QUESTION | \ac{AST}ERISK | ('[' INT COMMA (INT | ASTERISK) ']');
53
54  Native_type:
55    name+=("any" | "int" | "real" | "string" | "void"
56    | 'undefined' | 'double' | 'raw' )| type=[Type];
57
58
59 /************************END TYPES**********************************/
60
61 Embedded:
62   'embedded' '{' ('Java' | 'Jolie' | 'Javascript') COLON
63   string+=STRING (in+=ID name+=ID)?
64   (COMMA string+=STRING (in+=IDname+=ID)?)* '}';
65
66 Define:
67   'define' name=ID mainrocess=MainProcess;
68
69 Init:
70   name+='init' mainrocess=MainProcess;
71
72 Main:
73   name='main' mainrocess=MainProcess;
74
75 MainProcess:
76   {ParallelStatement} '{' parallelStatement=ParallelStatement '}';
77
78 Process:
79   {Process} '{' parallelStatement+=ParallelStatement '}' |
80   '(' parallelStatement+=ParallelStatement ')';
81
82 ParallelStatement:
```

```
83   {ParallelStatement} (sequenceStatement+=
84    SequenceStatement (VERT sequenceStatement+=
85     SequenceStatement)*);
86
87 SequenceStatement:
88   {SequenceStatement} (basicStatement+=BasicStatement
89   (SEMICOLON basicStatement+=BasicStatement)*);
90
91 NDChoiceStatement:
92
93   {NDChoiceStatement} ('[' (('linkIn' '(' ID ')') |
94   (operation+=[OneWayOperationSignature] |
95   operation+=[RequestResponseSignature]
96     op+=InputOrOutputOperationDefOrCall)) ']'
97       mainProcess+=MainProcess)+;
98
99 BasicStatement:
100  {Process} process=Process |
101  {BasicStatement} assignStatementOrPostIncrementDecrement=
102    AssignStatementOrPostIncrementDecrementOrInputOperation |
103  NDChoiceStatement=NDChoiceStatement |
104  preIncrementDecrement=PreIncrementDecrement |
105  With | Synchronized | Undef | For
106  | If | Foreach | While | {BasicStatement} 'nullProcess' |
107  linkIn | linkOut | call=[Define] |
108  operation=[OneWayOperationSignature] |
109  operation=[RequestResponseSignature]
110     op=InputOrOutputOperationDefOrCall |
111  scope=Scope | compensate=Compensate |
112        throw=Throw | install=Install | {cH} cH | {Exit} Exit;
113
114 Is_function:
115  {Is_function} ('is_defined' | 'is_string' |
```

```
116   'is_double' | 'is_int') '(' variablePath=VariablePath ')';
117
118 Install:
119   'install' '(' installFunction=InstallFunciton ')';
120
121 Throw:
122   'throw' '(' name+=ID (COMMA espression+=Expression)* ')';
123
124 Compensate:
125   'comp' name=ID mainProcess=MainProcess;
126
127 Scope:
128   'scope' '(' name=ID ')' mainProcess=MainProcess;
129
130 InputOrOutputOperationDefOrCall:
131   '(' variablePath=VariablePath? ')' inputOperation=InputOperation
132   AT outputPortCall=OutputPortCall;
133
134 linkIn:
135   {linkIn} 'linkIn' '(' name=ID ')';
136
137 linkOut:
138   {linkOut} 'linkOut' '(' name=ID ')';
139
140 cH:
141   'cH';
142
143 Exit:
144   'exit';
145
146 AssignStatementOrPostIncrementDecrementOrInputOperation:
147    variablePath=VariablePath rightSide=RightSide;
148
```

```
149 RightSide:
150   {RightSide} ASSIGN expression=Expression |
151   {RightSide} CHOICE | {RightSide} DECREMENT |
152   {RightSide} POINTSTO variablePath=VariablePath |
153   {RightSide} DEEPCOPYLEFT variablePath=VariablePath;
154
155 /****************END BASIC STATEMENTS************************/
156 Synchronized:
157   'synchronized' '(' name+=ID ')' mainProcess+=MainProcess;
158
159 Undef:
160   'undef' '(' variablePath=VariablePath ')';
161
162 OutputPortCall:
163
164   port=[InputPortStatement] | port=[OutputPortStatement]
165     '(' expression=Expression? ')' /*Notification Operation*/
166       ('(' (variablePath=VariablePath)? ')'
167           ('[' installFunction=InstallFunciton ']')?)?;
168
169 InstallFunciton:
170   (name+=ID | 'this') '=>' parallelStatement+=
171     ParallelStatement (COMMA (name+=ID | 'this')
172       '=>' parallelStatement+=ParallelStatement)*;
173
174 InputOperation:
175   {InputOperation} ('(' expression=Expression? ')'
176     mainProcess=MainProcess)?;
177
178 PreIncrementDecrement:
179   (CHOICE | DECREMENT) variablePath=VariablePath;
180
181 /********************** LOOPS *************************/
```

```
182 If:
183   'if' '(' condition+=Condition ')' ifProcess=BasicStatement
184     ('else' elseProcess=BasicStatement | ifNasted=If)?;
185
186 For:
187   'for' '(' parallelStatement+=ParallelStatement COMMA
188     condition=Condition COMMA parallelStatement+=
189       ParallelStatement body=Body;
190
191 Body:
192   ')' BasicStatement;
193
194 Condition:
195   NOT condition+=Condition | '(' condition+=Condition
196     ')' (('&&' | '||') condition+=Condition)? |
197   {Condition} (variablePath=VariablePath | INT |
198   STRING | isF=Is_function) rightCondition=RightCondition?
199   (('&&' | '||') condition+=Condition)?;
200
201 RightCondition:
202   (EQUAL | LANGLE | RANGLE | MAJOR_OR_EQUAL |
203   MINOR_OR_EQUAL | NOT_EQUAL) expression=Expression;
204
205 Foreach:
206   {Foreach} 'foreach' '(' var1=VariablePath COLON
207     var2=VariablePath body=Body;
208
209 While:
210   'while' '(' condition+=Condition ')' mainProcess=MainProcess;
211
212 /***********************END LOOPS **********************/
213
214 /*************** ARITHMETIC EXPRESSION*****************/
```

```
215 Expression:
216   TerminalExpression ({Operation.left=current}
217     op=(PLUS | MINUS | ASTERISK | DIVIDE) right=Expression)?;
218
219 TerminalExpression returns Expression:
220   '(' Expression ')' | {IntLiteral} MINUS? value=INT |
221   {RealLiteral} MINUS? value=REAL | {String} value=STRING |
222   (CHOICE | DECREMENT) variablePath=VariablePath |
223   variablePath=VariablePath (CHOICE | DECREMENT)?;
224
225 VariablePath:
226   {VariablePath} HASH? dot+=DOT? name+=(ID)
227     ('[' children+=Expression ']')? (((DOT (ID | 'global') ('['
228       children+=Expression ']')?) | (DOT '(' children+=Expression
229         ')')))* | {VariablePath} 'global' (((DOT (ID | 'global') ('['
230   children+=Expression ']')?) | (DOT '(' children+=Expression ')')))*
231 ;
232
233 With:
234   {With} 'with' '(' name=VariablePath ')' mainrocess=MainProcess;
235
236 /************END ARITHMETIC EXPRESSION*****************/
237
238
239 /**************************PORTS*****************************/
240 Port:
241   (inputPortStatement+=InputPortStatement |
242   outputPortStatement+=OutputPortStatement);
243
244 InputPortStatement:
245   'inputPort' name=ID '{' ((location+=Location)? &
246     (protocol+=Protocol)? & (oneWayOperation+=OneWayOperation)? &
247       (RequestResponseOperation+=RequestResponseOperation)? &
```

```
248         (redirects+=Redirects)? & (aggregates+=Aggregates)? &
249           (intefaces=Interfaces)?) '}';
250
251 OutputPortStatement:
252   'outputPort' name=ID '{' ((location+=Location)? &
253     (protocol+=Protocol)? &
254     (oneWayOperation+=OneWayOperation)? &
255         (RequestResponseOperation+=RequestResponseOperation)? &
256             (intefaces+=Interfaces)?) '}';
257
258 OneWayOperation:
259   {OneWayOperation} 'OneWay' COLON
260     name=OneWayOperationSignature;
261
262 RequestResponseOperation:
263   {RequestResponseOperation} 'RequestResponse' COLON
264     name=RequestResponseSignature;
265
266 OneWayOperationSignature:
267   {OneWayOperationSignature} name=ID
268     ('(' TypeOfThrow+=TypeOfThrow ')')?
269       (COMMA op=OneWayOperationSignature)?;
270
271 RequestResponseSignature:
272   {RequestResponseSignature} name=ID
273     ('(' TypeOfThrow+=TypeOfThrow ')'
274       '(' TypeOfThrow+=TypeOfThrow ')')?
275         ('throws' (faults+=ThrowsClause)+)?
276           (COMMA op=RequestResponseSignature)?;
277
278 ThrowsClause:
279   name+=ID ('(' (type=TypeOfThrow) ')')?;
280
```

```
281 TypeOfThrow:
282   type=[Type] | {TypeOfThrow} naiveType=Native_type;
283
284 Location:
285   'Location' COLON uri+=Uri;
286
287 Uri:
288   {Uri} name+=ID | {Uri} STRING; //|name=STRING; //to do...
289 Interfaces:
290   'Interfaces' COLON interface+=[Interface]
291       (COMMA interface+=[Interface])*;
292
293 Protocol:
294   'Protocol' COLON name+=ID
295       protocolConfiguration=ProtocolConfiguration?;
296
297 ProtocolConfiguration:
298
299   mainProcess=MainProcess;
300
301 Redirects:
302   'Redirects' COLON redRef+=RedirectDef
303       (COMMA redRef+=RedirectDef)*;
304
305 RedirectDef:
306   name=ID '=>' outputPortIdentifier=ID;
307
308 Aggregates:
309   'Aggregates' COLON name+=ID (COMMA name+=ID)*;
310
311
312 Interface:
313   'interface' name=ID '{'
```

```
314   (RequestResponseOperation+=RequestResponseOperation? &
315     oneWayOperation+=OneWayOperation?) '}';
316
317 /*************************TERMINAL*******************************/
318
319 terminal CONCURRENT:
320   'concurrent';
321
322 terminal SEQUENTIAL:
323   'sequential';
324
325 terminal SEMICOLON:
326   ';';
327
328 terminal COLON:
329   ':';
330
331 terminal PLUS:
332   '+';
333
334 terminal VERT:
335   '|';
336
337 terminal ASSIGN:
338   '=';
339
340 terminal DOT:
341   '.';
342
343 terminal COMMA:
344   ',';
345
346 terminal AT:
```

```
347   '@';
348
349 terminal CHOICE:
350   '++';
351
352 terminal DECREMENT:
353   '--';
354
355 terminal \ac{AST}ERISK:
356   '*';
357
358 terminal QUESTION:
359   '?';
360
361 terminal DIV\ac{IDE}:
362   '/';
363
364 terminal POINTSTO:
365   '->';
366
367 terminal DEEPCOPYLEFT:
368   '<<';
369
370 terminal MINUS:
371   '-';
372
373 terminal PERCENT_SIGN:
374   '%';
375
376 terminal EQUAL:
377   '==';
378
379 terminal LANGLE:
```

```
380   '<';
381
382 terminal RANGLE:
383   '>';
384
385 terminal HASH:
386   '#';
387
388 terminal MAJOR_OR_EQUAL:
389   '>=';
390
391 terminal MINOR_OR_EQUAL:
392   '<=';
393
394 terminal NOT_EQUAL:
395   '!=';
396
397 terminal NOT:
398   '!';
399
400 terminal REAL:
401   ('0'..'9')* DOT ('0'..'9')+
402    (('e' | 'E') ('0'..'9')+)?;
403
404 terminal ID:
405   ('^'? ('a'..'z' | 'A'..'Z' | '_')
406     ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*);
407
408 terminal INT returns ecore::EInt:
409   ('0'..'9')+;
410
411 terminal STRING:
412   '"' ('\\' ('b' | 't' | 'n' | 'f' | 'r' |
```

```
413     '"' | "'" | '\\') | !('\\' | '"'))* '"' |
414   "'" ('\\' ('b' | 't' | 'n' | 'f' | 'r' |
415     '"' | "'" | '\\') | !('\\' | "'"))* "'";
416
417 terminal ML_COMMENT:
418   '/*'->'*/';
419
420 terminal SL_COMMENT:
421   '//' !('\n' | '\r')* ('\r'? '\n')?;
422
423 terminal WS:
424   (' ' | '\t' | '\r' | '\n')+;
425
426 terminal ANY_OTHER:
427   .;
```

Bibliografia

[TeiRep81] Teitelbaum T., Reps T., *The Cornell program synthesizer: a syntax-directed programming environment*, Commun. ACM 24, 1981, 563-573.

[BahSne86] Bahlke R., Snelting G., *The PSG system: From formal language definitions to interactive programming environments*, ACM 8, 1986, 547-576.

[BCD+88] Borras P., Clement D., Despeyroux T., Incerpi J., Kahn G., Lang B., e Pascual V. *Centaur: The system*, Proceedings of the 3rd International Symposium on Practzcal Software Development Environments, SIGSOFT Notes 13, 1988,14-24.

[Eti06] Effinge S., *oAW Xtext - A framework for textual DSLs*, In Workshop on Modeling Symposium at Eclipse Summit, 2006.

[MLM+06] MacKenzie C.M, Laskey K., McCabe F., Brown P.F., Metz R. , Hamilton B.A., *OASIS SOA Reference Model TC. Reference model for service oriented architecture 1.0*, Technical report, OASIS, 2006.

[PapHeu07] Papazoglou M.P, Willem-Jan van den Heuvel W., *Service oriented architectures: approaches, technologies and research issues*, VLDB J. 16(3): 389-415 (2007)

[Erl05] Erl T., *Service-Oriented Architecture: Concepts, Technology, and Design*, Upper Saddle River, Prentice Hall PTR, 2005.

[KFG+06] Kuo D., Fekete A., Greenfield P., Nepal S., Zic J., Parastatidis S., Webber J.: *Expressing and Reasoning about Service Contracts in Service-Oriented Computing*, ICWS 2006:915-918

[W3C03] *Web Services Architecture*, URL:http://www.w3.org/TR/ws-arch/, 2003

[OASIS07] *Web Services Business Process Execution Language Version 2.0* OASIS Standard, URL: docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, 2007.

[W3C05] *Web Services Choreography Description Language Version 1.0*, URL:http://www.w3.org/TR/ws-cdl-10/, 2003

[W3C03a] Gudgin M. et al.*SOAP Version 1.2 Part 1: Messaging Framework*, URL:http://www .w3.org/TR/2003/REC-soap12-part1-20030624, 2003

[W3C04] , Chinnici R. et al, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, eds. August 3, 2004. http://www.w3.org/TR/2004/WD-wsdl20-20040803.

[MGLZ06] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. *JOLIE: a Java Orchestration Language Interpreter Engine*. In CoOrg06, volume to appear of ENTCS

[GLG+06] Guidi C., Lucchi R., Gorrieri R., Busi N., and Zavattaro G., *Sock: A calculus for service oriented computing*, In Service Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings, volume 4294 of LNCS, pages 327–338, 2006.

[MVS10] *Microsoft Visual Studio*, Official Web Site, URL: http://www.microsoft.com/visualstudio/

[Ecl10] *Eclipse - The Eclipse Foundation open source community website*, URL:http://www.eclipse.org/

[Net10] *NetBeans IDE*, Official Web Site, URL:http://netbeans.org/

[Ora10] *Code Conventions for the Java Programming Language*, URL:http://www.oracle.com/technetwork/java/codeconv-138413.html

[Vel10] Apache Velocity Site, *The Apache Velocity Project*, URL:http://velocity.apache.org/

[Fre10] *FreeMarker: Java Template Engine Library*, Overview, URL: http://freemarker.sourceforge.net/

[FMR99] Fowler M., *Refactoring: Improving the Design of Existing Code*, Addison - Wesley, Reading MA, 1999.

[YanJian07] Yang Z., Jiang M., *Using Eclipse as a Tool-Integration Platform for Software Development*, IEEE Software 24 (2) (2007) 87–89.

[Net10] *NetBeans Rich-Client Platform Development (RCP)*, URL:http://netbeans.org/features/platform/

[Vau03] Vaughan-Nichols S. J., *The Battle over the Universal Java IDE*, in IEEE Computer, Volume 36 Issue 4. pp. 21 - 23. ACM Press, Apr. 2003.

[OMG10] *The Object Management Group*, URL: http://www.omg.org/

[Gee05] Geer D, *Eclipse becomes the dominant Java IDE*, IEEE Computer, 38(7):16-18, 2005.

[Ecl05] *Eclipse Java development tools (JDT)*, URL: http://www.eclipse.org/jdt/

[Ecl06] *Eclipse C/C++ Development Tooling (CDT)*, URL: http://www.eclipse.org/cdt/

[Ecl06] *Eclipse Dynamic Languages Toolkit*, URL: http://www.eclipse.org/dltk/

[YRR+09] Yongying H., Ruisheng Z., Ruipeng W., Caihua H., Lian L., *Designing a Chemical Script Editor in Grid environment with DLTK*, in ChinaGrid Annual Conference, 2009. ChinaGrid '09. Fourth, pages 159 - 165, 21-22 Aug. 2009

[AngMar05] Angelov, K. Marlow, S. (2005), *Visual Haskell: a full-featured Haskell development environment*, In Proceedings of ACM Workshop on Haskell, Tallinn, Tallinn, Estonia. ACM.

[Gom08] Gomanyuk, S.V., *An Approach to Creating Development Environments fo a Wide Class of Programming Languages*, Programmirovanie, 2008, no. 4, pp.225–236 Programming Comput. Software (Engl. Transl.), 2008, vol. 34, no. 4, pp. 225–236.

[CFS07] Charles P., Fuhrer R. M., Sutton S. M., *IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse* In R. E. K. Stirewalt, A. Egyed, and B. Fischer editors, ASE 2007, pages 485–488. ACM, 2007.

[Eff06] Efftinge S.. *oAW Xtext - A framework for textual DSLs*, In Workshop on Modeling Symposium at Eclipse Summit, 2006.

[Ite10] *Itemis AG, Model Based Software Development*, URL: http://www.itemis.com/

[janPru08] Jancura J., Prusa D., *Generic framework for integration of programming languages into netbeans IDE*, in PEPM '08 Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, New York, NY, USA 2008

[ParQuo95] Parr T.J., Quong W., *ANTLR: A predicate-ll(k) parser generator*, Software: Practice and Experience, 25(7):789–810, July 1995.

[GreSho04] Greenfield J., K. Short, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*, John Wiley and Sons, 2004

[BezGer01] Bézivin J., Gerbé O. , *Towards a Precise Definition of the OMG/MDA Framework*, In Proceedings of the 16th IEEE international Conference on Automated Software Engineering, pages 273-280, IEEE Computer Society, Washington, DC, 2001.

[BSM+03] Budinsky, F., Steinberg, D., Merks E., Ellersick R., Grose T., *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[Van08] R. Vanbrabant, *Google Guice: Agile Lightweight Dependency Injection* Framework. APress, 2008.

[JHA+08] Johnson R., Hoeller J., Arendsen A., Sampaleanu C., Harrop R., Risberg T. et al. (2008), *The Spring framework, Reference documentation*, Version 2.5.6. URL: http://static.springsource.org/spring/docs/2.5.x/spring-reference.pdf

[Ecl04] *Eclipse Consortium, Eclipse Graphical Modelling Framework (GMF)* – Version 2.1.3, 2004, URL: http://www.eclipse.org/gmf.

[Osg03] *OSGi Service Platform Release 3*, OSGi Alliance, IOS Press, December 2003. URL: http://www.osgi.org/Specifications/

[BSE+03] Budinsky F., Steinberg D., Ellersick R., Merks E., Brodsky S.A., Grose T.J.: *Eclipse Modeling Framework*, Addison Wesley (2003)

[GruJac90] Grune D., Jacobs. J.H., *Parsing Techniques, A Practical Guide*, Ellis Horwood, Chichester, England, 1990

[Coo07] Copeland, T., *Generating Parsers with JavaCC* Centennial Books, 2007

[Joh78] Johnson S. C., *Yacc - Yet Another Compiler-Compiler*, Bell Laboratories Computing Science Technical Report n. 32, July 1978.

[Lpg10] *LPG, LALR parser generator*, URL: http://www.sourceforge.net/projects/lpg/

[BBB03] Bailliez A., Barozzi N., Bergeron J.. *Apache Ant User Manual*, The Apache Software Foundation, 2003