UNIVERSITA' DEGLI STUDI DI BARI
ALDO MORO

CORSO DI LAUREA IN
INFORMATICA MAGISTRALE

Documentazione

DepJaeger: un analizzatore di sorgenti prolog a caccia di informazioni su moduli, predicati e dipendenze fra di essi

AUTORI:
Nicola Alessandro Natilla
Damiano Romita

ANNO ACCADEMICO 2012/2013

"Per aspera sic itur ad astra."
- Seneca

Indice

Elenco delle figure

Capitolo 1

Introduzione, analisi dei requisiti e generalità

1.1 Finalità di DepJaeger

Il sistema DePJaeger nasce dalla necessità di esaminare le dipendenze di moduli e predicati facenti parte di programmi Prolog di una certa complessità, al fine di individuare eventuali moduli inutilizzati e snellire l'impalcatura software eliminandoli. Altra finalità è quella di disporre di uno strumento automatico per potersi fare rapidamente un'idea dei moduli utlizzati nel sistema in uso, di quali predicati questi moduli contengano ed esaminare le relazioni e interazioni fra i vari predicati e relativi moduli, possibilità che si rivela molto utile durante il refactoring dei suddetti sistemi.

1.2 Requisiti di sistema

Il sistema è stato testato su macchine sulle quali sono installati i seguenti software:

- S.O. Linux o Unix-based

- YAP 6.2.2, reperibile da
  http://www.dcc.fc.up.pt/ vsc/Yap/yap-6.2.2.tar.gz
- graphviz (per usufruire della funzione di generazione del grafo), reperibile da http://www.graphviz.org/

1.3 Modalità di esecuzione

Per eseguire DepJaeger è necessario che la cartella core sia nella stessa cartella in cui si trova il file .jar fornito. Si può poi procedere a digitare, dopo essersi posizionati nella cartella in cui è contenuto il file .jar, la seguente stringa nel terminale:

```
java -jar jaeger.jar path_assoluto_di_yap
```

Per visualizzare il path assoluto di yap è disponibile il comando *whereis yap* per tutti i sistemi opertaivi con base Unix. In particolare, sul sistema operativo MacOS è anche disponibile il comando *which yap* che rileva i binari installati in percorsi non-standard

Capitolo 2

Analisi dei requisiti

Come accennato nell'introduzione, il sistema deve poter svolgere i seguenti task:

- l'esplicitazione delle relazioni latenti presenti in ciascun file
- l'analisi delle strutture (parti destre) dei predicati al fine di analizzare le relazioni interne, individuabili in ciascun file
- l'analisi delle relazioni esterne, intercorrenti fra file diversi, al fine di analizzare le dipendenze attive/inattive fra questi

2.1 Prolog e la programmazione logica

La programmazione logica è un paradigma di programmazione che adotta la logica del primo ordine sia per rappresentare che per manipolare l'informazione, utilizzando le clausole di Horn, nonchè per elaborare teorie logiche e per compiere inferenze. La programmazione logica nasce all'inizio degli anni '70 grazie agli studi di due ricercatori, Kowalski e Colmerauer. Il primo elaborò le basi teoriche del paradigma, affermando la doppia interpretazione (procedurale e dichiarativa) delle clausole di Horn che ha permesso l'implementazione su calcolatore dei linguaggi di programmazione logica. Il secondo progettò ed implementò un interprete per un linguaggio logico. Da queste basi nasce Prolog.

La programmazione logica appartiene alla famiglia dei paradigmi di programmazione descrittivi, nei quali la componente descrizionale (i dati sul quale il programma opera) è totalmente slegata dalla componente operazionale (come elaborarli), che viene presa in carico totalmente dall'elaboratore. Ciò permette all'utente di operare ad un livello di astrazione più alto rispetto ad un linguaggio imperativo, in quanto egli deve solo definire le specifiche del problema senza istruire la macchina sul "come" fare.

Nella programmazione logica, i programmi sono descrizioni delle soluzioni (goal) e non del processo solutivo attraverso il quale raggiungerlo. Tale programma viene elaborato descrivendo in un linguaggio formale (le clausole di Horn) il dominio applicativo (oggetti, relazioni, fatti, ecc).

Prolog è un linguaggio di programmazione logica adatto a problemi che riguardano oggetti, strutturati e non, e relazioni fra di essi. Per poter elaborare le informazioni richieste, il Prolog permette di interagire "rispondendo" alle domande che l'utente gli pone: difatti, sfruttando la base di conoscenza (che l'utente gli ha impartito), egli è capace di eseguire delle deduzioni che corrispondono alle risposte alle domande dell'utente.

In sintesi, le componenti fondamentali sono:

- dichiarazione di fatti sugli oggetti e sulle loro relazioni
- dichiarazioni di regole sugli oggetti e sulle loro relazioni
- interrogazioni sugli oggetti e sulle loro relazioni

Le prime due componenti permettono di definire il dominio del problema (in manera descrittiva), mentre l'ultima permette di eseguire il programma (in maniera operativa). Come è facilmente intuibile dal titolo, si è scelto si seguire la strada del riconoscimento delle azioni.

2.2 Sintassi Prolog

Un programma Prolog è costituito da un insieme di clausole definite della forma:

- A.
- A :- $B_1, B_2, \ldots, B_n$

in cui A e B_i sono formule atomiche, A viene detta testa della clausola e la congiunzione $B_1, \ldots, B_n$ viene detto corpo della clausola. Una clausola come la 1 (ossia una clausola il cui corpo è vuoto) prende il nome di fatto (o asserzione), mentre una clausola come la 2 prende il nome di regola. Il simbolo "," viene utilizzato per indicare la congiunzione mentre il simbolo ":-" indica l'implicazione logica. Una formula atomica è una formula del tipo: $p(t_1, t_2, \ldots, t_m)$ in cui p è un simbolo di predicato e $t_1, \ldots, t_m$ sono termini. Per maggior chiarezza richiamiamo anche la definizione di termine, definita ricorsivamente:

- le costanti sono termini; in Prolog le costanti sono costituite dai numeri (interi e floating point) e dagli atomi alfanumerici (il cui primo carattere deve essere un carattere alfabetico minuscolo)
- le variabili sono termini; in Prolog le variabili sono stringhe alfanumeriche aventi come primo carattere un carattere alfabetico maiuscolo oppure il carattere "_"
- $f(t_1, t_2, \ldots, t_k)$ è un termine se f è un simbolo di funzione (o operatore) a k argomenti e $t_1, \ldots, t_k$ sono termini

Le costanti possono essere considerate come simboli funzionali a zero argomenti, come ad esempio *gino, pluto, pippo, a91,10.134*.
Le variabili sono caratterizzate dalla prima lettera maiuscola, come ad esempio *X,X1, Pippo, _x, _* (quest'ultima, definita con il solo carattere "_", prende il nome di variabile anonima).
Una costante può essere anche composta da un programma prolog o da clausole definite.
Un goal (o query) ha come forma generale:
:- $B_1, B_2, \ldots, B_n$. , dove $B_1, \ldots, B_n$ sono formule atomiche.

Il problema di individuazione di moduli e predicati utilizzati da un dato modulo o predicato è riconducibile ad un problema di scoperta dei nodi

adiacenti in un grafo orientato ciclico, a partire da determinati nodi.

Capitolo 3

Progettazione

In questa fase verranno individuati e descritti i concetti principali, ovvero:

- oggetti, ovvero le entità che intendiamo rappresentare e analizzare
- relazioni, che collegano fra di loro gli oggetti

3.1 Oggetti del dominio

Gli oggetti individuati inizialmente sono due: file contenenti codice sorgente Prolog e le singole righe contenute in ciascuno di essi e costituenti i token di base usati per tutte le fasi successive.
A loro volta, i file possono essere di due tipi: *moduli* e *non-moduli*. Verranno trattati in maniera leggermente differente, come documentato nei paragrafi successivi.
Verranno esaminate le sole relazioni intercorrenti fra predicati e moduli.

Per quanto riguarda i token analizzati, ovvero le righe di codice presenti nel file, questi possono risultare di tre tipi: *predicati*, *fatti* e *goal*.
I predicati sono in forma di clausole di Horn, ossia presentano un corpo formato da una serie di letterali legati dall'operatore *AND* e dotata di un singolo termine in testa (ad esempio *a(X) :- b(X), c(X,Y), d(Y).*). Inoltre, un predicato è composto da una o più clausole, nelle quali le teste hanno lo stesso nome e la stessa arità.

I fatti sono clausole sprovviste di corpo, mentre i goal sono clausole sprovviste della testa (ad esempio: *:- use_module(library(lists))* oppure *:- main.*).
Un predicato, in Prolog, è dotato di un identificatore univoco (all'interno di ciascun modulo) nella forma *Nome/Arietà*.
I moduli vengono visti come una collezione di predicati.

E' possibile che si crei ambiguità fra un "modulo" propriamente definito e strutturato, e il generico file contenente del codice prolog. E' perciò necessaria una precisazione: i moduli propriamente detti (sorgenti prolog recanti in testa la direttiva *:- module(nomeMod,PredEsposti)*) offrono un meccanismo di information hiding che limita la visibilità e la possibilità di utilizzo ai soli predicati presenti nella lista dei predicati pubblici presente nella direttiva di cui sopra (salvo poter comunque accedere a tutti i predicati contenuti nel modulo tramite l'uso del nome del modulo seguito dall'operatore *:/2*). E' diverso il caso in cui il file non sia un modulo, in quanto tutti i predicati sono pubblici.

3.2 Individuazione delle relazioni

Una volta individuati gli oggetti è indispensabile evidenzare le relazioni che intercorrono fra di essi. Di seguito c'è un semplice esempio di come un programma Prolog viene trasposto in un grafo di relazioni fra predicati.

Il codice riportato nella figura 3.1 viene trasformato nel grafo visibile in 3.2. Il contenuto di simili grafi verrà poi espresso in fatti Prolog, di tipo *rule/4*, che verranno descritti in dettaglio nei prossimi paragrafi.
Per quanto riguarda le dipendenze fra moduli, queste possono essere di due tipi: utili o inutili.
Le dipendenze utili sono quelle per le quali un modulo A importa espressamente il modulo B e almeno un predicato di A usa almeno un predicato dichiarato in B.
Le dipendenze inutili, invece, sono quelle per le quali un modulo A importa espressamente un modulo B e non esiste alcun predicato di A che ne usa uno dichiarato in B.

```
parent(ivana,samuele).
parent(modesto,samuele).

ascendent(X,Y) :- parent(X,Y).
ascendent(X,Y) :- parent(X,Z), ascendent(Z,Y).

brother(X,Y) :- parent(Z,X), parent(Z,Y), X \= Y.

uncle(X,Y) :- brother(X,Z), parent(Z,Y), X \= Y.

cousin(X,Y) :- uncle(Z,X), parent(Z,Y), X \= Y.
```

Figura 3.1: Esempio di semplice codice Prolog

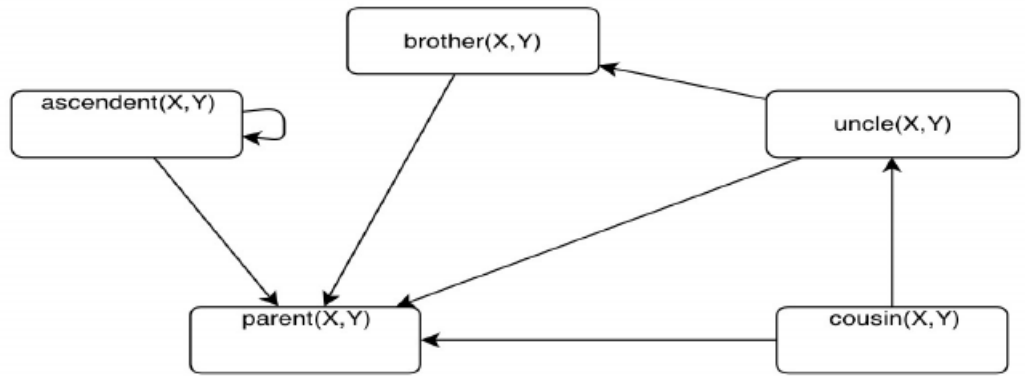

Figura 3.2: Trasposizione in forma di grafo del codice d'esempio

3.2.1 Espressione delle relazioni fra predicati all'interno del sistema

E' necessario inserire e conservare all'interno della memoria di lavoro tutte le relazioni dei tipi suddetti al fine di trarne delle informazioni rilevanti. Per preservare informazioni relative ai predicati, al loro contenuto, al loro modulo di appartenenza e al file i cui tale modulo si trova, vengono generati dei fatti con la seguente struttura: *rule(NomePredicato/Arietà, ListaPredicatiUsati, FilePathModulo, NomeModulo).*
A titolo di esempio, il predicato "test/4" seguente:

```
test(_,_,0,_) :-
        !.
test(Task,ProblemName,NRuns,Test_dir) :-
        RestNRuns is NRuns - 1,
        test(Task,ProblemName,RestNRuns,Test_dir),
        !, % per sicurezza (ma non dovrebbe lasciare punti di scelta)
        name(NRuns,Suff),
        %nl,write('Working Directory: '),
        %name(CDIR,Dir1),
        %cd(Test_dir,Dir),
        append(ProblemName,Suff,Prefix_classif),
        assert_pars(Task,Prefix_classif),
        inthelex(m).
```

Figura 3.3: Esempio di un predicato

viene trasformato nel seguente fatto di tipo *rule/4*:
rule(test/4, [(is)/2, test/4, !/0, name/2, append/3, assert_pars/2,inthelex/1], 'test_files/folle/folle.pl', folle).

3.2.2 Espressione delle relazioni fra moduli all'interno del sistema

Come precedentemente esposto, la relazione di utilizzo di un modulo A rispetto ad un modulo B può essere di due tipi: *utile* e *non-utile*.
Inizialmente, le dipendenze fra moduli vengono verificate e asserite attraverso fatti del tipo *module_imports(NomeModulo, ListaModuliImportati)*. In un momento successivo il sistema esprimerà se una relazione di dipendenza fra

due moduli è utile o non-utile.
Tale scelta è stata ponderata in fase di progettazione, in quanto l'operazione di ricerca di una dipendenza utile risulta essere molto onerosa dal punto di vista computazionale e potrebbe non interessare (inizialmente) l'utente finale, a cui potrebbe solo interessare conoscere la struttura interna di un modulo o di raccogliere informazioni su determinati predicati.

3.3 Implementazione e strategia solutiva

La parte Prolog del sistema è composta di tre file:

1. *core.pl*: contiene il nucleo del software. E' diviso in quattro sezioni
    - *high level predicates*, contenente i predicati per l'interazione col sistema da linea di comando
    - *medium level predicates*, contenente la logica applicativa
    - *low level predicates*, contenente i predicati adibiti alla individuazione ed esplicitazione delle relazioni di dipendenza
    - *java communication predicates*, contenente i predicati che permettono alla parte Java di consultare l'interprete Prolog: nella maggior parte dei casi, questi predicati riutilizzano i predicati di alto livello convertendo il loro formato in JSON.
2. *coreutils.pl*: contiene predicati di supporto all'attività di *core*, per esempio routine di servizio per il calcolo della dimensione di liste, e similari
3. *rendergraph*: contiene i metodi per la generazione delle immagini contenenti i grafi delle relazioni di dipendeza descritti più avanti.

E' inoltre presente la cartella *yap-json*, contenente la libreria utilizzata per la conversione di termini Prolog in oggetti JSON. Ulteriori dettagli verranno forniti nel prossimo capitolo.

Il sistema è in grado di analizzare sia file contenenti dei moduli, ossia dotati della direttiva *:- module*($nomeModulo, [listapredicatiesportati]$) in testa, sia file che non la contengono. Per risolvere tale ambiguità è stata adottata la seguente convenzione:
"Tutti i file vengono considerati moduli ai fini dell'analisi. Se un file è un modulo, gli verrà associato il nome dichiarato come argomento della direttiva module($nome\_modulo, lista\_predicati$) *in testa al file stesso. Altrimenti assumerà il nome del file, privato dell'estensione."*
Al fine di rispondere nel miglior modo possibile ai bisogni informativi dell'utente finale, i momenti durante i quali l'analisi di un file contenente codice Prolog viene svolta sono due:

1. analisi preliminare, capace di individuare le informazioni "statiche" riguardanti il file (predicati dichiarati in un modulo)

2. analisi approfondita, in grado di analizzare informazioni "dinamiche" (individuazione moduli utilizzati e non, individuazione name clashes, individuazione dei path dei moduli analizzati e non)

Riguardo all'analisi preliminare, il sistema analizza singolarmente ciascuno dei file indicati dall'utente. Ognuno di essi viene esaminato riga per riga alla ricerca di una delle possibili direttive:

- dichiarazione di un modulo:
  :- module(nome_modulo, lista_predicati_esportati)

- dichiarazione d'uso di moduli esterni: *consult, use_module, ensure_loaded* e *load_files* in tutte le loro varianti

- predicati in forma di clausole di Horn (e.g. *h(A) :- q(A), y(A, B), p(B).*)

Dopo aver letto una riga, il sistema ne valuta la struttura in base al contenuto. In una prima fase, la riga verrà esaminata alla ricerca di dichiarazioni d'uso di moduli esterni e, a partire da questa, verrà individuato il modulo o la libreria

utilizzata ed una serie di informazioni utili alla sua identificazione. In seguito, se la riga corrisponde ad un predicato ne verranno elaborate le parti destre e sinistre, ricostruendone le relazioni di dipendenza predicati-predicati e predicati-moduli. Al termine di questo processo disporremo, per ogni modulo analizzato, dei seguenti dati:

- predicati dichiarati (pubblici e non)
- moduli/librerie importati

Riguardo l'analisi approfondita, verrano esaminate nel dettaglio le informazioni precedentemente estratte: il sistema, per ogni libreria o modulo B importato da un modulo A, permette di determinare se B è una dipendenza utile o meno. Inoltre, per ogni elemento B disporremo del path assoluto del file che lo contiene. Come ultimo step, il sistema ricerca potenziali name clashes, ossia potenziali conflitti fra un modulo C, contenente un predicato *P=nome/arietà*, ed un modulo importato D che contiene un predicato P avente stesso nome ed arietà del precedente.

Ciascuno dei prossimi sotto-paragrafi sarà dedicato alla descrizione del comportamento del sistema in ciascuno dei possibili casi.

3.3.1 Inizializzazione

A partire da un elenco di file path (assoluti o relativi), per ognuno di essi il sistema verifica che contenga un modulo o no.
In base all'esito di tale controllo verrà adottata una diversa strategia iniziale, volta a uniformare le strutture presenti in ciascun file secondo la modalità di rappresentazione adottata dal sistema.
Di seguito viene illustrato in dettaglio il comportamento del sistema dei due casi:

- *Il file contiene un modulo*: ai fini dell'analisi, il sistema estrae dalla direttiva *:- module(nomeModulo, listaPredicatiPubblici)* le informazioni rilevanti memorizzandole all'interno del fatto *exports(nomeModulo, listaPredicatiPubblici)*

- *Il file non contiene un modulo*: non viene asserito un fatto di tipo *exports/2*, in quanto i file privi della direttiva module per default espongono pubblicamente tutti i predicati in essi dichiarati. Inoltre, il nome associato al modulo risulterà essere il nome del file privato dell'estensione. Per il resto, il file viene trattato come un modulo, compresa la generazione dei suddetti fatti di tipo *rule/4* per i suoi predicati

A questo punto, conclusa la fase di inizializzazione, il file viene letto e trasformato in una lista di termini attraverso il predicato *read_file_to_terms(+Filepath, -Terms)*.

3.3.2 Analisi dei moduli importati

Tramite la procedura *import(+Terms, -ListaModuliUtilizzati)*, che prende in input il contenuto del file trasformato nella lista *Terms*, si ottiene la lista dei moduli utilizzati. Gli step intermedi sono i seguenti:

1. Riconoscimento e acquisizione delle importazioni tramite le tre seguenti procedure:

   - *getExplicitImports(+Terms, -Moduli)*: prende in input la suddetta Terms e restituisce la lista dei moduli dichiarati esplicitamente, cercando tutte le varianti delle procedure per importare o consultare dei file. Tali procedure sono: *use_module*, *consult*, *load_files*, *ensure_loaded*. Di queste, *ensure_loaded* e *consult* possono avere solo un argomento (il nome del file o libreria da consultare), *load_files* necessita di due argomenti, mentre la *use_module* può avere da uno a tre argomenti.
   - *getInlineImports(+Terms, -Moduli)*: cerca tutti gli import "inline", ossia import presenti all'interno di parti destre di predicati.

   - *getSingleLineImports(Terms, Moduli)*: cerca tutti gli import multipli presenti su una singola riga.

```
:- ensure_loaded(library(lists)).
:- ensure_loaded(library(readutil)).
:- ensure_loaded(library(system)).
:- ensure_loaded([coreutils]).
:- ensure_loaded([cli]).
:- ensure_loaded([renderGraph]).
:- ensure_loaded(['yap-json/json.pl']).
```

Figura 3.4: Esempio di import esplicite

```
%+FilePath, -Filename
removeExtension(FilePath, Filename) :-
        file_base_name(FilePath, Fname),
        ensure_loaded(library(lineutils)),
        all(N,
                (atom_codes(Fname,L),
                atom_codes('.',Sep),
                split(L,Sep,RX),
```

Figura 3.5: Esempio di import in-line

```
:- use_module(library(lists)), use_module(library(dgraphs)).
```

Figura 3.6: Esempio di import multipli su una singola riga

I tre suddetti predicati di riconoscimento delle importazioni fanno uso della seguente importante procedura, la *recognizeImport(+Term, -moduleName)*: tramite unificazione, questa sotto-procedura riconosce ogni procedura di importazione, e per ciascuna di esse è in grado di determinare il tipo di percorso (se assoluto, relativo o di sistema) dei file o componenti passati come primo argomento. Se non si tratta di una libreria di sistema, è necessario capire se il file in oggetto è un modulo o meno. Se è un modulo, ne vengono estratti il nome e i predicati che esporta, altrimenti il file viene scansionato ricercando i predicati in esso definiti, assumendo che se un file non costituisce un modulo tutti i predicati che

contiene siano pubblici. Se l'importazione presenta un path esplicito, viene direttamente letto il file per estrarre le informazioni di cui sopra. Altrimenti, se l'argomento è una libreria di sistema (dichiarata con ":-library(nomeModulo)"), prima di poter effettuare la lettura del file, è necessario risalire al suo percorso assoluto. Se invece il path è implicito, prima di poter leggere il file è necessario ricostruire il suo percorso assoluto utilizzando il fatto *module_path* precedentemente asserito.

2. concatenazione delle liste finali, rimozione elementi duplicati e ordinamento dell'output.

3. asserzione dei fatti di tipo *module_imports(nomeModulo, ListaModuliUtilizzati).*

3.3.3 Analisi dei predicati

L'analisi del codice sorgente viene svolta da due procedure. La procedura *getCodeGraph(Terms, Grafo)* a partire dal contenuto del file è capace di individuare ciascuna clausola presente nel codice sorgente e restituirla sotto forma di una lista di predicati con le rispettive clausole, avente struttura $[TestaPredicato/Arieta' - [PredicatoContenuto_1/Arieta', \dots], PredicatoContenuto_n/Arieta']$. In un momento successivo, gli oggetti presenti in questa lista verranno raggruppati in base alla testa per costruire le regole definitive. Si riporta lo pseudocodice della procedura di generazione dei fatti rule/4:

- *getCodeGraph(Terms, Grafo)*: come specificato sopra, i predicati assumono la forma di un grafo. Scopo di questo metodo è esplicitare le relazioni latenti nel corpo del codice tramite i seguenti passi:
  - $\forall$ clausola di Horn C
    - C = *(Testa :- Corpo)*
    - *normalizza(Testa, Nome).* Normalizza il termine $Testa = testaPredicato(p_1, p_2, \dots, p_n)$ nel formato *testaPredicato/N* (dove N è l'arietà del predicato).
    - $\forall$ predicato P in $Corpo$

- *collectorize*$(+P, -P_1)$. Preliminarmente esamina ciascun corpo alla ricerca di eventuali goal collectors (setof, bagof, all, findall) e dei costrutti "catch" e "commit" (ovvero $P \rightarrow Q$, $P \rightarrow Q; R$). Se la *collectorize* non ha individuato un costrutto come quelli riportati sopra, restituisce il valore di input $+P$ invariato.
- $\forall$ elemento E di P_1, *normalizza(E, NomeElemento).*

* restituisci tutto in una lista rappresentante il grafo delle dipendenze $G = [[TestaClausola_1/Arieta' - [PredicatoCorpo_1/Arieta', ...]], [TestaClausola_2/Arieta' - [PredicatoCorpo_2/Arieta', ...]] \dots]$.

- costruzione regole intermedie: una volta esaminate le clausole, è necessario raggrupparle in base alla testa (un predicato può avere più clausole). Pertanto si procederà innanzitutto a costruire dei fatti temporanei di tipo $tmp(NomeModulo, Testa, [ClausoleCorpo])$.

- una volta costruite le suddette regole temporanee, vengono costruiti i fatti finali aventi una struttura del tipo $rule(NomeModulo, TestaPredicato_1/Arieta', [PredicatoCorpo_1/Arieta', ...], Path)$. Questa struttura consente di legare ciascuna testa ai rispettivi corpi, conservando al contempo i riferimenti al nome del modulo e al path del file che lo contiene).

- vengono cancellati i fatti intermedi $tmp/3$.

- dopo aver asserito i fatti finali viene stampato un report contenente il numero di importazioni e di predicati individuati.

3.3.4 Valutazione ed estrazione delle informazioni obiettivo

Una volta compiuta l'analisi preliminare (ricerca dei moduli importati e trasformazione dei predicati nella struttura $rule/4$), il sistema permette di

stampare le informazioni relative alla struttura dei predicati e dei moduli individuati.

Valutazione dipendenze fra i predicati

Tramite il comando *queryP(nomePredicato/Arietà)* il sistema presenta le informazioni relative ad un predicato (pubblico o non pubblico) presente in un file precedentemente analizzato. Si presenta uno pseudocodice degli step compiuti:

- *unique(nomePred)*: verifica l'esistenza e l'univocità del nome del predicato. Se il nome non è univoco, il sistema chiederà di disambiguare attraverso il comando *queryP(Predicato, Modulo)*.

- stampa il nome del modulo in cui è dichiarato il predicato ed il percorso del file in cui è memorizzato (dati ricavati tramite i predicati *declared_in(+Predicato, -Modulo, -RelPath)* e *moduleAbsoluteFilePath(+Modulo, -AbsPath)*).

- verifica e stampa se il predicato sia pubblico o meno attraverso il predicato *is_public(+Predicato, +Modulo, -Responso)*.

- stampa una lista dei predicati utilizzati, utilizzando la lista *ListaPredicatiUtilizzati* presente nel fatto *rule(Predicato, ListaPredicatiUtilizzati, FilePath, Modulo)* relativo al predicato in analisi. Questa avviene per mezzo del predicato *uses(+Predicato, +Modulo, -ListaPredUtilizzati)*.

- stampa una lista dei predicati che utilizzano il predicato in analisi tramite l'uso del predicato $used\_by(+Predicato_1, -Predicato_2)$. Tale ricerca viene effettuata incrociando i riferimenti fra i fatti di tipo *rule/4* asseriti durante l'analisi preliminare. Se il predicato non è utilizzato da nessuno, *used_by/2* restituisce una lista vuota.

- stampa la lista dei moduli che contengono i predicati utilizzati dal predicato in analisi. Tale ricerca viene effettuata tramite il predicato *mod_depends(+Pred, +Modulo, -ListaModuli)* che, per ogni predicato

appartente al corpo di *+Pred*, ricerca il modulo all'interno del quale il predicato è dichiarato attraverso il predicato *belongs_to(Predicato, Modulo)*. Quest'ultimo predicato ricerca la presenza del *Predicato* nei seguenti posti:

- all'interno dei fatti di tipo *rule/4*
- Se sono stati analizzati precedentemente altri moduli e se questi hanno importato a loro volta dei moduli, si cerca fra i predicati che questi ultimi esportano. Tale informazione è contenuta nei fatti di tipo *external_module/3*.
- Se il predicato passato è nel formato *Modulo:Predicato/Arietà*, viene restituito *Modulo*.
- Infine viene valutata l'appartenenza del predicato ad un modulo caricato nella working memory (importato come codice sorgente, cfr. *load_files*, voce *-compilation_mode=source*)

Valutazione dipendenze fra i moduli

Di seguito si presentano le procedure usate per valutare le dipendenze fra i moduli:

- *findModule(nomeModulo)*: verifica l'esistenza modulo
- *moduleAbsoluteFilePath(+nomeModulo, -FilePath)*: restituisce il percorso assoluto del modulo in oggetto
- *effectiveImportedModuleDeps(+nomeModulo, -ListaDeps)*: stampa le dipendenze dichiarate, individuate durante l'analisi preliminare, che risultano effettivamente utilizzate. L'effettivo utilizzo è determinato tramite i seguenti step:

  1. usa i fatti *module_imports/2* asseriti durante l'analisi preliminare per avere la lista di quali moduli sia dichiarata l'importazione.
  2. *used_imports/2*: per ogni modulo M_1 contenuto nella lista di moduli importati ottenuta nello step precedente, verifica che il modulo in esame M utilizzi almeno un predicato dichiarato in M_1. Tale

analisi viene effettuata tramite il predicato $usesModule(M, M_1)$ che verifica che all'interno di un qualsiasi predicato del modulo M venga utilizzato un predicato appartenente al modulo M_1 tramite la valutazione dell'intersezione fra l'insieme dei predicati usati nel modulo M e quelli dichiarati nel modulo M_1. Se tale intersezione è vuota, il predicato fallisce.

3. *un_used_imports/2.* Alla lista dei moduli importati, ottenuta nello step 1, viene sottratta la lista di moduli effettivamente utilizzati. Il risultato di tale differenza è la lista di moduli non utilizzati, i cui nomi vengono segnalati all'utente.

- per ogni modulo M_1 effettivamente utilizzato viene stampato il percorso assoluto.

- *clash_search*: restituisce una lista di elementi del formato $nomePredicato/arieta - [mod_1, mod_2, \ldots, mod_n]$ che rappresenta i predicati associati ad una lista di moduli all'interno dei quali sono state individuate delle dichiarazioni omografe e con pari arietà e quindi potenziali fonti di name clash. La ricerca dei clash viene fatta fra i predicati esportati dal modulo in analisi e quelli esportati dai moduli importati dal modulo in analisi.

Capitolo 4

Descrizione del sistema

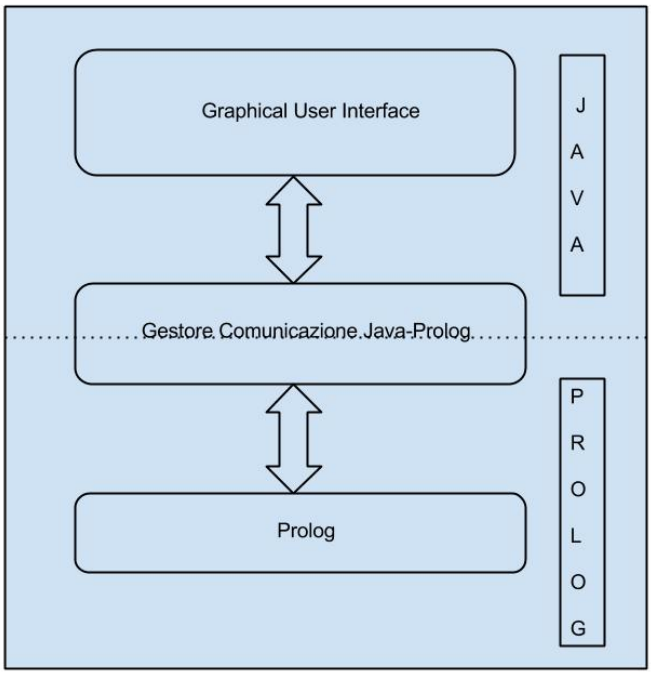

Figura 4.1: Architettura di DepJaeger

DepJaeger esibisce un'interfaccia grafica in grado di gestire le interazioni con il nucleo del programma (scritto in Prolog), liberando quindi l'utente dall'interazione col terminale, nonchè di presentare le informazioni restituite

in maniera più leggibile e più comodamente fruibile. A parte l'evidente comodità d'uso, il sistema rende la fruizione anche intuitiva e scevra da errori dovuti ad errori di tipografia o di sequenza delle operazioni effettuate. Il gestore dell'interfaccia attuerà dei controlli tali da permettere l'attivazione di varie funzioni solo quando le azioni richieste per farle funzionare sono state compiute, e guiderà l'utente verso la loro esecuzione. Gli elementi principali dell'interfaccia, con i relativi pulsanti, la cui funzione e uso verranno descritti in dettaglio nel prossimo paragrafo, sono i seguenti (nella figura 4.2, i numeri delle aree corrispondono alle rispettive funzioni assegnate enumerate nell'elenco seguente):

1. Sezione di caricamento dei file
2. Sezione di analisi dei moduli
3. Sezione di analisi dei predicati
4. Sezione di output finale

4.1 Sezioni ed operazioni disponibili

Le operazioni che è possibile effettuare nel sistema sono descritte di seguito.

4.1.1 Caricamento di file

Premendo il pulsante "add files" verrà visualizzata una finestra di selezione di file, tramite la quale sarà possibile caricare file singoli, un loro insieme o intere cartelle. In ogni caso, gli unici file selezionabili sono quelli con estensioni Prolog (*.pl,*.yap). Poichè il sistema di comunicazione basato su Interprolog non è in grado di gestire più di 14 file alla volta, è stato posto un controllo sul numero di file analizzabili di volta in volta. L'elenco completo dei percorsi assoluti dei file caricati verrà visualizzato nella finestra di fianco al bottone utilizzato. Una volta caricati i file, i nomi dei moduli in essi contenuti verranno inseriti nella lista presente nell'area di gestione dei moduli.

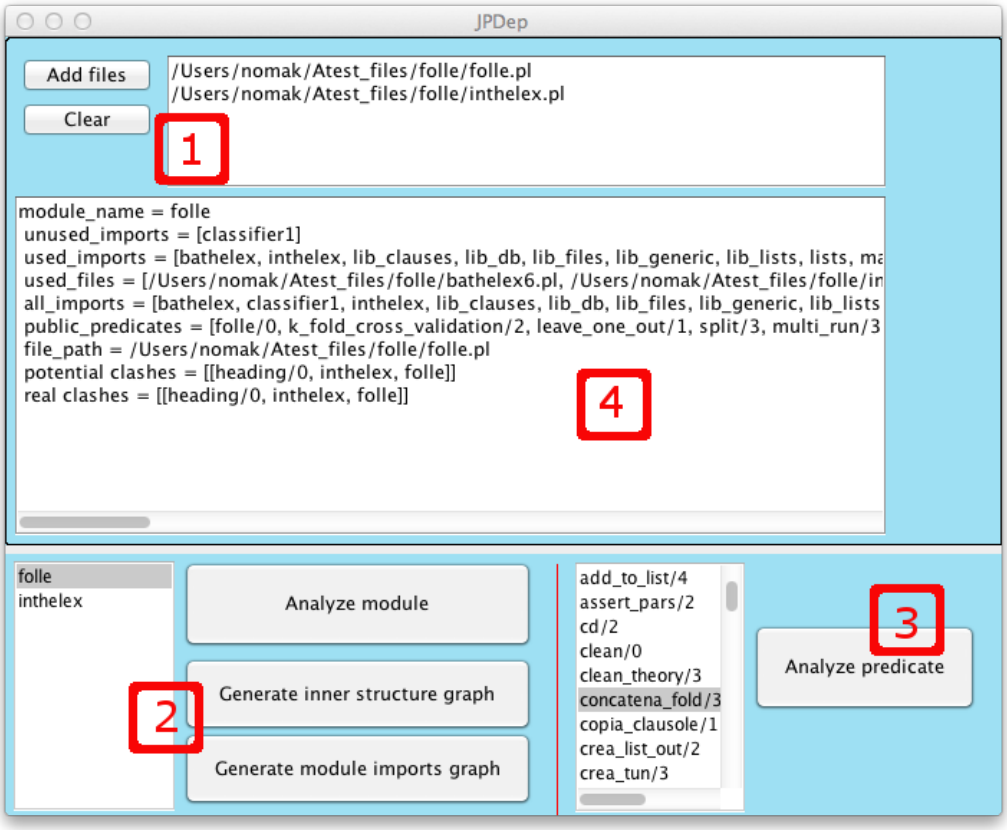

Figura 4.2: Funzioni delle aree dell'interfaccia

4.1.2 Analisi dei moduli

Per effettuare l'analisi delle dipendenze dei moduli, è necessario aver inserito almeno un file contenente codice Prolog. Dopo aver soddisfatto questa condizione, verrà abilitato il pulsante "Analyze module", la cui pressione invierà una richiesta all'analizzatore, che restituirà le seguenti informazioni:

- Percorso assoluto del file analizzato
- Lista delle dipendenze individuate
- Lista di moduli importati ma non utilizzati dal modulo
- Lista di moduli importati ed utilizzati
- Lista dei moduli importati ed utilizzati con i relativi percorsi assoluti

- Eventuali name clashes individuati

4.1.3 Generazione dei grafi delle dipendenze

Dopo aver caricato almeno un file e selezionato un modulo dalla lista, per esso potranno essere generate delle rappresentazione grafica delle dipendenze, consistenti di grafi contenuti in un'immagine in formato PNG che verrà memorizzata sull'hard disk, in una posizione scelta dall'utente. I due grafi che è possibile generare sono uno relativo alla struttura interna del modulo e alle interazioni fra i predicati al suo interno, mentre l'altro mostra i moduli importati.

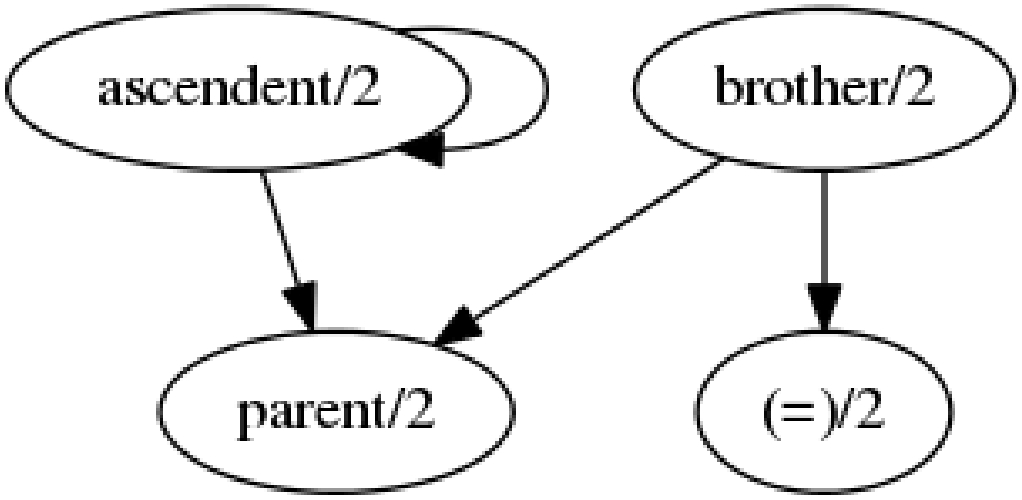

Figura 4.3: Esempio di grafo della struttura interna del modulo

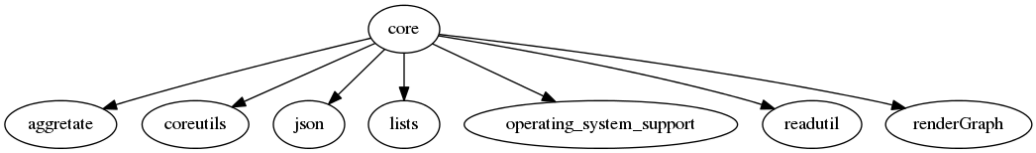

Figura 4.4: Esempio di grafo delle importazioni

4.1.4 Analisi dei predicati

E' disponibile una funzione di analisi dei predicati, che permette di ottenere le seguenti informazioni relative ad un dato predicato fornito in input:

- nome del predicato
- arietà del predicato
- modulo all'interno del quale è dichiarato il predicato
- file all'interno del quale è dichiarato il predicato
- predicati utilizzati dal predicato
- predicati che utilizzano il predicato
- moduli utilizzati dal predicato

4.1.5 Reset delle condizioni iniziali

Tramite il pulsante "Clear" si ripristinano le condizioni iniziali del sistema. Viene ripristinato lo stato iniziale sia dell'interfaccia che del modulo Prolog, la cui working memory viene liberata dei fatti appresi durante l'esecuzione. I grafi precedentemente generati, invece, permangono nell'hard disk dell'utente.

4.2 Interazione con l'utente

L'interazione avviene principalemente tramite l'interfaccia grafica descritta nelle sezioni seguenti. La comunicazione fra il lato Java e quello Prolog sono riportate nella finestra di terminale da cui si è lanciato l'eseguibile. In alternativa, è possibile utilizzare il sistema direttamente dalla linea di comando dell'interprete Prolog, caricando il file *core.pl*. Le funzionalità offerte rimangono invariate. Utilizzare il predicato *help/0* per informazioni relative all'utilizzo da riga di comando.

4.2.1 Protocollo di comunicazione Java-Prolog

Per consentire la comunicazione fra Java e Prolog è stato progettato ed implementato un protocollo di comunicazione che fa uso della libreria Interprolog e di JSON, un protocollo per la serializzazione di oggetti e lo scambio di dati. La scelta di progettare da zero un protocollo di comunicazione è stata dettata dalla cospicua inefficienza delle librerie correntemente utilizzate: JPL è risultata inadatta agli scopi al contrario di Interprolog, che però viene utilizzato solo per inviare i comandi al core del sistema scritto in Prolog, che ritornerà a sua volta le informazioni desiderate sotto forma di oggetto serializzato in JSON.
La scelta di utilizzare JSON è motivata dal fatto che Interprolog fornisce una rappresentazione dell'output dei goal difficile da gestire. Combinando la facilità di uso di Interprolog con la versatilità e la praticità di JSON si riesce ad attuare una comunicazione efficace e stabile fra le due parti.

Di seguito viene riportata la sequenza dei passi rappresentativa dell'interazione fra i due linguaggi:

- Java invia una richiesta, tramite Interprolog, al core Prolog. (es. *predicates(nomeModulo, X)*).
- Prolog elabora la richiesta, creando un file temporaneo nel quale andrà a riporre l'output (in formato JSON) e unifica il path del file con X.
- Java legge il contenuto della variabile X contenente il path del file, tramite Interprolog, e ne deserializza il contenuto. A seconda della richiesta formulata il file verrà deserializzato come un oggetto appartenente alle classi Predicate, Module e GenericOutput.

Nello specifico, il protocollo interagisce con i seguenti predicati:

- *predicate_json_to_file(+Modulo:+Predicato, -File)*: fornisce le informazioni riguardanti il Predicato oggetto dell'analisi, restituendole in un file contenente l'oggetto in formato JSON

- *module_json_to_file(+Modulo, -File)*: fornisce le informazioni riguardanti il Modulo oggetto dell'analisi, restituendole nella stessa modalità del predicato precedentemente illustrato
- *predicates(json, +Modulo, -File)*: restituisce in un File contenente un oggetto JSON la lista dei predicati pubblici e privati dichiarati nel Modulo in input
- *modules(json, -JsonFile)*: restituisce in un File contenente un oggetto JSON la lista dei moduli correntemente caricati nel sistema
- *module_2_png(+Module, +FilePath)*: genera la rappresentazione grafica della struttura interna del Modulo in input, memorizzandola nel FilePath indicato
- *moduleInteraction2png(+Module, +FilePath)*: genera la rappresentazione grafica delle interazioni fra i moduli

La trasformazione delle informazioni in JSON lato Prolog è molto semplice. Allo scopo viene utilizzata la libreria 'yap-json' che permette, a partire da un fatto $json([attributo_1 = valore_1, attributo_2 = [lista\_di\_valori], \ldots])$, di ottenere un oggetto serializzato in JSON. Di seguito è riportato un esempio di trasformazione di un termine in JSON e viceversa.

```
?- Term = json([name-alessandro,surname-natilla]),
   term2json(Term, Json).
Json = '{"name":"alessandro","surname":"natilla"}',
Term = json([name-alessandro,surname-natilla])

?- Json = '{"name":"alessandro","surname":"natilla"}',
   json2term(Json, Term).
Term = json([name-alessandro,surname-natilla]),
Json = '{"name":"alessandro","surname":"natilla"}'.
```

Figura 4.5: Modello del protocollo Yap to JSON

4.2.2 Classi Java

La classe *Predicate* permette di raccogliere informazioni riguardo ad un predicato, nello specifico:

1. Nome del predicato
2. Arietà
3. Numero di clausole
4. Modulo in cui è dichiarato
5. File in cui è contenuto
6. Lista di predicati che utilizza
7. Lista dei predicati da cui esso è utilizzato
8. Moduli contenenti i predicati della parte destra
9. Status del predicato (pubblico/non pubblico)

La classe *Module* incapsula tutte le informazioni riguardo un modulo, nello specifico:

1. Nome del modulo
2. Lista dei moduli importati e utilizzati
3. Lista dei moduli importati e non utilizzati
4. Lista complessiva dei moduli importati
5. Lista di file path ove i moduli utilizzati risiedono
6. Lista dei name clash potenziali individuati
7. Lista dei name clash effettivi individuati
8. Lista dei predicati pubblici che esso espone

9. File in cui il modulo risiede

Infine, la classe GenericOutput è utile per scambiare informazioni generiche, non ricadenti in nessuna delle precedenti categorie (esempio: la lista dei predicati contenuti in un modulo).

Di seguito si riporta il diagramma delle classi coinvolte nella comunicazione, in formato UML:

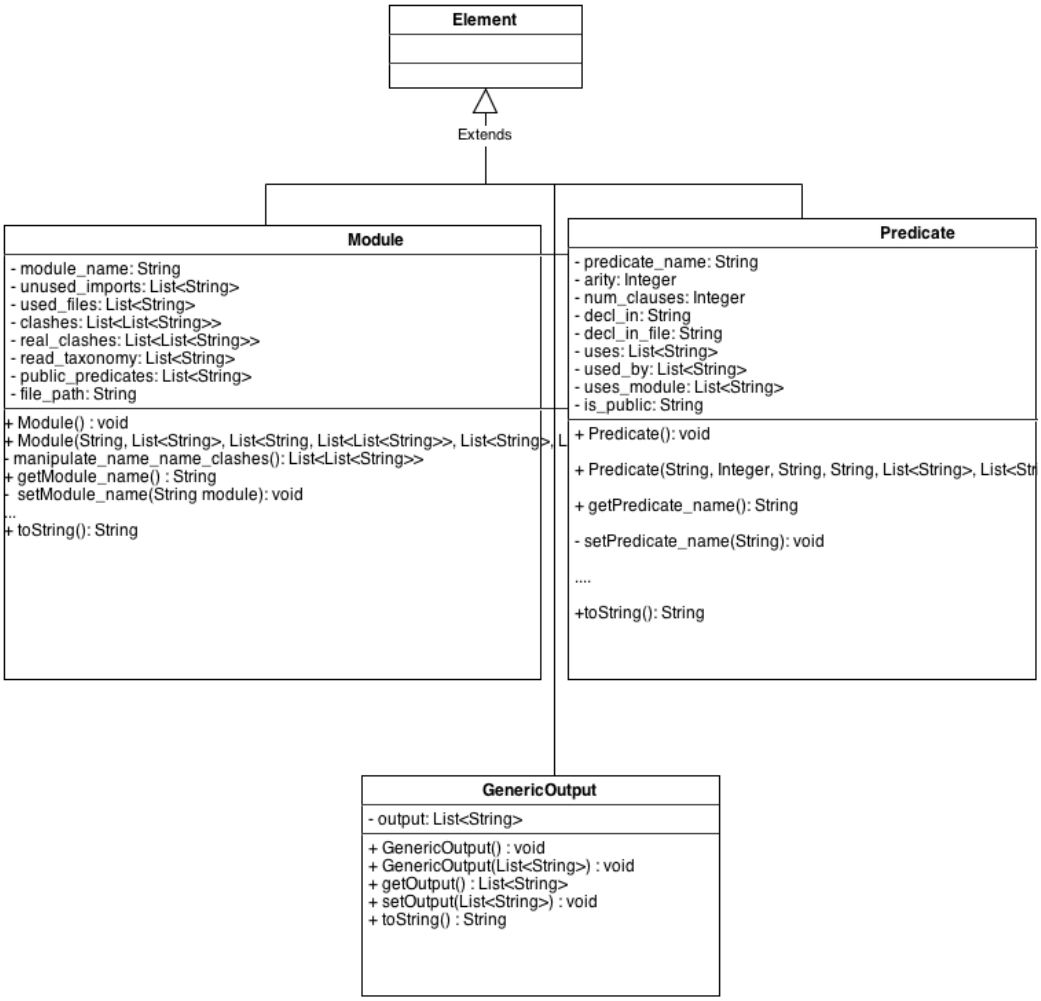

Figura 4.6: Diagramma UML delle classi

4.3 Funzionamento di DepJaeger

Subito dopo l'avvio, il sistema si presenta nella seguente maniera: come

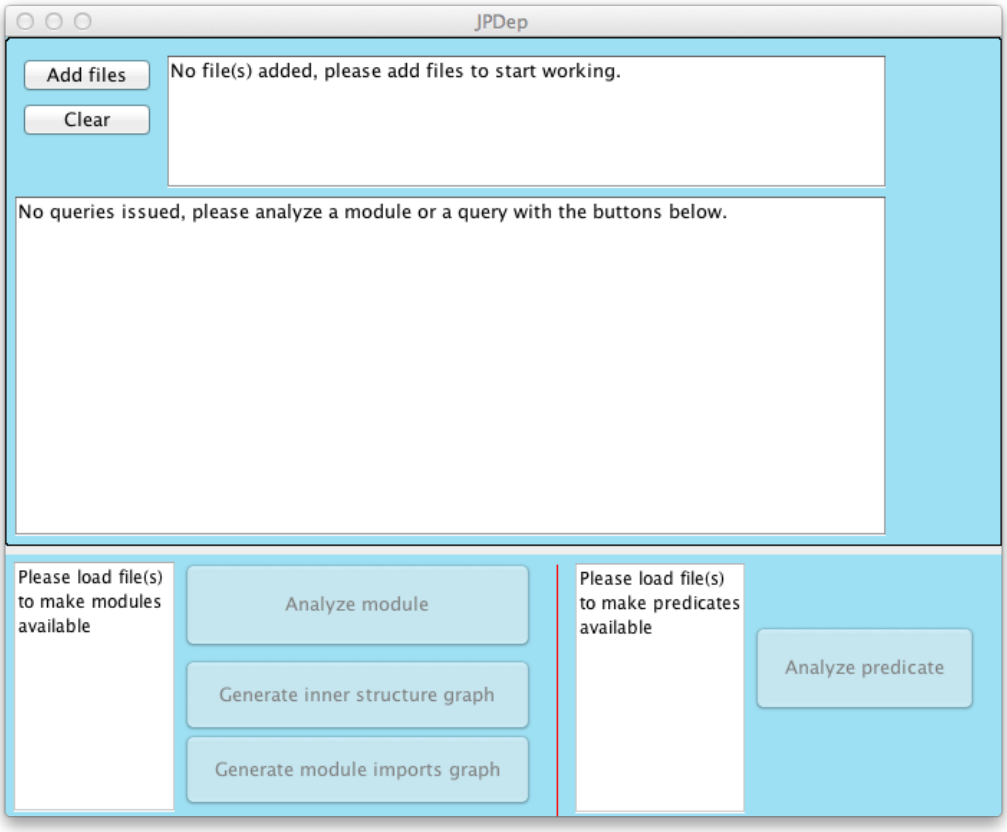

Figura 4.7: Stato iniziale del sistema

si vede, i pulsanti sulla parte inferiore relativi all'esecuzione dell'analisi sui predicati e sui moduli sono disattivati, e in ciascuna delle caselle di testo sono visibili le istruzioni per far progredire il sistema verso uno stato più "operativo", e le uniche operazioni che è possibile fare sono la pulizia del sistema (che non avrà alcun effetto percepibile) e l'aggiunta di file.

4.3.1 Caricamento di file

Il caricamento avviene tramite un file browser standard che apparirà alla pressione del tasto "Add files", ed è possibile scegliere uno o più file, o un'in-

tera cartella (e relative sottocartelle), sempre nei limiti dei 14 file massimo che Interprolog è in grado di gestire.

Dopo aver aggiunto i file che si desidera analizzare, lo stato del sistema muterà nel seguente: Come si vede, vengono ora visualizzati i nomi e i per-

Figura 4.8: Stato dopo il caricamento dei file

corsi del file nell'area di gestione dei file, mentre nella parte in basso sono visibili i moduli e i predicati rilevati nei suddetti file. Tali predicati sono presentati nella forma *nomePredicato/arietà*, come da standard Prolog. Le liste in cui i nomi sono collocati sono interattive, e di default viene sempre selezionato il primo modulo della lista. Sono ora inoltre attivi i pulsanti per l'esecuzione delle query e per la generazione dei grafi.
Nella casella centrale vi è ancora un testo di guida per l'utente, che invita ad effettuare un'analisi di un modulo o di un predicato.

4.3.2 Analisi di un modulo

L'analisi del modulo si effettua tramite l'apposito pulsante "Analyze module". Vediamo ora come si modifica lo stato del sistema dopo l'analisi: nella casella

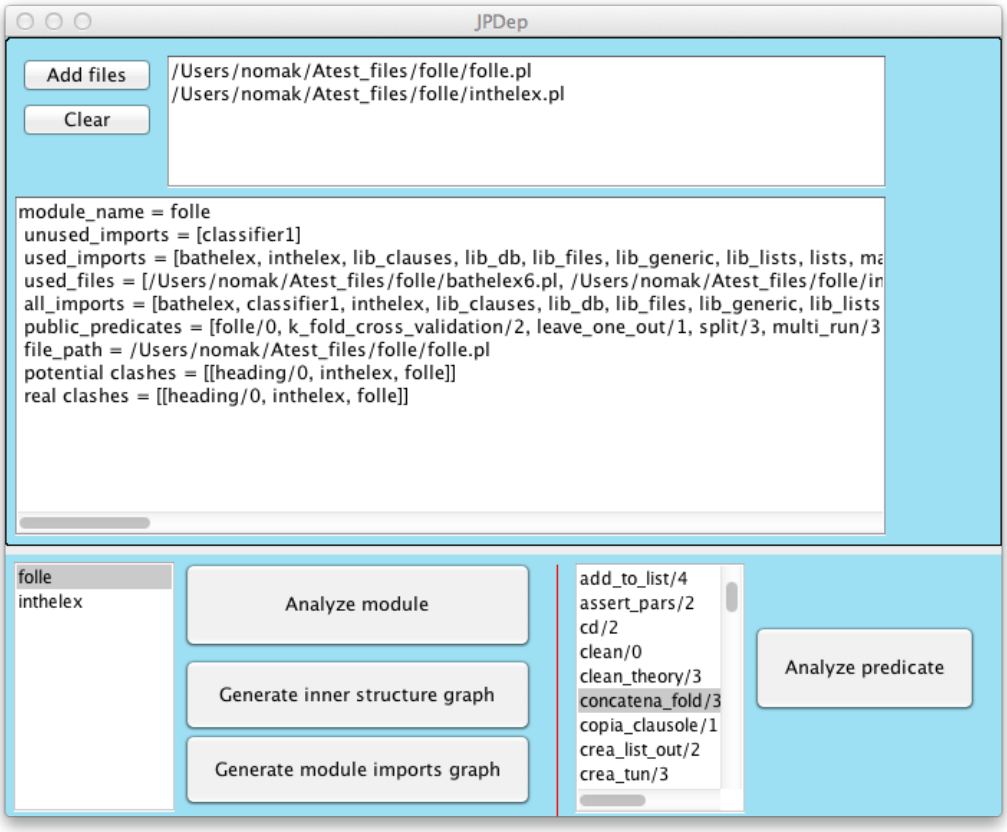

Figura 4.9: Stato dopo l'analisi di un modulo

centrale è ora visibile l'output dell'analisi eseguita lato Prolog descritta nei paragrafi precedenti: viene riportato per prima cosa il nome "interno" del modulo, seguito da il nome dei moduli che vengono importati ma non usati. Tale informazione è molto importante per le finalità del sistema, permettendo di rendersi conto dei moduli di cui è possibile fare a meno.
Seguono le liste dei moduli e dei file effettivamente utilizzati, e che quindi non possono essere rimossi. Viene poi indicata la lista completa dei moduli importati dal modulo esaminato, i predicati che nel modulo in esame sono dichiarati come pubblici e il percorso del file che contiene il modulo. Come

ultimo, importante dettaglio, sono segnalati i potenziali name clashes fra predicati aventi stesso nome ed arietà, ma definiti in moduli diversi, che vanno quindi a sovrapporsi e potenzialmente a generare ambiguità durante l'esecuzione, forzando la ridefinizione di uno dei due e oscurando, di fatto, l'altro. Subito dopo sono indicati i name clash effetivi. Nello specifico, sono presentate due liste di triple nel formato $[nomePredicato/arieta', nomeModulo_1, nomeModulo_2, \ldots, nomeModulo_n]$.

4.3.3 Analisi di un predicato

Similmente a quanto detto nel precedente paragrafo, l'analisi del predicato si effettua con il pulsante "Analyze predicate".
Di seguito vediamo lo stato del sistema dopo un'analisi su un predicato. Nell'area di testo centrale sono visibili in forma esplicita il nome del predicato analizzato e l'arietà secondo lo standard Prolog. Viene poi indicato se il predicato sia pubblico o no, seguito dal nome del modulo in cui il predicato è dichiarato, e il nome del file in cui questo si trova. Di seguito ci sono le liste in cui vengono indicati, nel consueto formato *nomePredicato/arietà*, i predicati effettivamente usati dal predicato analizzato e quelli da cui quest'ultimo è effettivamente usato. Infine, vengono elencati i moduli esterni in cui sono dichiarati i predicati usati dal predicato analizzato, che sono quindi effettivamente necessari.

Figura 4.10: Stato dopo l'analisi di un predicato

Capitolo 5

Conclusioni e sviluppi futuri

Allo stato attuale, DepJaeger è stato capace di analizzare correttamente sorgenti Prolog complessi.
Le informazioni fornite sono utili in contesti quali la reingegnerizzazione e il refactoring. Gli unici limiti attualmente presenti sono legati a carenze della libreria Interprolog, usata nell'ambito del protocollo di comunicazione. Per aggirare tali limiti è possibile utilizzare DepJaeger da linea di comando, come indicato nella sezione 4.2. Inoltre, si potrebbe valutare la possibilità di implementare un protocollo di comunicazione basato su socket e JSON, facendo quindi a meno di Interprolog.
Eventuali sviluppi futuri potrebbero riguardare una riduzione dei tempi di completamento dei task.

Bibliografia

[1] L. Console, E. Lamma, P. Mello, M. Milano - Programmazione Logica e Prolog, Nuova ed. - UTET Libreria

[2] Documentazione di YAP, `http://www.dcc.fc.up.pt/~vsc/Yap/documentation.html`

[3] Documentazione di InterProlog, `http://www.declarativa.com/interprolog/htmldocs/overview-summary.html`

[4] Yap to JSON, `http://github/anatilla/yap-json`