UNIVERSITÀ
DI PARMA

Dipartimento di Scienze
Matematiche, Fisiche e Informatiche

Corso di Laurea in Informatica

Generazione automatica ed esecuzione di casi di test per la libreria Java JSetL

Automatic generation and execution of test cases for the Java library JSetL

Relatore:
Prof. Gianfranco Rossi

Candidato:
Francesco Vetere

Anno Accademico 2018/2019

Ai miei genitori, Giusy e Domenico.
A mia sorella, Giulia.

Indice

Introduzione

Nell'ambito dell'Ingegneria del Software, il termine *testing* denota il processo di analisi di un sistema software avente lo scopo di individuare errori logici a run-time e di verificare la correttezza del sistema rispetto alle specifiche iniziali concordate con il committente.
Tale processo si colloca all'interno del ciclo di vita del software, e ne costituisce una fase fondamentale: un testing ben strutturato permette quasi sempre un beneficio non trascurabile in termini di risparmio di tempo in fase manutentiva, e di conseguenza anche in termini di spese sostenute.
Avere strumenti in grado di automatizzare questo processo è spesso una scelta vincente, sia per quanto concerne l'affidabilità dei test generati, sia in termini di rapidità con cui il processo si svolge.
Scopo di questo lavoro di tesi è quello di realizzare uno strumento software atto ad automatizzare parte del processo di testing per la libreria Java JSetL.
Sebbene siano molti gli strumenti ad oggi disponibili per effettuare testing automatico (si veda a tal proposito [5]), in questo lavoro di tesi si è scelto di realizzare uno strumento ex novo, specifico per la libreria JSetL.

JSetL è una libreria Java sviluppata presso il Dipartimento di Matematica e Informatica dell'Università di Parma.
La libreria nasce con l'obiettivo di combinare il paradigma object-oriented di Java con alcuni dei concetti classici del paradigma logico a vincoli, tra i quali variabili logiche, unificazione, risoluzione di vincoli e non determinismo (tutti i dettagli sono reperibili in [1] e [2]).
Lo strumento software realizzato in questo lavoro di tesi si focalizza sul testing dei vincoli offerti dalla libreria JSetL, analizzandone sia aspetti di correttezza semantica, sia aspetti di efficienza.
L'approccio utilizzato per la realizzazione dello strumento si basa sulla creazione di due generatori automatici di casi di test, i quali forniscono in output una casistica completa di test sulla base di precise specifiche di input inserite dall'utente.

L'elaborato di tesi è così strutturato:

- **Capitolo 1**: In questo capitolo verrà brevemente illustrato il concetto di testing, classificandone le varie tipologie, individuandone le diverse fasi che ne compongono il processo, ed infine analizzando pro e contro del testing manuale e di quello automatizzato (si veda a tal proposito [3]).
- **Capitolo 2**: In questo capitolo verrà presentato l'ambiente in cui si colloca lo strumento software realizzato: la libreria JSetL, il linguaggio CLP($\mathcal{SET}$)/L$_{\mathcal{BR}}$, l'interprete {log} ed infine il framework JUnit (per approfondimenti, si rimanda a [4]).
- **Capitolo 3**: In questo capitolo verrà presentata l'architettura dello strumento software realizzato per questo lavoro di tesi, includendo le motivazioni che hanno portato al suo sviluppo.
- **Capitolo 4**: In questo capitolo verrà analizzato nel dettaglio lo strumento software realizzato, descrivendo sintassi e semantica dei diversi file di input/output e degli script di compilazione. Verranno infine presentati i due programmi generatori scritti in linguaggio C++.
- **Capitolo 5**: In questo capitolo verranno analizzati 12 test set per JSetL prodotti tramite lo strumento realizzato, fornendo una dimostrazione pratica di utilizzo dello stesso.
  Verranno infine fatte alcune analisi sui tempi di esecuzione rilevati da un campione di 5 test set.
- **Capitolo 6**: In quest'ultimo capitolo verranno tratte alcune conclusioni sul lavoro di tesi, analizzando i punti di forza dello strumento realizzato ed in particolare in che modo lo strumento potrà fornire supporto al continuo sviluppo della libreria JSetL.
  Infine, verrà lasciato spazio ad alcuni spunti di rilievo in merito ad eventuali lavori futuri e miglioramenti apportabili allo strumento realizzato.

1 Il testing

Nell'ambito dell'Ingegneria del Software, si intende con il termine *testing* il processo atto ad individuare errori logici a run-time e a verificare la correttezza del sistema rispetto alle specifiche iniziali concordate con il committente.
In particolare:

- Per **errore logico** (noto come *bug*) si intende una sequenza di istruzioni che, quando eseguita in determinate circostanze, porta a malfunzionamenti di vario tipo nel programma (es.: risultati inattesi o errati).
  Tipicamente si tratta di errori semantici, ossia errori commessi dal programmatore in fase di progettazione dell'algoritmo a causa di una comprensione non adeguata del problema.
  La difficoltà nell'individuare questo tipo di errori spesso risiede nel fatto che essi tipicamente sono visibili solo a run-time: non è quindi possibile stabilire con anticipo quale sarà il comportamento del programma se non eseguendolo. Errori più banali sono ad esempio quelli sintattici, individuabili già a compile-time spesso anche da un semplice editor di testo.

- Per **correttezza del sistema** si intende la congruenza che deve sussistere tra il comportamento del sistema ed i requisiti iniziali definiti dal committente: lo sviluppatore verifica, insieme al committente, che il sistema abbia tutte le caratteristiche e le qualità stabilite nelle specifiche iniziali.
  Si parla in questo caso di test di *validazione*, una fase del processo di testing che deve necessariamente essere svolta in un momento successivo all'individuazione degli errori logici: tipicamente infatti, la validazione avviene in fase conclusiva di sviluppo.

Il testing fa parte del ciclo di vita del software, ed è una delle sue fasi più importanti: se realizzato in maniera adeguata infatti, permette spesso un beneficio non trascurabile in termini di tempo e spese sostenute.
Il problema principale del testing è rappresentato dal noto *Teorema di Dijkstra (1969)*:

> Il test di un sistema software può rilevare la presenza di malfunzionamenti, ma mai dimostrarne l'assenza.

L'idea è quindi quella di verificare il comportamento del sistema su un insieme di casi sufficientemente ampio da rendere plausibile che il suo comportamento sia analogo anche nei restanti casi.

1.1 Testing In The Large vs. Testing In The Small

L'attività di testing si può dividere in due macro-categorie, che si distinguono in base all'approccio utilizzato per condurre i test.

- **Testing in the large** (*black-box* testing)
  Utilizzando l'approccio testing in the large, noto come black-box testing, si osserva il comportamento del sistema in relazione a determinati input, e si stima quanto l'output che ne risulta è distante dall'output effettivamente desiderato.
  Questo permette a chi effettua il testing di non dover necessariamente essere a conoscenza dei dettagli implementativi del sistema software, in quanto gli unici fatti di rilievo nel test sono le corrispondenze tra input e output.
  Vi sono due tipologie principali di testing in the large, distinte in base a come viene istanziato il cosiddetto *test vector* (il vettore contenente l'insieme degli input forniti ad un sistema per testarlo):

- **Boundary Value Analysis**: in questa tipologia di black-box testing, il test vector è formato da valori limite per i domini considerati.
  Questo permette un'analisi in casi particolari e delicati per il sistema, ma non è ovviamente un'analisi completa.

- **Equivalence Class Partitioning**: in questa seconda tipologia di black-box testing, gli input possibili per il sistema vengono divisi per classi di equivalenza, ed infine si assembla un test vector scegliendo da ognuna di queste classi di equivalenza un valore campione.
  È un ottimo metodo per evitare ridondanza negli input ed ottenere un test il più possibile esaustivo.
  Resta da definire come accoppiare i vari parametri di input per formare dei test vector significativi: vi sono appositi criteri, detti criteri *IPS* (Input Space Partitioning), che definiscono appunto come scegliere tali parametri.
  Ad esempio:

  - **Each Choice**: questo criterio stabilisce che ogni valore di una classe di equivalenza dovrà comparire in almeno un caso di test;
  - **All Combinations**: tutti i valori di input possibili per una classe di equivalenza vengono combinati con tutti i valori possibili per le restanti classi.

La scelta di un test ideale per un programma non è quindi un problema banale, come evidenziato dal *Teorema di Howden (1975)*:

> Non esiste un algoritmo che, dato un programma qualsiasi, generi per esso un test ideale (definito cioè da un criterio affidabile e valido).

- **Testing in the small** (*white-box* testing)
  L'approccio opposto è invece quello del testing in the small, in cui l'obiettivo diventa la verifica del funzionamento interno di un componente software, piuttosto che la verifica delle sue funzionalità (non a caso black-box testing e white-box testing sono anche noti, rispettivamente, come testing funzionale e testing strutturale).
  Questa tipologia di test è spesso a carico del programmatore, in quanto può essere condotta solamente conoscendo la struttura interna della porzione di codice da testare.
  Si distinguono 3 tipi di white-box testing, noti anche come test di *coverage*:

  - **Statement testing**: questo tipo di test ha l'obiettivo di testare ogni statement presente nella porzione di codice interessata (un errore infatti non può essere individuato se la parte di codice che lo contiene non viene eseguita almeno una volta).
    Il criterio utilizzato consiste nel selezionare un insieme di test T tale che, a seguito dell'esecuzione del programma P su tutti i casi di T, ogni istruzione elementare di P venga eseguita almeno una volta.

  - **Branch testing**: in questo tipo di test si mira ad ottenere una copertura di tutti i possibili path generati dai branch presenti nel grafo di controllo del programma.
    Il criterio utilizzato consiste nel selezionare un insieme di test T tale che, a seguito dell'esecuzione del programma P su tutti i casi di T, ogni arco del grafo di controllo di P sia attraversato almeno una volta.

  - **Branch and condition testing**: questo tipo di test è ancora più approfondito rispetto al precedente.
    L'obiettivo è sempre quello di ottenere una copertura di tutti i possibili path generati dai branch, ma in aggiunta si richiede anche la valutazio-

ne dei risultati delle condizioni composte.
Il criterio utilizzato consiste nel selezionare un insieme di test T tale che, a seguito dell'esecuzione del programma P su tutti i casi di T, ogni arco del grafo di controllo di P sia attraversato e tutti i possibili valori delle condizioni composte siano valutati almeno una volta.

1.2 Fasi del processo di testing

Il processo di testing si articola tipicamente in 5 fasi, di seguito descritte.

- **Unit testing**
  In questa prima fase, si effettua il testing di ogni singolo modulo software visto come entità indipendente dal resto del sistema.
  Per simulare la presenza di componenti esterni che possono a loro volta inviare/richiedere servizi, si ricorre talvolta al *mock* dei dati utilizzati, simulando quindi ipotetiche interazioni con l'esterno al fine di potersi concentrare completamente sul singolo modulo.
  Esistono diversi tool che permettono di fare unit testing in maniera efficace: i più noti sono quelli della famiglia xUnit, in cui "x" è un riferimento al linguaggio di programmazione per il quale è pensato il tool (per Java ad esempio, il framework prende il nome di JUnit).

- **Integration testing**
  La fase di integration testing rileva eventuali bug che non sono stati individuati durante la fase di unit testing, spesso a causa delle problematiche di interfacciamento tra i vari moduli che nello unit testing vengono completamente ignorate.
  Esistono due tipiche strategie che guidano l'integrazione dei moduli software:

- **Bottom-Up testing**
  Questa metodologia prevede che vengano testate prima le unità più piccole del sistema; una volta che il loro testing ha successo, si passa alla loro integrazione, e si procede allo stesso modo fino ad aver integrato tutti i sotto-moduli in un'unica architettura software globale.
  L'approccio è sicuramente semplice e graduale, ma il rischio concreto è quello di rilevare tardi errori dovuti all'integrazione fra i moduli, e ciò non è affatto banale: nel testing, come regola generale, gli errori devono essere rilevati quanto prima possibile.

- **Top-Down testing**
  Questo approccio è diametralmente opposto al precedente: il sistema viene inizialmente testato nella sua interezza, simulando i sotto-componenti tramite degli *stub*, elementi che forniscono dati spesso pseudo-casuali al fine di permettere un testing globale pur in assenza delle unità di base. Questa metodologia è più onerosa sotto tutti i punti di vista, ma ha un grosso vantaggio: gli errori di integrazione vengono rilevati notevolmente in anticipo rispetto ad un approccio bottom-up.

- **System testing**
  In questa fase il sistema viene testato e collaudato in relazione a specifici parametri di qualità. Esempi di test eseguiti in questa fase sono:

  - **Stress test**: test atto a verificare le proprietà del sistema in condizioni di sovraccarico;
  - **Robustness test**: test atto a verificare le proprietà del sistema in presenza di dati non corretti;
  - **Security test**: test atto a verificare le proprietà di sicurezza del sistema.

- **Acceptance testing**
  In questa fase del testing, nota anche come fase di validazione, le proprietà del sistema software vengono comparate con i requisiti iniziali dettati dal committente (spesso, è proprio il cliente finale a testare il software in questa fase).
  Vi sono usualmente due prototipi realizzati per un prodotto software:
  - L'**alpha test** è un primo prototipo del sistema utilizzato all'interno dell'azienda produttrice del software, al fine di rilevare ulteriori bug non evidenziati dalle fasi precedenti prima del rilascio ai clienti finali.
  - Il **beta test** è invece il test con il committente, in cui il sistema viene testato nel suo ambiente dai clienti finali, i quali avranno la possibilità di evidenziare eventuali discrepanze con quanto inizialmente stabilito nei requisiti.

- **Installation testing**
  In questa ultima fase del processo di testing, il software è ormai stato accettato formalmente dal cliente, e si procede quindi all'installazione ed alla messa in esercizio.
  È dunque una verifica finale del comportamento del sistema in un contesto reale, con dati non più fittizi ma presi direttamente dall'ambiente in cui opera il cliente.

1.3 Manual Testing vs. Automated Testing

Un ulteriore aspetto di fondamentale importanza nel processo di testing consiste nello scegliere se i test dovranno essere sviluppati manualmente o tramite l'ausilio di un software apposito.

- Nel **testing manuale**, i test cases vengono scritti direttamente dal programmatore.
  Il rischio di ottenere imprecisioni dovute all'errore umano è sicuramente alto, così come lo è il rischio di non ottenere una copertura efficace a causa di alcune casistiche non trattate dai test case prodotti.
  Oltre a ciò, non è da sottovalutare anche il fatto che questo approccio richiede l'impiego di una risorsa umana per diverso tempo.
  In contesti molto ampi, questa strategia si rivela spesso impraticabile.

- Nel **testing automatizzato**, i test cases vengono generati in automatico da un software.
  Il primo vantaggio tangibile è l'affidabilità: l'errore umano è eliminato del tutto o quasi, e la copertura dei test è garantita in fase di progettazione del software.
  Uno strumento automatico è inoltre molto più rapido nel generare test cases: l'utente deve semplicemente fornire in input un file (scritto tramite un'opportuna sintassi) in cui esplicita quali parti del sistema software in oggetto è interessato a testare.
  Sicuramente lo svantaggio principale di questo approccio è il costo iniziale per lo sviluppo del tool, ma sul lungo termine il risparmio di risorse umane, tempo, spese sostenute ed errori non è trascurabile.
  Esistono in commercio diversi tool per la generazione automatica dei test cases, tra i quali citiamo *Tobias*, *JPet* ed *EvoSuite*.

2 Descrizione dell'ambiente

In questo capitolo viene presentato l'ambiente in cui opera lo strumento software realizzato per questo lavoro di tesi.

2.1 JSetL

2.1.1 Principali features

JSetL è una libreria Java sviluppata presso il Dipartimento di Matematica e Informatica dell'Università di Parma.
Il suo scopo è quello di combinare il paradigma object-oriented di Java con alcuni concetti classici del paradigma logico a vincoli, tra i quali:

- **Variabili logiche**
  Sono le variabili tipiche del paradigma logico-funzionale.
  Possono essere inizializzate o non inizializzate, e il loro valore può potenzialmente essere di ogni tipo ammesso dal linguaggio: può in particolare essere determinato come risultato della risoluzione di vincoli che coinvolgono le variabili stesse.

- **Unificazione**
  Il meccanismo dell'unificazione è fondamentale in questo paradigma di programmazione, e permette sia di testare l'uguaglianza tra oggetti, sia di assegnare valori alle variabili logiche non inizializzate eventualmente contenute al loro interno nel tentativo di rendere uguali i due termini del confronto.

- **Risoluzione di vincoli**
  JSetL fornisce una serie di vincoli di base quali uguaglianza, disuguaglianza, operazioni insiemistiche di base come differenza, unione, intersezione e molte altre.
  Un vincolo viene prima aggiunto allo store dei vincoli (*constraint store*), e successivamente viene risolto tramite un risolutore (*solver*) attraverso regole di riscrittura sintattica.
  JSetL permette inoltre la creazione di vincoli composti (tramite congiunzione, disgiunzione e implicazione) e vincoli definiti da utente.

- **Non determinismo**
  I vincoli in JSetL vengono risolti in maniera non deterministica: sia perché l'ordine di risoluzione degli stessi non è rilevante, sia perché nel processo di risoluzione si fa ampio di uso di choice-points e di backtracking.

Inoltre, JSetL fornisce **insiemi** e **liste** come strutture dati fondamentali, su cui è possibile operare tramite una serie di vincoli, in particolare vincoli insiemistici che implementano le classiche operazioni su insiemi, quali unione, intersezione, appartenenza, ecc...
La principale differenza tra liste e insiemi risiede nel fatto che nelle liste sono rilevanti ordine e ripetizioni degli elementi, negli insiemi no.
Entrambe le strutture dati permettono di collezionare elementi di tipo generico: in particolare, esse possono essere parzialmente specificate (ovvero possono contenere variabili logiche non inizializzate, aspetto molto importante in JSetL).

Vengono di seguito brevemente analizzate le classi principali che compongono la libreria e che compariranno di frequente in questa tesi.

2.1.2 Classi fondamentali

LVar

La classe `LVar` implementa il concetto di variabile logica descritto in precedenza. Oggetti di classe `LVar` possono essere creati associandovi un valore (*bound*), lasciandoli non inizializzati (*unbound*), e possono o meno avere un nome identificativo associato.
Esempi:

```
// Creazione di una LVar unbound senza nome associato
LVar A = new LVar();

//Creazione di una LVar bound senza nome associato
LVar B = new LVar(1);

//Creazione di una LVar bound con nome associato
LVar C = new LVar("C", 3);
```

LSet

La classe `LSet` implementa il concetto di insieme logico.
Come visto in precedenza, un insieme logico può contenere qualsiasi oggetto (in particolare anche un altro `LSet`), ignorando ordine e ripetizione degli elementi.
Similmente alle `LVar`, oggetti di tipo `LSet` possono essere bound o unbound, con o senza nome identificativo associato, e possono inoltre essere completamente specificati (*chiusi*) oppure parzialmente specificati (*aperti*).
Esempi:

```
/* Creazione di un LSet unbound senza nome associato
   (rappresenta un insieme completamente variabile) */
LSet A = new LSet();

/* Creazione di un LSet completamente specificato, bound e senza nome associato
   (rappresenta l'insieme {1,2,3}) */
LSet B = LSet.empty().ins(1).ins(2).ins(3);
```

```
/* Creazione di un LSet parzialmente specificato, unbound e con nome associato
   (rappresenta l'insieme {_LV1, _LV2 | C}) */
LSet C = new LSet("C").ins(new LVar()).ins(new LVar());
```

Come si vedrà in seguito, recentemente è stata aggiunta a JSetL una nuova importante categoria di insiemi logici, quella dei Restricted Intensional Set (si veda [6]), che verranno analizzati e testati nel capitolo 5.9.
Grazie ad un recente lavoro di tesi inoltre (si veda [7]), JSetL implementa, in aggiunta al linguaggio CLP($\mathcal{SET}$) che verrà brevemente discusso nel capitolo 2.2, anche il linguaggio $L_{\mathcal{BR}}$, ossia l'estensione di CLP($\mathcal{SET}$) con il concetto di relazione binaria.
Questa estensione comporta l'aggiunta di classi come `LRel` e `LMap`, di seguito descritte.

LRel

La classe `LRel` implementa le relazioni binarie in JSetL.
Una relazione binaria è rappresentata come un particolare insieme i cui elementi sono coppie ordinate: per questo motivo la classe `LRel` estende la classe `LSet`.
Un'istanza di `LRel` è dunque un'istanza di `LSet` in cui gli elementi sono coppie logiche, rappresentate dalla classe `LPair`.
Esempi:

```
/* Creazione di una LRel unbound con nome associato "A" */
LRel A = new LRel("A");

/* Creazione della LRel {[0,1],[2,3]}, senza nome associato */
LRel B = LRel.empty().ins(new LPair(0, 1)).ins(new LPair(2, 3));
```

La classe LMap

La classe `LMap` implementa le funzioni parziali in JSetL.
Una funzione parziale è rappresentata come una particolare relazione binaria in cui i primi componenti di tutte le coppie della relazione sono diversi gli uni dagli altri: per questo motivo la classe `LMap` estende la classe `LRel`.
Esempi:

```
/* Creazione di una LMap unbound con nome associato "A" */
LMap A = new LMap("A");

/* Creazione della LMap {[1,'a'],[2,'b']}, senza nome associato */
LMap B = LMap.empty().ins(new LPair(1, 'a')).ins(new LPair(2, 'b'));
```

La classe Constraint

La classe `Constraint` rappresenta la nozione di vincolo in JSetL.
Un vincolo è un'operazione che può essere applicata ad una variabile logica (`LVar`) o ad un insieme logico (`LSet`, `LRel`, `LMap`).
Un vincolo può essere atomico o composto.

- **Vincolo atomico**
  Un vincolo atomico può essere di queste tipologie:
  - Vincolo *vuoto*, denotato da `[]`
  - $e_0.op(e_1, ..., e_n)$ oppure $op(e_0, e_1, ..., e_n)$ con $n = 0, ..., 3$
    dove *op* è il nome del vincolo, e $e_i (0 \leq i \leq 3)$ sono espressioni il cui tipo dipende da *op*.

- **Vincolo composto**
  Un vincolo composto può essere ottenuto tramite:
  - Congiunzione ($c_1.and(c_2)$)
  - Disgiunzione ($c_1.or(c_2)$)
  - Implicazione ($c_1.impliesTest(c_2)$)

Vengono di seguito riportate due tabelle con i vincoli principali per `LSet` e `LRel`.

Vincoli di LSet

Vincolo	Sintassi	Significato
eq	Constraint eq(LSet s)	`this` $=$ `s`
disj	Constraint disj(LSet s)	`this` $\cap$ `s` $= \varnothing$
union	Constraint union(LSet s, LSet q)	`q` $=$ `this` $\cup$ `s`
in	Constraint in(LSet s)	`this` $\in$ `s`
diff	Constraint diff(LSet s, LSet q)	`q` $=$ `this` $\setminus$ `s`
inters	Constraint inters(LSet s, LSet q)	`q` $=$ `this` $\cap$ `s`
subset	Constraint subset(LSet s)	`this` $\subseteq$ `s`
size	Constraint size(IntLVar n)	`n` $=$ $\lvert$ `this` $\rvert$

Vincoli di LRel

Vincolo	Sintassi	Significato
id	Constraint id(LSet a)	$this = \{(x, x) \mid x \in a\}$
inv	Constraint inv(LRel s)	$s = \{(y, x) \mid (x, y) \in this\}$
comp	Constraint comp(LRel s, LRel q)	$q = \{(x, z) \mid \exists y : (x, y) \in this \wedge (y, z) \in s\}$
dom	Constraint dom(LSet a)	$a = \{x \mid \exists y : ((x, y) \in this)\}$
ran	Constraint ran(LSet a)	$a = \{y \mid \exists x : ((x, y) \in this)\}$

La classe Solver

Il risolutore di vincoli è implementato dalla classe `Solver`.
La classe fornisce metodi per aggiungere, mostrare e risolvere vincoli.
I vincoli vengono risolti tramite regole di riscrittura sintattica, e la procedura di risoluzione termina quando non vi sono più regole da applicare (oppure nel momento in cui viene lanciata un'eccezione di tipo `Failure`, nel caso in cui il vincolo non sia soddisfacibile).
I metodi principali della classe sono i seguenti:

- public void **add**(Constraint c)
  Aggiunge il vincolo `c` al constraint store del risolutore.

- public void **showStore**()
  Stampa la congiunzione di tutti i vincoli presenti nel constraint store del risolutore ancora in forma non risolta.

- public void **solve**()
  Risolve i vincoli nel constraint store del risolutore: se la risoluzione non è possibile, solleva un'eccezione di tipo `Failure`.

- public boolean **check**()
  Analogo al metodo `solve()`, con la differenza che non vengono sollevate eccezioni: ritorna `true` se i vincoli presenti nel constraint store del risolutore sono soddisfacibili, `false` altrimenti.

2.2 CLP($\mathcal{SET}$) e {log}

La programmazione con vincoli (*CP, Constraint Programming*) può essere vista come una forma di programmazione dichiarativa che permette di manipolare esplicitamente vincoli (ossia relazioni su opportuni domini).
In particolare, se il linguaggio utilizzato per la programmazione con vincoli è di tipo logico si parla di programmazione logica con vincoli (*CLP, Constraint Logic Programming*).
Un esempio di linguaggio di programmazione logica con vincoli è CLP($\mathcal{SET}$), in cui il dominio dei vincoli è quello degli insiemi logici. Come visto in precedenza nel capitolo 2.1.1 inoltre, l'estensione del linguaggio a vincoli presente in CLP($\mathcal{SET}$) con l'aggiunta delle relazioni binarie prende il nome di linguaggio $L_{\mathcal{BR}}$.

$L_{\mathcal{BR}}$ è implementato nella libreria JSetL, in un contesto di programmazione O-O. Esiste un'implementazione di $L_{\mathcal{BR}}$ anche per Prolog, che prende il nome di {log} (da leggersi *setlog*).

{log} è un interprete inizialmente sviluppato presso il Dipartimento di Matematica e Informatica di Udine.
Il suo scopo è quello di implementare il linguaggio CLP($\mathcal{SET}$) e le successive estensioni del suo linguaggio a vincoli in un contesto di programmazione logica.
{log} è di fatto molto simile a Prolog nella sintassi, ed è utilizzabile sia in modalità interattiva a riga di comando, sia come modulo esterno da includere nel proprio sorgente Prolog.

Anche {log} fornisce una serie di vincoli di base su insiemi e relazioni: di seguito viene proposta una panoramica dei vincoli visti in precedenza per il caso di JSetL.

Vincoli su insiemi

Vincolo	Sintassi	Significato
=	A = B	$\mathtt{A} = \mathtt{B}$
disj	disj(A, B)	$\mathtt{A} \cap \mathtt{B} = \varnothing$
un	un(A, B, C)	$\mathtt{C} = \mathtt{A} \cup \mathtt{B}$
in	x in A	$\mathtt{x} \in \mathtt{A}$
diff	diff(A, B, C)	$\mathtt{C} = \mathtt{A} \setminus \mathtt{B}$
inters	inters(A, B, C)	$\mathtt{C} = \mathtt{A} \cap \mathtt{B}$
subset	subset(A, B)	$\mathtt{A} \subseteq \mathtt{B}$
size	size(A, N)	$\mathtt{N} = \mid \mathtt{A} \mid$

Vincoli su relazioni

Vincolo	Sintassi	Significato
id	id(R, A)	$\mathtt{R} = \{(x, x) \mid x \in A\}$
inv	inv(R, S)	$\mathtt{R} = \mathtt{S}^{-1}$
comp	comp(R, S, T)	$\mathtt{T} = \mathtt{R} \circ \mathtt{S}$
dom	dom(R, A)	$\mathtt{dom\ R} = \mathtt{A}$
ran	ran(R, A)	$\mathtt{ran\ R} = \mathtt{A}$

2.3 JUnit

JUnit è un framework open-source per il linguaggio di programmazione Java pensato per fare unit testing in maniera semplice ed organizzata.
Il framework si basa principalmente su annotazioni e asserzioni, utilizzate in maniera opportuna.
Uno *unit test* è tipicamente realizzato attraverso una classe al cui interno sono presenti diversi metodi, detti *test cases*.
Vi sono delle precise regole da seguire per la creazione di uno unit test:

- Ogni metodo di test deve avere visibilità `public` e tipo di ritorno `void`;
- Ogni metodo di test deve essere annotato con una delle annotazioni fornite da JUnit (la più tipica è `@Test`);
- La valutazione del risultato deve essere fatta attraverso i metodi della classe `junit.framework.Assert`, la quale fornisce diverse tipologie di asserzioni.

In JUnit 4 (il più utilizzato al momento della scrittura di questa tesi) tutti i metodi di test vengono eseguiti in maniera sequenziale, ma in un ordine non meglio precisato.
Da JUnit 5 è stata invece introdotta la possibilità di eseguire più metodi di test in parallelo tra loro.

2.3.1 Annotazioni principali

Le annotazioni principali fornite da JUnit sono:

- **@Test**
  Dichiara che ciò che segue è un metodo di test. Ogni metodo preceduto da questa annotazione verrà testato a run-time.

- **@Before**
  Dichiara che il metodo che segue deve essere eseguito prima di ogni esecuzione di un caso di test (tipicamente per re-inizializzare i parametri prima di ogni test).

- **@After**
  Dichiara che il metodo che segue deve essere eseguito dopo ogni esecuzione di un caso di test (tipicamente per rilasciare le risorse precedentemente allocate).

- **@BeforeClass**
  Dichiara che il metodo che segue sarà necessariamente statico, e verrà eseguito una sola volta prima di eseguire il primo test case della classe (tipicamente per inizializzare dati globali).

- **@AfterClass**
  Dichiara che il metodo che segue sarà necessariamente statico, e verrà eseguito una sola volta dopo aver eseguito l'ultimo test case della classe (tipicamente per effettuare operazioni finali una volta ottenuti tutti i risultati dei test, come ad esempio un report su file).

2.3.2 Asserzioni principali

Le asserzioni principali fornite da JUnit sono:

- public static void **assertTrue**(boolean condition)
  Asserisce che `condition` sia valutata a `true`.

- public static void **assertFalse**(boolean condition)
  Asserisce che `condition` sia valutata a `false`.

- public static void **assertEquals**(Object expected, Object actual)
  Asserisce che `expected` sia valutato uguale a `actual`.

- public static void **assertNull**(Object obj)
  Asserisce che `obj` sia valutato a `null`.

2.3.3 Gestione delle eccezioni

Altro aspetto da considerare è la gestione delle eccezioni, diversa a seconda che si utilizzi JUnit 4 o JUnit 5.

In JUnit 4 è necessario aggiungere, di seguito alla direttiva `@Test`, l'eccezione che si presuppone venga sollevata a causa dell'esecuzione del metodo, tramite la seguente sintassi:

```
@Test(expected = MyException.class)
```

In JUnit 5 viene introdotta invece l'asserzione `assertThrows` che permette un controllo molto più preciso sulle eccezioni, in quanto è possibile testarne più di una in un unico test case.
La sintassi, che fa uso delle lambda-expressions di Java 8, è la seguente:

```
assertThrows(MyException.class, () -> doOperation(),
            "Expected doOperation() to throw MyException,
             but it didn't");
```

2.3.4 Un esempio completo

Di seguito viene presentato un esempio di struttura completa di uno unit test, prendendo come esempio un'ipotetica classe `Razionale` che implementa appunto i numeri razionali.

```
public class TestRazionale {
    //...

    @Test
    public void test1() {
        Razionale a = new Razionale(2, 3); // crea il numero razionale 2/3
        Razionale b = new Razionale(4, 6);
        assertTrue(a.equals(b));
    }

    @Test
    public void test2() {
        Razionale a = new Razionale(1, 2);
        assertNotNull(a);
    }

    //...
}
```

3 Introduzione allo strumento sviluppato

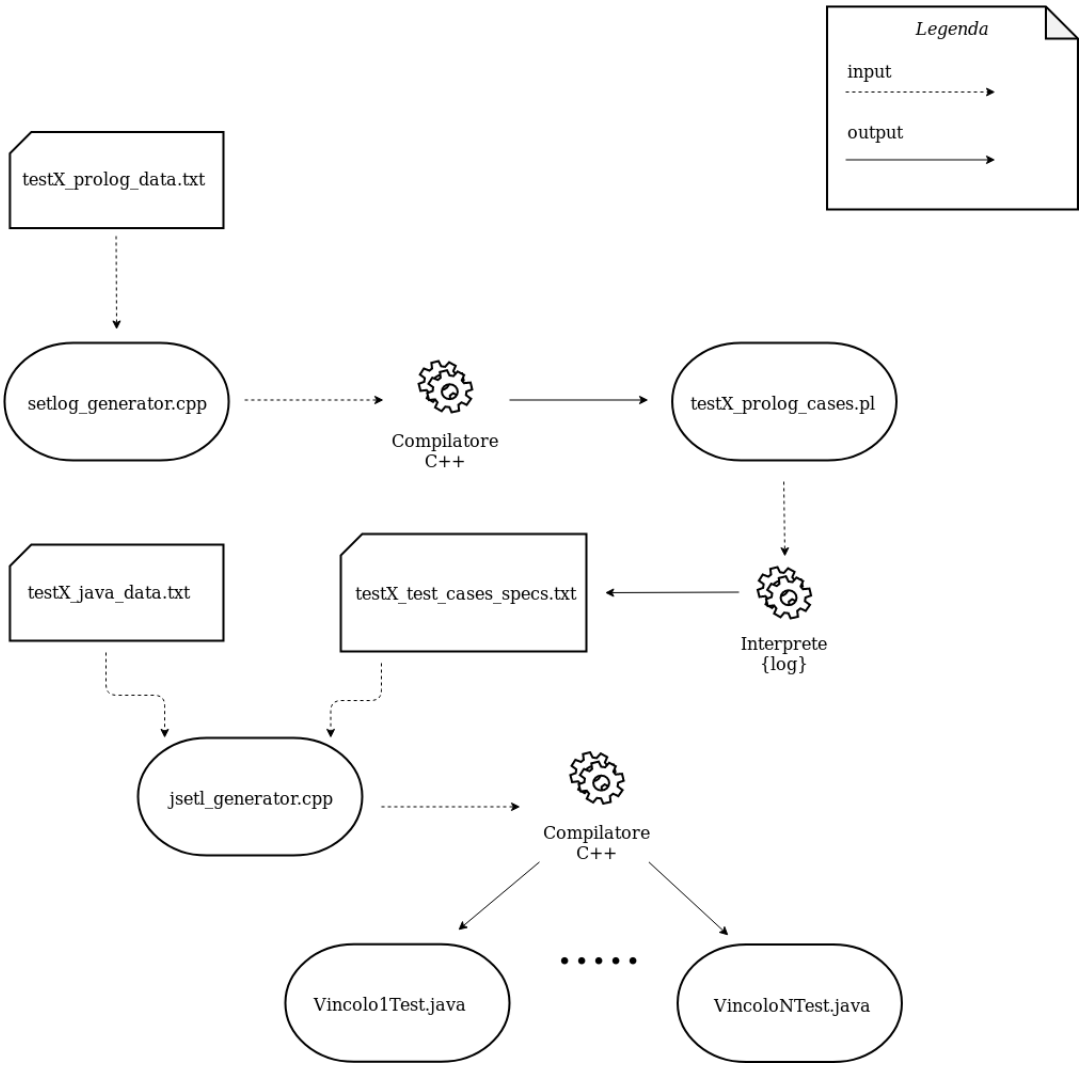
3.1 Motivazioni

Lo strumento software sviluppato per questo lavoro di tesi ha lo scopo di automatizzare il processo di testing di vincoli per la già citata libreria JSetL.
Il linguaggio principale di implementazione è il C++, ma sono presenti anche parti di codice in Prolog e in Java.
Rifacendosi alla classificazione delle tipologie di testing illustrata nel capitolo 1, lo strumento si inserisce nella categoria di *black-box* testing, ed in particolare nella sottocategoria *boundary-value analysis* con criterio di input space partitioning *all combinations*.
Per quanto riguarda le fasi del processo di testing, lo strumento si occupa in particolare di trattare la fase di *unit testing*.
A differenza dei classici tool di generazione automatica di test, quello realizzato in questo lavoro di tesi è pensato specificatamente per JSetL: ciò permette di avere qualche vantaggio in più, come evidenziato di seguito.
JSetL è sostanzialmente l'implementazione Java del linguaggio $L_{\mathcal{BR}}$.
Esiste però, come citato in precedenza, anche un'analoga implementazione per il linguaggio Prolog, ossia {log}.
Il fatto di avere una doppia implementazione non è assolutamente banale per un sistema software, e per uno strumento che deve effettuarne il testing può rivelarsi un notevole punto di forza.
Un primo vantaggio è dato dal fatto che all'utente è richiesto solamente di specificare la forma che dovrà avere il test set, senza mai dover esplicitare il risultato di ogni test in anticipo (non è necessario sapere se un vincolo è o meno soddisfacibile). La soddisfacibilità è infatti "calcolata" dall'interprete {log}: lo strumento sfrutta {log} per ricavare i risultati dei singoli test (true o false), per poi riportarli (senza ricalcolarli) sui test JSetL.

In questo modo si ha un ulteriore vantaggio indiretto: un'eventuale incongruenza sul risultato di un test è subito visibile, in quanto la direttiva *assert* corrispondente nel codice Java non andrebbe a buon fine.
Questa incongruenza è spesso un campanello d'allarme che può aiutare lo sviluppatore a capire se vi sono incoerenze tra le due implementazioni.

3.2 Architettura del sistema

L'architettura del sistema software sviluppato è descritta dal seguente schema.



L'utente avvia il processo di generazione dei test fornendo in input al sistema due file: `testX_prolog_data.txt` e `testX_java_data.txt`.

I due file contengono le informazioni fondamentali sul test set che si intende generare, ossia quali sono i vincoli che si vogliono testare e su quali argomenti essi devono essere istanziati (per test set si intende una collezione di casi di test riferiti ad uno o più vincoli, dove ogni singolo caso è un test istanziato su una particolare combinazione degli argomenti).

`testX_prolog_data.txt` specifica questi vincoli e argomenti in sintassi {log}, mentre in `testX_java_data.txt` sono espressi in sintassi Java.
Il contenuto semantico dei due file è però identico: vincoli e argomenti devono essere uguali in numero e devono soprattutto mantenere il medesimo significato.
Una volta forniti questi due file di input, la cui sintassi sarà illustrata in seguito nel dettaglio, il processo di generazione può avere inizio.

Il programma `setlog_generator.cpp` riceve in input il file `testX_prolog_data.txt`.
Una volta compilato, genera in output il file `testX_prolog_cases.pl`.
Quest'ultimo è un programma Prolog contenente il test di ogni vincolo su tutti i possibili argomenti specificati, ovviamente con sintassi {log}.

Il file `testX_prolog_cases.pl` viene quindi dato in input all'interprete {log}, che genererà in output il file `testX_test_cases_specs.txt`.
Questo file testuale contiene una serie di record in cui vengono specificati il nome del vincolo, i suoi argomenti ed il risultato ottenuto tramite l'interprete {log} eseguendo quel particolare caso di test (true o false).

Quest'ultimo file ed il file `testX_java_data.txt` sono infine dati in input al programma `jsetl_generator.cpp`, il quale genera in output n classi Java (con $n =$ numero di vincoli definiti nelle specifiche dall'utente).

Ogni classe Java si occupa quindi del test di un solo vincolo, e per ogni vincolo vengono generati i relativi metodi di test.
Gli argomenti su cui ogni metodo di test è istanziato sono ottenuti dal file `testX_java_data.txt`, mentre il risultato del test (true o false) è ricavato dal file `testX_test_cases_specs.txt`, ossia dai risultati generati dall'interprete {log}.

Durante l'esecuzione di `jsetl_generator.cpp`, è inoltre richiesto all'utente se è interessato o meno ai tempi di esecuzione di ciascun vincolo.
Nel caso la risposta sia affermativa, ogni classe Java verrà integrata con una parte aggiuntiva di codice per permettere la stampa dei tempi di esecuzione su un file denominato `times.txt`.

Una volta ottenute le n classi Java, è sufficiente caricarle nel proprio IDE Java, assicurarsi di avere disponibili sia JSetL che JUnit, e dare il comando *"run as JUnit test"*.
Se la stampa dei tempi è stata abilitata, dopo aver dato questo comando verrà inoltre creato il file `times.txt`, in cui verrà aggiunto un record contenente il nome del vincolo appena testato con il relativo tempo di esecuzione espresso in millisecondi, calcolato come somma di tutti i tempi di esecuzione dei singoli metodi di test.

4 Lo strumento nel dettaglio

Come visto nel capitolo precedente, lo strumento si compone di due programmi principali aventi lo scopo di generare casi di test.
Entrambi i programmi ricevono in input alcuni file e ne generano in output altri: scopo di questo capitolo è quello di analizzare nel dettaglio questi file, descrivendone il loro contenuto sia a livello sintattico che a livello semantico, con relativi esempi d'utilizzo. Al termine del capitolo vengono infine presentati i due programmi generatori in C++, `setlog_generator.cpp` e `jsetl_generator.cpp`.

4.1 File di input

4.1.1 testX_prolog_data.txt

Il file `testX_prolog_data.txt` (dove X è una stringa che individua in modo univoco lo specifico test set, ad es.: un numero progressivo) è fornito dall'utente, ed ha l'obiettivo di specificare quali vincoli andare a testare (es.: `=`/2, `disj`/2,...) e su quali argomenti (es.: insieme vuoto, insieme chiuso, ...).
Vincoli e argomenti devono essere specificati in sintassi {log}.
Il file deve essere scritto secondo una precisa sintassi, di seguito riportata.

```
$$$ user rules $$$
<regola_1>
...

$$$ args $$$
<nome_arg>$<arg_1>$...$<arg_n>
...

$$$ constraints $$$
<nome_vincolo_jsetl>$<arita'>$<prefisso/infisso>$<nome_vincolo_{log}>
...
```

Il file è diviso in 3 sezioni:

- `$$$ user rules $$$`
  Questa sezione, che può essere eventualmente vuota, contiene una serie di regole Prolog che l'utente può specificare per creare test ad hoc di vincoli nuovi. Ad esempio, è possibile testare la congiunzione di due vincoli esplicitando la seguente regola:

  ```
  union4(A,B,C,D) :- un(A,B,C) & un(A,B,D).
  ```

  Tramite questa regola Prolog, si definisce un nuovo vincolo denominato `union4` che risulterà soddisfacibile se e solo se:
  $\mathtt{A} \cup \mathtt{B} = \mathtt{C} \wedge \mathtt{A} \cup \mathtt{B} = \mathtt{D}$
  (dove il simbolo = indica l'operazione di unificazione logica).

- `$$$ args $$$`
  Questa sezione contiene gli argomenti su cui istanziare i vincoli (es.: insiemi, relazioni, ...). Ogni argomento è così strutturato:

  `<nome_arg>`: è il nome che l'utente sceglie per identificare quel tipo di argomento.

  `<arg_i>`: è il valore i-esimo, espresso in sintassi {log}, che l'utente sceglie di assegnare per quello specifico argomento.

  Ad esempio, 4 tipici argomenti per vincoli insiemistici sono i seguenti:

  ```
  $$$ args $$$
  vuoto${}${}${}${}
  var$S1$S2$S3$S4
  chiuso${X1,Y1}${X2,Y2}${X3,Y3}${X4,Y4}
  aperto${A1/B1}${A2/B2}${A3/B3}${A4/B4}
  ```

Ogni argomento ha quindi un nome, seguito da una lista di possibili valori. Nell'esempio appena riportato, gli argomenti sono 4:

- Il primo argomento identifica la classe degli insiemi vuoti;
- Il secondo, identifica la classe degli insiemi variabili;
- Il terzo, identifica la classe degli insiemi di cardinalità massima prefissata (in questo caso, 2);
- Il quarto, identifica la classe degli insiemi parzialmente specificati (`AX` è un elemento dell'insieme, `BX` è la parte variabile).

Notare che il numero dei possibili valori non è mai casuale, ma è dettato dai vincoli scelti: volendo infatti testare un vincolo avente arità n, è necessario fornire per ogni argomento almeno n possibili valori.
Il numero di valori deve quindi essere almeno pari alla massima arità presente nei vincoli specificati (in questo caso 4, siccome l'unico vincolo presente è il vincolo `union4`).

- `$$$ constraints $$$`
  Questa sezione contiene i vincoli che si intendono testare in questo test set (ogni vincolo specificato formerà, al termine del processo, una classe Java). Il generico vincolo è così strutturato:

  `<nome_vincolo_jsetl>`: è il nome del vincolo in JSetL. Il nome JSetL può essere differente dal nome {log}: è importante non confondere i due, in quanto le informazioni sui nomi dei vincoli sono determinanti per la corretta generazione dei test da parte dello strumento (in particolare, il nome del vincolo JSetL sarà poi riportato nel file `testX_test_cases_specs.txt`);

  `<arità>`: è il numero di argomenti su cui questo vincolo deve essere istanziato;

<prefisso/infisso>: indica se il vincolo è in forma prefissa (prefix) o infissa (infix);

<nome_vincolo_{log}>: è il nome del vincolo in {log}.

Esempio di alcuni vincoli insiemistici:

```
$$$ constraints $$$
eq$2$infix$=
neq$2$infix$neq
disj$2$prefix$disj
union$3$prefix$un
union4$4$prefix$union4
```

Nota: anche il vincolo union4, definito da utente, deve essere riportato nella lista dei vincoli.

4.1.2 testX_java_data.txt

Il file testX_java_data.txt è il secondo ed ultimo file fornito dall'utente.
La sua funzione è analoga a quella del file precedente (file testX_prolog_data.txt), con la differenza che la sintassi con cui vengono specificati vincoli e argomenti è ora quella di Java, esteso con le classi e i metodi forniti da JSetL.
È molto importante che gli argomenti e i vincoli siano identici in numero e abbiano lo stesso significato sia in testX_prolog_data.txt che in testX_java_data.txt.

La struttura del file è la seguente:

```
$$$ global $$$
<dichiarazioni Java visibili in tutta la classe>

$$$ local $$$
<dichiarazioni Java locali ad ogni metodo>

$$$ args $$$
<nome_arg>$<arg_1>$...$<arg_n>
...

$$$ constraints $$$
<nome_vincolo_jsetl>$<arita'>$<prefisso/infisso>
...
```

Il file risulta quindi composto di 4 sezioni:

- `$$$ global $$$`
  Questa sezione contiene una lista di dichiarazioni Java visibili ad ogni singolo metodo della classe.
  Lo scopo di questa sezione è analogo a quello della sezione `$$$ user rules $$$` del file precedente: inserire regole definite da utente.
  La singola regola è incapsulata all'interno di un metodo Java: questa sezione contiene quindi una lista, eventualmente vuota, di metodi Java.
  Ad esempio, volendo riutilizzare la regola utente definita nel file precedente, occorrerebbe scrivere quanto segue:

  ```
  $$$ global $$$
  public Constraint union4(LSet A, LSet B, LSet C, LSet D) {
      return C.union(A, B).and(D.union(A, B));
  }
  ```

- `$$$ local $$$`

  Questa sezione contiene dichiarazioni Java che verranno ripetute all'inizio di ogni metodo della classe.

  Questa sezione è necessaria in quanto in Java esistono problematiche di scoping non presenti in Prolog.

  In particolare, volendo utilizzare più volte una stessa variabile all'interno di un metodo di test, è necessario dichiarare tale variabile prima di utilizzarla (cosa non vera in Prolog).

- `$$$ args $$$`

  Questa sezione è analoga a quella del file precedente.

  I nomi degli argomenti devono essere esattamente gli stessi del file precedente, mentre i singoli valori di ogni argomento devono essere convertiti con la sintassi prevista da JSetL, avendo cura di preservare la semantica data in {log}. Ad esempio, volendo riprendere gli argomenti presenti nel file precedente, si scriverebbe quanto segue:

```
$$$ args $$$
vuoto$LSet E1 = LSet.empty()$LSet E2 = LSet.empty()
$LSet E3 = LSet.empty()$LSet E4 = LSet.empty()

var$LSet V1 = new LSet("S1")$LSet V2 = new LSet("S2")
$LSet V3 = new LSet("S3")$LSet V4 = new LSet("S4")

chiuso$LSet C1 = LSet.empty().ins(new LVar("X1")).ins(new LVar("Y1"))
$LSet C2 = LSet.empty().ins(new LVar("X2")).ins(new LVar("Y2"))
$LSet C3 = LSet.empty().ins(new LVar("X3")).ins(new LVar("Y3"))
$LSet C4 = LSet.empty().ins(new LVar("X4")).ins(new LVar("Y4"))

aperto$LSet A1 = new LSet("B1").ins(new LVar("A1"))
$LSet A2 = new LSet("B2").ins(new LVar("A2"))
$LSet A3 = new LSet("B3").ins(new LVar("A3"))
$LSet A4 = new LSet("B4").ins(new LVar("A4"))
```

- `$$$ constraints $$$`
  Questa sezione è identica a quella del file precedente, con l'unica differenza che l'ultimo campo, quello relativo al nome del vincolo in {log}, è omesso (in quanto risulta un'informazione inutile giunti a questo punto del processo).
  Volendo quindi testare gli stessi vincoli presenti nel file precedente, si scriverebbe quanto segue:

```
$$$ constraints $$$
eq$2$infix
neq$2$infix
disj$2$infix
union$3$infix
union4$4$prefix
```

  Nota: è necessario riportare tutti i vincoli presenti nel file `testX_prolog_data.txt`, anche quelli definiti da utente.
  Inoltre, come evidente dall'esempio, il fatto che un vincolo sia prefisso in {log} non implica che esso lo sia anche in JSetL.

4.1.3 Valori speciali per gli argomenti

Come visto in precedenza, entrambi i file `testX_prolog_data.txt` e `testX_java_data.txt` hanno una sezione dedicata alla dichiarazione degli argomenti che si intendono utilizzare nel test set.
Vi sono però due tipologie di valori speciali assegnabili agli argomenti che non sono ancora state menzionate, ma che possono talvolta risultare molto utili.

Una tipologia di valore speciale assegnabile ad un argomento è `***`.
Inserendo questa stringa speciale in posizione i-esima tra i valori di un argomento, si ottiene come conseguenza il fatto che il vincolo non sarà mai istanziato con quell'argomento in posizione i-esima.

Grazie a questa sintassi è possibile modellare la tipologia di test che si intende generare definendo le posizioni dei diversi argomenti (esempi di utilizzo sono riportati nel capitolo 5, ad esempio in test2, test3 e test9).

Un'altra tipologia di valore speciale assegnabile ad un argomento è identificata da una delle seguenti 8 stringhe:

- `#LSetFullySpecifiedGround_N`
- `#LSetFullySpecifiedNotGround_N`
- `#LSetPartiallySpecifiedGround_N`
- `#LSetPartiallySpecifiedNotGround_N`
- `#LRelFullySpecifiedGround_N`
- `#LRelFullySpecifiedNotGround_N`
- `#LRelPartiallySpecifiedGround_N`
- `#LRelPartiallySpecifiedNotGround_N`

Ognuna di queste 8 stringhe rappresenta una scorciatoia per ottenere insiemi/relazioni grandi, mantenendo piccoli i file di specifica.
Come intuibile dal nome, ciascuna stringa si occupa di istanziare un insieme/una relazione, completamente/parzialmente specificato/a, di $N \geq 1$ elementi ground/non-ground.
L'utilizzo di queste stringhe è particolarmente utile se abbinato alla stampa dei tempi di esecuzione.
Nel capitolo 5, in test11 e test12, vengono riportati esempi di utilizzo.

4.2 File di output intermedi

4.2.1 testX_prolog_cases.pl

Il file `testX_prolog_cases.pl` è il primo file prodotto in output dallo strumento: in particolare, è il risultato dell'esecuzione del programma `setlog_generator.cpp`. Il file prodotto è un programma Prolog contenente una lista di direttive di test. Ogni direttiva di test si occupa di effettuare il test, tramite l'interprete {log}, di un vincolo su una particolare combinazione degli argomenti.
Ogni vincolo è testato su tutte le possibili combinazioni degli argomenti: in particolare, se sono stati specificati n argomenti ed il vincolo attuale ha arità m, il numero di test case per quel vincolo sarà pari a n^m.

La file `testX_prolog_cases.pl` è così strutturato:

```
test(Op,C,Arg1,Arg2) :-
    setlog(C,_),!,write(Op),write(' '),write(Arg1),write(' '),
    write(Arg2),write(true),nl.
test(Op,_C,Arg1,Arg2) :-
    write(Op),write(' '),write(Arg1),write(' '),
    write(Arg2),write(false),nl.

...

:- tell('testX/testX_test_cases_specs.txt').

<lista (opzionale) di direttive setlog_clause>

<lista dei test>

:- told.
:- halt.
```

Come evidente, il programma si compone di diverse sezioni:

- Le prime due regole sono quelle che si occupano dell'esecuzione vera e propria di ogni test: la prima è quella per il caso true, la seconda per il caso false. La regola che tra le due viene soddisfatta (notare che esse sono in mutua esclusione) ne riporta anche il risultato (true/false) su un file denominato `testX_test_cases_specs.txt`, che verrà dettagliato meglio nel prossimo paragrafo. Le regole in realtà non sono sempre 2, ma crescono di numero a coppie: se nel file `testX_prolog_data.txt` la massima arità fosse m, verrebbero generate in automatico tutte le regole Prolog per gestire i casi di arità compresi tra 2 e m (generando quindi $2(m-2)$ regole in totale).

- La lista di direttive `setlog_clause` è costituita da una serie, eventualmente vuota, di direttive aventi la forma:

  ```
  :- setlog_clause(<regola utente definita in testX_prolog_data.txt>).
  ```

  Questa lista di direttive ha lo scopo di informare l'interprete {log} riguardo i nuovi vincoli definiti da utente.
  Ad esempio, se l'utente avesse inserito il vincolo `union4` nel file `testX_prolog_data.txt`, sarebbe stata generata la seguente direttiva:

  ```
  :- setlog_clause(union4(A,B,C,D) :- un(A,B,C) & un(A,B,D)).
  ```

- La lista dei test è costituita da una serie di direttive aventi la forma:

  ```
  :- test(nome_vincolo_jsetl, nome_vincolo_setlog(arg1,...,argn),
  nome_arg1, ..., nome_argn).
  ```

  Esempio di test per il vincolo `disj` istanziato su due insiemi vuoti:

  ```
  :- test(disj, disj({}, {}), vuoto, vuoto).
  ```

Supponendo siano stati specificati 4 tipi di argomenti, e sapendo che il vincolo `disj` ha arità pari a 2, si avrebbero in totale $2^4 = 16$ casi di test solamente per questo vincolo. Considerando che tipicamente vi sono più vincoli in uno stesso test set, si intuisce come la lista dei test sia la sezione predominante del programma `testX_prolog_cases.pl`.

4.2.2 testX_test_cases_specs.txt

Il file `testX_test_cases_specs.txt` è il risultato dell'interpretazione del programma generato allo step precedente, ossia `testX_prolog_cases.pl`.
Il file contiene la lista dei test appena eseguiti dall'interprete {log}, con il relativo risultato (true/false).
Ogni record del file ha la seguente struttura:

```
<nome_vincolo_jsetl>  <arg1>  ...  <argn>  <risultato>
```

Esempio di record per il vincolo `disj` nel caso di due insiemi vuoti:

```
disj  vuoto  vuoto  true
```

4.3 File di output finali: test JUnit

I programmi Java finali (uno per ogni vincolo) sono l'output dell'esecuzione del programma `jsetl_generator.cpp`, il quale a sua volta riceve in input due file: `testX_java_data.txt` e `testX_test_cases_specs.txt`.
Grazie a questi due file, il programma riesce a generare una serie di metodi di test per il framework JUnit, e ad incapsularli in una classe avente il nome del vincolo che si sta testando.
Ad esempio, il programma Java che testa il generico vincolo `myConstr` avrà la seguente struttura.

```
public class MyConstrTest {

    @Test
    public void testMyConstr_NomeArg1_..._NomeArgN() {
        Solver solver = new Solver();
        <Dichiarazione Arg1>
        ...
        <Dichiarazione ArgN>
        solver.add(Arg1.myConstr(Arg2,...,ArgN));
        <assertTrue(solver.check()) / assertFalse(solver.check())>
    }

    ...

}
```

I file Java finali sono quindi costituiti da un'unica classe, contenente una serie di metodi di test JUnit che condividono una struttura comune.
Ogni metodo è marcato con l'annotazione `@Test`, che lo identifica come metodo di test JUnit.
I metodi di test marcati `@Test` vengono eseguiti sequenzialmente ed in mutua esclusione, ma senza un ordine predefinito.
Ogni metodo di test inizia il suo flusso d'esecuzione creando un'istanza della classe `Solver`, la classe JSetL che fornisce i metodi per aggiungere vincoli, risolverli mediante regole di riscrittura, controllarne la soddisfacibilità e mostrare lo stato attuale del constraint store.
È fondamentale che ogni metodo di test dichiari una nuova istanza di `Solver`: per essere certi della correttezza del test che si sta eseguendo infatti, è necessario avere la garanzia che il constraint store sia vuoto prima dell'aggiunta del vincolo attuale allo store (fatto non garantito se si scegliesse di utilizzare un'unica istanza globale di `Solver`).
Il flusso d'esecuzione del metodo di test prosegue quindi con le dichiarazioni degli argomenti, ricavate dal file `testX_java_data.txt`.

Avendo a disposizione l'istanza della classe `Solver` e gli argomenti da utilizzare, è quindi possibile istanziare il vincolo con gli argomenti dichiarati e aggiungerlo al constraint store: questo è il compito del metodo `add` della classe `Solver`.
Infine, il metodo `check()` della classe `Solver` si occupa di risolvere il vincolo mediante regole di riscrittura e restituirne il risultato (true o false).
Come visto in precedenza però, il risultato di ogni caso di test è già noto dal file `testX_test_cases_specs.txt`: questa informazione viene sfruttata per scegliere quale tipo di assert inserire nel metodo di test attuale (`assertTrue` o `assertFalse`).

Esempio di metodo di test per il vincolo `disj` istanziato su due insiemi vuoti:

```
@Test
public void testDisj_Vuoto_Vuoto() {
    Solver solver = new Solver();
    LSet E1 = LSet.empty();
    LSet E2 = LSet.empty();
    solver.add(E1.disj(E2));
    assertTrue(solver.check());
}
```

Una volta ottenuti i file Java, è sufficiente caricarli nel proprio IDE, assicurarsi che sia presente il framework JUnit, ed eseguire ogni programma in modalità *"JUnit Test"* (in Eclipse: tasto destro sul file Java → Run As → JUnit Test).

Come accennato nel capitolo 3, un'eventuale incongruenza tra i risultati forniti dall'interprete {log} (tramite il file `testX_test_cases_specs.txt`) e quelli calcolati ora da JUnit, causa un errore visibile a tempo di esecuzione: l'IDE Java utilizzato segnalerebbe il fatto che uno o più metodi di test contengono delle asserzioni contraddittorie (in particolare, verrebbe sollevata un'eccezione di tipo `java.lang.AssertionError`).

4.4 Tempi d'esecuzione

Lo strumento offre, in parallelo al testing di soddisfacibilità dei vincoli, anche la possibilità di misurare le prestazioni dei test eseguiti.
Il calcolo dei tempi è particolarmente utile se abbinato al caso di insiemi/relazioni grandi: aumentando le dimensioni degli argomenti, è facile andare ad individuare quali vincoli scalano bene e quali no.

La stampa dei tempi è totalmente automatizzata, ed è attivabile dall'utente in fase di esecuzione del programma `jsetl_generator.cpp`.
Nel caso in cui l'utente scelga di attivarla, le classi Java di test generate dallo strumento vengono leggermente modificate:

```
public class MyConstrTest {

    //somma dei tempi parziali di tutti i metodi
    private static long totTime = 0;

    @Test
    public void testMyConstr_NomeArg1_..._NomeArgN() {
        //timer start
        long start = System.currentTimeMillis();

        Solver solver = new Solver();
        <Dichiarazione Arg1>
        ...
        <Dichiarazione ArgN>
        solver.add(Arg1.myConstr(Arg2,...,ArgN));
        <assertTrue(solver.check()) / assertFalse(solver.check())>

        //timer stop
        long stop = System.currentTimeMillis();
        totTime += (stop - start);
    }
```

```
    //...

    @AfterClass
    public static void printTotTime() {
        PrintWriter writer;
        try {
            String dir = "my/path/.../";
            writer = new PrintWriter(
                new FileOutputStream(
                    new File(dir + "times.txt"), true
                    )
                );
            writer.println("MyConstr" + " " + totTime);
            writer.close();
        } catch (Throwable t) {t.printStackTrace();}

    }

}
```

Come evidente, è stata inserita una nuova variabile statica `totTime` inizializzata a 0, che conterrà il tempo di esecuzione totale dell'intera classe, espresso in millisecondi. Ogni metodo registra due istanti temporali, uno all'inizio del test ed uno al termine. Calcolando la differenza tra l'ultimo ed il primo si ottiene il tempo di esecuzione del metodo appena eseguito.
Come ultima istruzione, ogni metodo incrementa la variabile `totTime` con il suo tempo parziale appena calcolato.

Quando ogni metodo marcato come `@Test` è stato eseguito, JUnit prevede che vengano eseguiti eventuali metodi marcati come `@AfterClass`.
Per questo motivo è stato inserito il metodo `printTotTime()`: esso ha il compito di stampare su un file denominato `times.txt` il tempo di esecuzione totale della classe, contenuto nella variabile `totTime`.

Le scritture avvengono in modalità *append*: per ogni esecuzione del programma Java quindi, verrà prodotto un nuovo record nel file `times.txt`.

Ogni record del file `times.txt` ha la seguente forma:

```
<nome_vincolo_jsetl>  <tempo_di_esecuzione>
```

4.5 Gli script di automazione

Come visto in precedenza, il processo di generazione dei test si compone di diversi step. Per rendere lo strumento più user-friendly, è stato inoltre creato uno script avente lo scopo di automatizzare compilazione ed esecuzione dei diversi programmi: in questo modo si ha un notevole risparmio di tempo e viene minimizzato il più possibile l'eventuale errore umano.

Lo script è scritto in due versioni, una per sistemi Windows, l'altra per sistemi Linux. Entrambi svolgono gli stessi step in successione, come raffigurato all'inizio del capitolo 3.

Vengono di seguito riportati i due script.

File `script_linux.bash`

```
#!/bin/sh

echo "Insert test set: "
read test_set

echo "Compiling 'setlog_generator.cpp' ..."
g++ -Wall -Wextra setlog_generator.cpp -o setlog_generator

echo "Executing 'setlog_generator' ..."
./setlog_generator $test_set

echo "Launching swipl ..."
swipl -s setlog/setlog496-5c.pl ${test_set}/${test_set}_prolog_cases.pl

echo "Compiling 'jsetl_generator.cpp' ..."
g++ -Wall -Wextra jsetl_generator.cpp -o jsetl_generator

echo "Executing 'jsetl_generator' ..."
./jsetl_generator $test_set

echo "Test generation completed."

rm setlog_generator jsetl_generator
```

File `script_windows.bat`

```
@ECHO OFF

SET /P test_set=Insert test set:

ECHO Compiling 'setlog_generator.cpp' ...
g++ -Wall -Wextra setlog_generator.cpp -o setlog_generator.exe

ECHO Executing 'setlog_generator.exe' ...
setlog_generator.exe %test_set%

ECHO Launching swipl...
swipl -s setlog/setlog496-5c.pl %test_set%/%test_set%_prolog_cases.pl

ECHO Compiling 'jsetl_generator.cpp' ...
g++ -Wall -Wextra jsetl_generator.cpp -o jsetl_generator.exe

ECHO Executing 'jsetl_generator.exe' ...
jsetl_generator.exe %test_set%

ECHO Test generation completed.

DEL setlog_generator.exe jsetl_generator.exe

PAUSE
```

4.6 Programmi generatori

Come visto in precedenza, i programmi `setlog_generator.cpp` e `jsetl_generator.cpp` costituiscono la parte più importante dello strumento, in quanto fungono da generatori di casi di test a partire da uno o più file di specifica.
Entrambi sono scritti in linguaggio C++: in particolare si attengono allo standard C++98, per garantire una maggiore retro-compatibilità.

Vengono di seguito descritti i due programmi.

4.6.1 setlog_generator.cpp

Il programma `setlog_generator.cpp` inizia il suo flusso d'esecuzione effettuando il parsing del file `testX_prolog_data.txt` (la cui sintassi è analizzata nel capitolo 4.1.1).
Questo parsing è realizzato attraverso un loop che cicla su ogni record del file di testo, e termina non appena viene raggiunto il carattere speciale `EOF (End Of File)`.
In base al contenuto del record, il loop prevede azioni differenti:

- Se viene letto un `commento`, si ignora il record corrente e si passa al successivo;
- Se viene letta la stringa `$$$ user rules $$$`, tutti i record che seguono (a meno dei commenti e dei separatori di sezione) verranno letti come regole utente.
  Una regola utente viene trattata in modo molto semplice: essa viene incapsulata in una direttiva `setlog_clause`, e la nuova stringa così generata viene aggiunta in modalità append ad una stringa globale denominata `user_defined_rules`, che conterrà quindi la lista delle direttive `setlog_clause` di tutte le regole definite da utente.

- Se viene letta la stringa `$$$ args $$$`, tutti i record che seguono (a meno dei commenti e dei separatori di sezione) verranno letti come argomenti.
  Gli argomenti sono memorizzati tramite la struttura dati `map`.
  Ogni elemento della mappa è una coppia ⟨chiave, valore⟩, in cui la chiave è rappresentata dal nome dell'argomento, il valore è rappresentato dal vettore dei valori definiti per quell'argomento.
- Se viene letta la stringa `$$$ constraints $$$`, tutti i record che seguono (a meno dei commenti e dei separatori di sezione) verranno letti come vincoli.
  I vincoli sono memorizzati tramite la struttura dati `vector`.
  Ogni elemento del vettore è una `struct` contenente i 4 campi di tipo stringa definiti nel capitolo 4.1.1.

Una volta terminato il parsing del file `testX_prolog_data.txt`, il flusso d'esecuzione procede con la scrittura del file `testX_prolog_cases.pl`.

La parte più onerosa di questa operazione di scrittura è sicuramente rappresentata dal loop che si occupa della generazione dei singoli casi di test.
Il loop procede estraendo i vincoli dal loro vettore globale, e termina quando non vi sono più vincoli da esaminare nel vettore.
Una volta estratto il vincolo attuale, si procede associando questo vincolo a tutte le possibili combinazioni di valori degli argomenti: ogni combinazione, rappresenta un singolo caso di test.

La generazione di tutte le possibili combinazioni è la sezione fondamentale del programma `setlog_generator.cpp`, ed è realizzata tramite il calcolo del prodotto cartesiano fra i vettori degli argomenti.
Per maggiore chiarezza, viene di seguito riportata la funzione che si occupa di tale calcolo:

```
/*
 * Dato in input il vector { {"A1","A2"}, {"{X1, X2}","{X3, X4}"} }
 * restituisce in output il vector { {"A1", "{X1, X2}"}, {"A1", "{X3, X4}"}, {"
     A2", "{X1, X2}"}, {"A2", "{X3, X4}"} }
 */

vector<vector<string> > cart_product (const vector<vector<string> >& v) {
    vector<vector<string> > s;

    s.resize(1);

    for(unsigned int i = 0; i < v.size(); ++i) {
        vector<string> u = v[i];

        vector<vector<string> > r;
        for (unsigned int j = 0; j < s.size(); ++j) {
            vector<string> x = s[j];
            for (unsigned int k = 0; k < u.size(); ++k) {
                string y = u[k];
                r.push_back(x);
                r.back().push_back(y);
            }
        }
        s.swap(r);
    }
    return s;
}
```

4.6.2 jsetl_generator.cpp

Il programma `jsetl_generator.cpp` inizia il suo flusso d'esecuzione effettuando il parsing del file `testX_java_data.txt` (la cui sintassi è analizzata nel capitolo 4.1.2), in maniera del tutto analoga al parsing che `setlog_generator.cpp` effettua su `testX_prolog_data.txt`.

La differenza principale consiste nella generazione dei casi di test: mentre in `setlog_generator.cpp` è necessario calcolare esplicitamente tutte le possibili combinazioni degli argomenti, in `jsetl_generator.cpp` ciò non è più necessario, in quanto il file `testX_test_cases_specs.txt` contiene già tutte le informazioni necessarie: una serie di record contenenti i vincoli, tutte le possibili combinazioni di argomenti e, per ogni combinazione, il risultato prodotto dall'interprete {log}.
Il flusso d'esecuzione di `jsetl_generator.cpp` procede infatti con il parsing del file `testX_test_cases_specs.txt` (la cui sintassi è analizzata nel capitolo 4.2.2). Ogni record letto, corrisponde alla specifica di un metodo di test Java.
Il singolo metodo di test contiene parti di codice costanti, presenti in tutti i metodi, e parti di codice variabili, le quali dipendono dai tipi di argomenti e dal risultato del test.

Per gestire le parti variabili, si interroga la mappa degli argomenti costruita in fase di parsing del file `testX_java_data.txt`: in particolare, per ogni nome di argomento incontrato durante la lettura del record corrente, si ricercano nella mappa quali sono i possibili valori assegnabili (ossia una delle sintassi Java esplicitate dall'utente per quel tipo di argomento).
In questo modo, ogni possibile valore è accoppiato con tutti i restanti possibili valori. Ottenuti questi valori tramite mappatura, si hanno tutti gli elementi per poter creare il metodo di test Java (tenendo conto dell'arità del vincolo corrente e della sua forma infissa/prefissa).

Una volta generato il singolo metodo di test, esso verrà aggiunto alla classe Java che si occupa del vincolo corrente.
Terminata la generazione dei metodi di test relativi ad un vincolo, la procedura di generazione prosegue analizzando i successivi vincoli, e termina non appena i vincoli da analizzare sono esauriti.

5 Esempi di generazione di test

In questo capitolo descriviamo 12 test set per JSetL, mostrando per ciascuno di essi il file di input `testX_prolog_data.txt`, e laddove significativo, anche parti dei file `testX_java_data.txt` e dei file Java finali.

Oltre all'intento ovvio di testare in generale i vincoli di JSetL, lo scopo di questi test set è anche quello di fornire una dimostrazione pratica di come affrontare alcuni scenari frequenti in cui l'utente potrebbe ritrovarsi (es.: testare vincoli definiti da utente, vincoli i cui argomenti condividono più variabili, vincoli su insiemi RIS (Restricted Intensional Set), vincoli su relazioni, ...).

5.1 test1: eq, disj, union - casi standard

Questo test set testa tre vincoli basilari di `LSet` in casi standard, ossia senza variabili ripetute negli insiemi.

- Constraint **eq**(LSet s): soddisfacibile $\iff$ `this` = `s` (ovvero se l'`LSet this` unifica con l'`LSet s`).
- Constraint **disj**(LSet s): soddisfacibile $\iff$ `this` $\cap$ `s` $= \varnothing$.
- Constraint **union**(LSet s, LSet q): soddisfacibile $\iff$ `q` = `this` $\cup$ `s`.

Vengono di seguito riportati i file di specifica Prolog e Java, e una parte del file `EqTest.java`.

`test1_prolog_data.txt`

```
//regole definite da utente
$$$ user rules $$$

//insiemi o relazioni del test set
$$$ args $$$
vuoto${}${}${}
var$S1$S2$S3
chiuso${X1,Y1}${X2,Y2}${X3,Y3}
aperto${A1/B1}${A2/B2}${A3/B3}

//vincoli del test set
$$$ constraints $$$
eq$2$infix$=
disj$2$prefix$disj
union$3$prefix$un
```

`test1_java_data.txt`

```
//porzione visibile ad ogni metodo
$$$ global $$$

//porzione interna allo scope di ogni metodo
$$$ local $$$

//insiemi o relazioni del test set
$$$ args $$$
vuoto$LSet E1 = LSet.empty()$LSet E2 = LSet.empty()$LSet E3 = LSet.empty()
var$LSet V1 = new LSet("S1")$LSet V2 = new LSet("S2")$LSet V3 = new LSet("S3")
chiuso$LSet C1 = LSet.empty().ins(new LVar("X1")).ins(new LVar("Y1"))$LSet C2 =
     LSet.empty().ins(new LVar("X2")).ins(new LVar("Y2"))$LSet C3 = LSet.empty
    ().ins(new LVar("X3")).ins(new LVar("Y3"))
```

```
aperto$LSet A1 = new LSet("B1").ins(new LVar("A1"))$LSet A2 = new LSet("B2").
    ins(new LVar("A2"))$LSet A3 = new LSet("B3").ins(new LVar("A3"))

//vincoli del test set
$$$ constraints $$$
eq$2$infix
disj$2$infix
union$3$infix
```

EqTest.java

```
public class EqTest {

    @Test
    public void testEq_Aperto_Aperto() {
        Solver solver = new Solver();
        LSet A1 = new LSet("B1").ins(new LVar("A1"));
        LSet A2 = new LSet("B2").ins(new LVar("A2"));
        solver.add(A1.eq(A2));
        assertTrue(solver.check());
    }

    @Test
    public void testEq_Aperto_Chiuso() {
        Solver solver = new Solver();
        LSet A1 = new LSet("B1").ins(new LVar("A1"));
        LSet C2 = LSet.empty().ins(new LVar("X2")).ins(new LVar("Y2"));
        solver.add(A1.eq(C2));
        assertTrue(solver.check());
    }

    @Test
    public void testEq_Aperto_Var() {
        Solver solver = new Solver();
        LSet A1 = new LSet("B1").ins(new LVar("A1"));
        LSet V2 = new LSet("S2");
```

```
        solver.add(A1.eq(V2));
        assertTrue(solver.check());
    }

    @Test
    public void testEq_Aperto_Vuoto() {
        Solver solver = new Solver();
        LSet A1 = new LSet("B1").ins(new LVar("A1"));
        LSet E2 = LSet.empty();
        solver.add(A1.eq(E2));
        assertFalse(solver.check());
    }

    // altri 12 metodi di test

}
```

5.2 test2: in, nin

Questo test set testa i seguenti due vincoli di `LSet`:

- Constraint **in**(LSet s): soddisfacibile $\iff$ `this` $\in$ `s`.
- Constraint **nin**(LSet s): soddisfacibile $\iff$ `this` $\notin$ `s`.

In particolare, si è stabilito che `this` fosse sempre una `LVar` come espediente per mostrare l'uso della sintassi `***`: come visto nel capitolo 4.1.3 infatti, inserendo questa stringa speciale in posizione i-esima tra i valori di un argomento, l'effetto ottenuto è quello che il vincolo non avrà mai quell'argomento in posizione i-esima. Grazie a questa sintassi (utile anche in test successivi, come in test3 e test9) è possibile ad esempio decidere che in prima posizione dovranno apparire solamente `LVar`.

Viene di seguito riportato il file di specifica Prolog.

test2_prolog_data.txt

```
$$$ user rules $$$

$$$ args $$$
lVarNotBound$LVNB$***
lVarBound$3$***
vuoto$***${}
var$***$X
chiuso$***${A,B}
aperto$***${A/B}

$$$ constraints $$$
in$2$infix$in
nin$2$infix$nin
```

5.3 test3: eq, disj, union - casi speciali

Questo test set testa gli stessi vincoli del test1, ma in casi speciali.
In particolare, si sono scelte due categorie di casi speciali: insiemi in cui compare una stessa variabile ripetuta più volte e variabili esterne che compaiono in insiemi diversi.
Il primo caso speciale è utile per verificare che i duplicati vengano effettivamente eliminati in un `LSet`, mentre il secondo caso speciale è utile per verificare la falsità del paradosso di Russell.
Queste due casistiche sono ottenute ancora una volta grazie alla sintassi `***`, come visibile nei file di specifica Prolog e Java (in quello Java, si noti l'utilizzo della sezione `$$$ local $$$`).

test3_prolog_data.txt

```
$$$ user rules $$$

$$$ args $$$
var$A$B$B
chiuso${A}${A,A}${A,B,C}
aperto1$***${X/A}$***
aperto2$***${A/R}$***

$$$ constraints $$$
eq$2$infix$=
disj$2$prefix$disj
union$3$prefix$un
```

test3_java_data.txt

```
$$$ global $$$

$$$ local $$$
LSet A = new LSet("A");
LSet B = new LSet("B");

$$$ args $$$
var$LSet V1 = A$LSet V2 = B$LSet V3 = B
chiuso$LSet C1 = LSet.empty().ins(A)$LSet C2 = LSet.empty().ins(A).ins(A)$LSet
    C3 = LSet.empty().ins(A).ins(B).ins(C)
aperto1$***$LSet A1 = A.ins(new LVar("X"))$***
aperto2$***$LSet A1 = new LSet("R").ins(A)$***

$$$ constraints $$$
eq$2$infix
disj$2$infix
union$3$infix
```

I due casi speciali sopra menzionati sono testati in particolare dai seguenti due metodi generati:

```
//Eliminazione dei duplicati
@Test
public void testEq_Chiuso_Chiuso() {
    LSet A = new LSet("A");
    LSet B = new LSet("B");

    Solver solver = new Solver();
    LSet C1 = LSet.empty().ins(A);
    LSet C2 = LSet.empty().ins(A).ins(A);
    solver.add(C1.eq(C2));
    assertTrue(solver.check());
}

//... altri metodi di test...

//Paradosso di Russell
@Test
public void testEq_Var_Aperto2() {
    LSet A = new LSet("A");
    LSet B = new LSet("B");

    Solver solver = new Solver();
    LSet V1 = A;
    LSet A1 = new LSet("R").ins(A);
    solver.add(V1.eq(A1));
    assertFalse(solver.check());
}
```

5.4 test4: vincolo utente

Questo test set ha lo scopo di mostrare come testare un vincolo definito da utente. Si è scelto di testare il seguente vincolo:

- Constraint **myTest**(LSet A, LSet B, LSet C):
  soddisfacibile $\iff$ `A` $\cup$ `B` $=$ `C` $\wedge$ `A` $\cup$ `B` $\neq$ `C`.
  (il vincolo è volutamente sempre falso).

Vengono di seguito riportati i file di specifica Prolog e parte di quello Java (in particolare, la sezione `$$$ global $$$`).

`test4_prolog_data.txt`

```
$$$ user rules $$$
myTest(A,B,C) :- un(A,B,C) & nun(A,B,C).

$$$ args $$$
vuoto${}${}${}
var$S1$S2$S3
chiuso${X1,Y1}${X2,Y2}${X3,Y3}
aperto${A1/B1}${A2/B2}${A3/B3}

$$$ constraints $$$
myTest$3$prefix$myTest
```

`test4_java_data.txt`

```
$$$ global $$$
public Constraint myTest(LSet A, LSet B, LSet C) {
    return C.union(A, B).and(C.nunion(A, B));
}
```

5.5 test5: diff, inters, subset

Questo test set ha lo scopo di testare tre ulteriori vincoli di `LSet` aventi la caratteristica di essere vincoli composti: sono ottenuti come congiunzione di altri vincoli.

- Constraint **diff**(LSet s, LSet q): soddisfacibile $\iff$ `q` = `this` $\setminus$ `s`.
- Constraint **inters**(LSet s, LSet q): soddisfacibile $\iff$ `q` = `this` $\cap$ `s`.
- Constraint **subset**(LSet s): soddisfacibile $\iff$ `this` $\subseteq$ `s`.

Viene di seguito riportato il file di specifica Prolog.

`test5_prolog_data.txt`

```
$$$ user rules $$$

$$$ args $$$
vuoto${}${}${}
var$S1$S2$S3
chiuso${X1,Y1}${X2,Y2}${X3,Y3}
aperto${A1/B1}${A2/B2}${A3/B3}

$$$ constraints $$$
diff$3$prefix$diff
inters$3$prefix$inters
subset$2$prefix$subset
```

5.6 test6: comp, inv, id

Questo test set ha lo scopo di testare tre vincoli di `LRel`.

- Constraint **comp**(LRel s, LRel q): soddisfacibile $\iff$ `q` è la composizione relazionale di `this` e `s`, ovvero:
  $q = \{(x, z) \mid \exists y : (x, y) \in this \ \wedge \ (y, z) \in s\}$
- Constraint **inv**(LRel s): soddisfacibile $\iff$ `s` è la relazione inversa di `this` ovvero:
  $s = \{(y, x) \mid (x, y) \in this\}$
- Constraint **id**(LSet a): soddisfacibile $\iff$ `this` è la relazione identica all'`LSet a`, ovvero:
  $this = \{(x, x) \mid x \in a\}$

Da notare il fatto che, sebbene `id` richieda un `LSet` come argomento, è perfettamente legale passare un `LRel` al suo posto, in quanto `LRel` è sottoclasse di `LSet` (una relazione può essere vista come un insieme di coppie).
Vengono di seguito riportati i file di specifica Prolog e Java.

`test6_prolog_data.txt`

```
$$$ user rules $$$

$$$ args $$$
vuoto${}${}${}
var$A$B$C
chiuso${[A1,B1],[A2,B2]}${[A3,B3],[A4,B4]}${[A5,B5],[A6,B6]}
aperto${[X1,Y1]/R1}${[X2,Y2]/R2}${[X3,Y3]/R3}

$$$ constraints $$$
comp$3$prefix$comp
inv$2$prefix$inv
id$2$prefix$id
```

test6_java_data.txt

```
$$$ global $$$

$$$ local $$$

$$$ args $$$
vuoto$LRel L1 = LRel.empty()$LRel L2 = LRel.empty()$LRel L3 = LRel.empty()
var$LRel L4 = new LRel("L4")$LRel L5 = new LRel("L5")$LRel L6 = new LRel("L6")
chiuso$LRel L7 = LRel.empty().ins(new LPair(new LVar(),new LVar())).ins(new
    LPair(new LVar(),new LVar()))$LRel L8 = LRel.empty().ins(new LPair(new LVar
    (),new LVar())).ins(new LPair(new LVar(),new LVar()))$LRel L9 = LRel.empty
    ().ins(new LPair(new LVar(),new LVar())).ins(new LPair(new LVar(),new LVar
    ()))
aperto$LRel L10 = new LRel("L10").ins(new LPair(new LVar(),new LVar())).ins(new
     LPair(new LVar(),new LVar()))$LRel L11 = new LRel("L11").ins(new LPair(new
     LVar(),new LVar())).ins(new LPair(new LVar(),new LVar()))$LRel L12 = new
    LRel("L12").ins(new LPair(new LVar(),new LVar())).ins(new LPair(new LVar(),
    new LVar()))

$$$ constraints $$$
comp$3$infix
inv$2$infix
id$2$infix
```

Un esempio di metodo generato per la classe `IdTest` è il seguente:

```
    @Test
    public void testId_Var_Var() {
        Solver solver = new Solver();
        LRel L4 = new LRel("L4");
        LRel L5 = new LRel("L5");
        solver.add(L4.id(L5));
        assertTrue(solver.check());
    }
```

5.7 test7: size

Questo test set ha lo scopo di testare un vincolo particolare di `LSet`, il vincolo `size`.

- Constraint **size**(IntLVar n): soddisfacibile $\iff$ `n` = | `this` |

Ovviamente, se `n` è lasciata come variabile unbound, il vincolo è sempre soddisfacibile. Per questo sono state inserite nel test set più possibilità per `n`: in particolare, essa può essere anche una variabile bound (legata ai valori numerici 1, 2 o 3).

Viene di seguito riportato il file di specifica Prolog.

`test7_prolog_data.txt`

```
//regole definite da utente
$$$ user rules $$$

//insiemi o relazioni del test set
$$$ args $$$
lVarNotBound$***$N
lVarBound3$***$3
lVarBound2$***$2
lVarBound1$***$1
vuoto${}$***
var$X$***
chiuso${A,B}$***
aperto${A/B}$***

//vincoli del test set
$$$ constraints $$$
size$2$prefix$size
```

5.8 test8: ndisj, neq, nunion

In questo test set vengono testati gli stessi vincoli testati nel test1, in forma negata. Viene di seguito riportato il file di specifica Prolog.

test8_prolog_data.txt

```
//regole definite da utente
$$$ user rules $$$

//insiemi o relazioni del test set
$$$ args $$$
vuoto${}${}${}
var$S1$S2$S3
chiuso${X1,Y1}${X2,Y2}${X3,Y3}
aperto${A1/B1}${A2/B2}${A3/B3}

//vincoli del test set
$$$ constraints $$$
neq$2$infix$neq
ndisj$2$prefix$ndisj
nunion$3$prefix$nun
```

5.9 test9: insiemi RIS

Questo test set ha lo scopo di testare vincoli su RIS (Restricted Intensional Sets). A differenza dei classici insiemi estensionali, i quali sono definiti enumerando gli elementi che li compongono, un insieme intensionale è definito attraverso le proprietà che i suoi elementi devono soddisfare.

Un RIS è un insieme avente questa forma:

$\{c[\boldsymbol{x}] : D \mid F[\boldsymbol{x}] \bullet P[\boldsymbol{x}]\}$

- `c`: è il termine di controllo del RIS;
- $\boldsymbol{x}$: è il vettore delle variabili presenti in `c`, ossia $\boldsymbol{x} = \langle x_1, ..., x_n \rangle$
- `D`: è il dominio di `c` (deve essere un insieme finito, eventualmente unbound)
- `F[`$\boldsymbol{x}$`]`: è il filtro del RIS, ossia un vincolo contenente $\boldsymbol{x}$
- `P[`$\boldsymbol{x}$`]`: è il pattern del RIS, ossia un'espressione contenente $\boldsymbol{x}$

Il significato di un insieme così costruito è il seguente:
l'insieme di tutte le istanze di `P[`$\boldsymbol{x}$`]` tali che `c[`$\boldsymbol{x}$`]` appartiene a `D`, e `F[`$\boldsymbol{x}$`]` vale.
Ad esempio, l'insieme intensionale $\{2x \mid x \in D \wedge x > 0\}$
può essere scritto in forma di RIS come $\{x : D \mid x > 0 \bullet 2x\}$.

In JSetL, un insieme di questo tipo è istanziabile, ad esempio, attraverso il seguente costruttore della classe `Ris` (la quale è una sottoclasse di `LSet`):

```
public Ris(LObject controlTerm, LSet domain, Constraint filter,
LObject pattern);
```

Per rappresentare l'insieme precedente in JSetL, occorrerebbe quindi scrivere:

```
IntLSet D = new IntLSet("D");
IntLVar x = new IntLVar("x");
Ris a = new Ris(x, D, x.lt(9), x.mul(2));
```

Per utilizzare un RIS nei file di specifica, si farà ovviamente ampio uso della sintassi speciale ***, come visto in precedenza.
Vengono di seguito riportati i file di specifica Prolog e Java.

test9_prolog_data.txt

```
$$$ user rules $$$
myRis(ControlTerm, Domain, Filter, Pattern, InsiemeDaUnificare) :-
    InsiemeDaUnificare = ris(ControlTerm in Domain, Filter, Pattern).

$$$ args $$$
//costruisco cosi' gli elementi del Ris:
//al primo posto, il control term
//al secondo posto, il dominio
//al terzo posto, il filtro
//al quarto posto, il pattern
//al quinto posto, l'insieme da unificare (es.: C = ris(...) )

lvar$LV1$***$***$***$***

vuoto$***${}$***$***${}
chiuso$***${X2,Y2}$***$***${X3,Y3}

filtroTrue$***$***$true$***$***
filtroFalse$***$***$false$***$***
filtroCustom$***$***$LV1 = 4$***$***

pattern1$***$***$***$LV1$***

$$$ constraints $$$
myRis$5$prefix$myRis
```

test9_java_data.txt

```
$$$ global $$$

// es.: s1 = ris(X in {2,3,4},true,X).
public Constraint myRis(LVar controlTerm, LSet domain, Constraint filter,
    LObject pattern, LSet insiemeDaUnificare) {
    return insiemeDaUnificare.eq(new Ris(controlTerm, domain, filter, pattern));
}

$$$ local $$$

$$$ args $$$
//costruisco cosi' gli elementi del Ris:
//al primo posto, il control term
//al secondo posto, il dominio
//al terzo posto, il filtro
//al quarto posto, il pattern
//al quinto posto, l'insieme da unificare (es.: C.eq(new Ris(...)) )

lvar$LVar LV1 = new LVar("LV1")$***$***$***$***

vuoto$***$LSet E2 = LSet.empty()$***$***$LSet E3 = LSet.empty()
chiuso$***$LSet C2 = LSet.empty().ins(new LVar("X2")).ins(new LVar("Y2"))$***$
    ***$LSet C3 = LSet.empty().ins(new LVar("X3")).ins(new LVar("Y3"))

filtroTrue$***$***$Constraint ConstrTrue = new Constraint("_true")$***$***
filtroFalse$***$***$Constraint ConstrFalse = new Constraint("_false")$***$***
filtroCustom$***$***$Constraint ConstrCustom = LV1.eq(4)$***$***

pattern1$***$***$***$LObject pattern1 = LV1$***

$$$ constraints $$$
myRis$5$prefix
```

La classe Java finale è così strutturata:

```
public class MyRisTest {

    public Constraint myRisTest(LVar controlTerm,LSet domain,Constraint filter,
        LObject pattern,LSet insiemeDaUnificare) {
        return insiemeDaUnificare.eq(new Ris(controlTerm, domain, filter,
            pattern));
    }

    @Test
    public void testMyRisTest_Lvar_Chiuso_FiltroCustom_Pattern1_Chiuso() {
        Solver solver = new Solver();
        LVar LV1 = new LVar("LV1");
        LSet C2 = LSet.empty().ins(new LVar("X2")).ins(new LVar("Y2"));
        Constraint ConstrCustom = LV1.eq(4);
        LObject pattern1 = LV1;
        LSet C3 = LSet.empty().ins(new LVar("X3")).ins(new LVar("Y3"));

        solver.add(myRisTest(LV1, C2, ConstrCustom, pattern1, C3));

        assertTrue(solver.check());
    }

    // altri 11 metodi di test

}
```

5.10 test10: vincolo utente con arità > 3

Questo test set ha lo scopo di mostrare come testare un vincolo definito da utente, in modo analogo a quanto visto in test4, con la differenza che il vincolo ha adesso arità > 3.
Il fatto di poter dichiarare vincoli con arità > 3 è reso possibile, a livello di codice sorgente, da una funzione che genera dinamicamente il numero di argomenti tramite l'operazione di prodotto cartesiano, come visto nel capitolo 4.6.1.
Sarebbe infatti impossibile gestire singolarmente ogni particolare caso di arità con una funzione specifica, poiché l'arità può potenzialmente assumere un qualsiasi valore naturale > 0 (in particolare ciò è vero poiché l'utente può sempre definire un nuovo vincolo come composizione di un numero arbitrario di vincoli).
L'esistenza di questo meccanismo permette all'utente di dichiarare vincoli con arità generica in maniera assolutamente naturale, come riportato di seguito nel file Prolog.

test10_prolog_data.txt

```
$$$ user rules $$$
union4(A,B,C,D) :- un(A,B,C) & un(A,B,D).

$$$ args $$$
vuoto${}${}${}${}
var$S1$S2$S3$S4$
chiuso${X1,Y1}${X2,Y2}${X3,Y3}${X4,Y4}
aperto${A1/B1}${A2/B2}${A3/B3}${A4/B4}

$$$ constraints $$$
union4$4$prefix$union4
```

5.11 test11: insiemi di molti elementi

Questo test set ha lo scopo di mostrare come testare vincoli di `LSet` su insiemi grandi a piacere.
L'utente non deve ovviamente esplicitare tutti gli elementi, ma userà la sintassi apposita per identificare gli insiemi di molti elementi, come visto nel capitolo 4.1.3.

Vengono di seguito riportati i file di specifica Prolog e Java.

`test11_prolog_data.txt`

```
//regole definite da utente
$$$ user rules $$$

//insiemi o relazioni del test set
$$$ args $$$
vuoto${}${}${}
var$S1$S2$S3

chiusoGrossoGround$#LSetFullySpecifiedGround_20$***$***
chiusoPiccoloNonGround$***${X2,Y2}${X3,Y3}

aperto${A1/B1}${A2/B2}${A3/B3}

//vincoli del test set
$$$ constraints $$$
eq$2$infix$=
neq$2$infix$neq
disj$2$prefix$disj
union$3$prefix$un
```

test11_java_data.txt

```
//porzione visibile ad ogni metodo
$$$ global $$$

//porzione interna allo scope di ogni metodo
$$$ local $$$

//insiemi o relazioni del test set
$$$ args $$$
vuoto$LSet E1 = LSet.empty()$LSet E2 = LSet.empty()$LSet E3 = LSet.empty()
var$LSet V1 = new LSet("S1")$LSet V2 = new LSet("S2")$LSet V3 = new LSet("S3")

chiusoGrossoGround$#LSetFullySpecifiedGround_20$***$***
chiusoPiccoloNonGround$***$LSet C2 = LSet.empty().ins(new LVar("X2")).ins(new
    LVar("Y2"))$LSet C3 = LSet.empty().ins(new LVar("X3")).ins(new LVar("Y3"))

aperto$LSet A1 = new LSet("B1").ins(new LVar("A1"))$LSet A2 = new LSet("B2").
    ins(new LVar("A2"))$LSet A3 = new LSet("B3").ins(new LVar("A3"))

//vincoli del test set
$$$ constraints $$$
eq$2$infix
neq$2$infix
disj$2$infix
union$3$infix
```

Di seguito viene riportato un frammento di una delle classi Java finali, al fine di mostrare come si traduce in codice la stringa speciale utilizzata per denotare un insieme di molti elementi (in questo particolare caso, un insieme completamente specificato di elementi ground).

```
public class EqTest {

    LSet genLSetFSGround(int n) {
        Integer[] values = new Integer[n];
        for(int i = 0; i < n; ++i) {values[i] = i;}
        LSet res = LSet.empty().insAll(values);
        return res;
    }

    // ...

    @Test
    public void testEq_ChiusoGrossoGround_Vuoto() {

        //timer start
        long start = System.currentTimeMillis();

        Solver solver = new Solver();
        LSet LSetFullySpecifiedGround0 = genLSetFSGround(20);
        LSet E2 = LSet.empty();
        solver.add(LSetFullySpecifiedGround0.eq(E2));
        assertFalse(solver.check());

        //timer stop
        long stop = System.currentTimeMillis();
        totTime += (stop - start);
    }

    // ...

}
```

Come evidente dal codice, la stringa speciale `LSetFullySpecifiedGround_20` è stata tradotta in un metodo Java che si occupa appunto della generazione di un insieme completamente specificato di 20 elementi ground.
Questo metodo viene poi richiamato dove necessario, come riportato ad esempio nel metodo `testEq_ChiusoGrossoGround_Vuoto()`.

Le restanti 7 stringhe speciali mostrate nel capitolo 4.1.3 sono tradotte in codice Java in maniera del tutto analoga.

5.12 test12: relazioni di molti elementi

Questo test set ha lo scopo di testare gli analoghi casi del test11 nel caso di vincoli su `LRel`.
Viene di seguito riportato il file di specifica Prolog.

`test12_prolog_data.txt`

```
$$$ user rules $$$

$$$ args $$$
vuota${}${}${}
var$A$B$C

chiusaGrossaGround$#LRelFullySpecifiedGround_20$***$***
chiusaPiccolaNotGround$***${[X1, X2]}${[X3, X4]}

aperta${[X3,Y3]/R1}${[X4,Y4]/R2}${[X5,Y5]/R3}

$$$ constraints $$$
comp$3$prefix$comp
inv$2$prefix$inv
```

5.13 Considerazioni sui tempi di esecuzione

Visto l'utilizzo di insiemi/relazioni di grandi dimensioni in test11 e test12, si è sfruttato questo fatto per attivare la stampa dei tempi e trarre qualche conclusione dai dati forniti in output sul file `times.txt`.

Prima di effettuare questa analisi nel caso di grandi dimensioni, verranno analizzati i tempi nel caso più semplice di insiemi ridotti.
Per ottenere dati significativi, è necessario effettuare l'esecuzione delle classi Java almeno un paio di volte: questo perché ovviamente i tempi potrebbero oscillare di qualche millisecondo, e più esecuzioni si effettuano, maggiore sarà l'accuratezza con cui si potrà stimare il tempo medio d'esecuzione dei test della classe.

5.13.1 test1 - insiemi completamente specificati (piccoli)

Per il caso di insiemi di dimensioni ridotte si è scelto di analizzare i tempi di esecuzione dei vincoli del test1.
Attivando la stampa dei tempi, il contenuto del file `times.txt` dopo qualche esecuzione è il seguente.

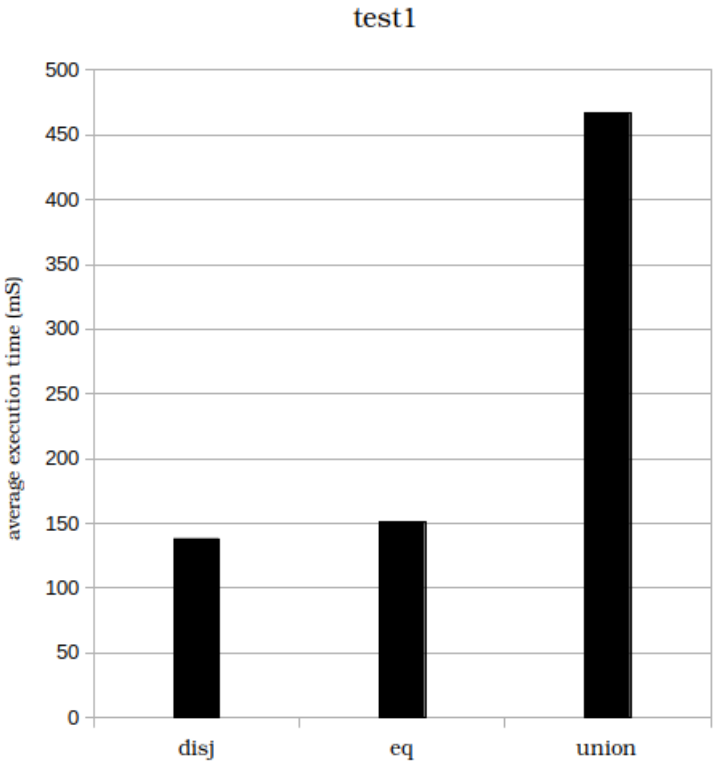
```
Disj 138
Disj 100
Disj 131
Disj 119
Disj 114
Disj 119

Eq 151
Eq 213
Eq 220
Eq 183
Eq 211
Eq 186

Union 467
Union 541
Union 465
Union 544
Union 468
Union 488
```

Come evidente, sono state fatte 6 esecuzioni per ognuna delle 3 classi Java.
I risultati sottolineano il fatto che il vincolo `union` rappresenta il vero collo di bottiglia: la risoluzione di tale vincolo è infatti per il solver più onerosa rispetto alla risoluzione degli altri due, anche a causa dell'arità più alta.

La rappresentazione ad istogramma di seguito riportata rende più chiara la visualizzazione di questi dati, evidenziando quali siano i valori medi dei tempi di esecuzione in millisecondi.



Questa discrepanza diventa ancora più evidente aumentando le dimensioni degli insiemi, come illustrato nel prossimo paragrafo.

5.13.2 test11 - insiemi completamente specificati (grandi)

Nel test11 vengono testati i medesimi vincoli analizzati nel test1, inserendo però tra gli argomenti del test set un insieme completamente specificato di 20 elementi ground.

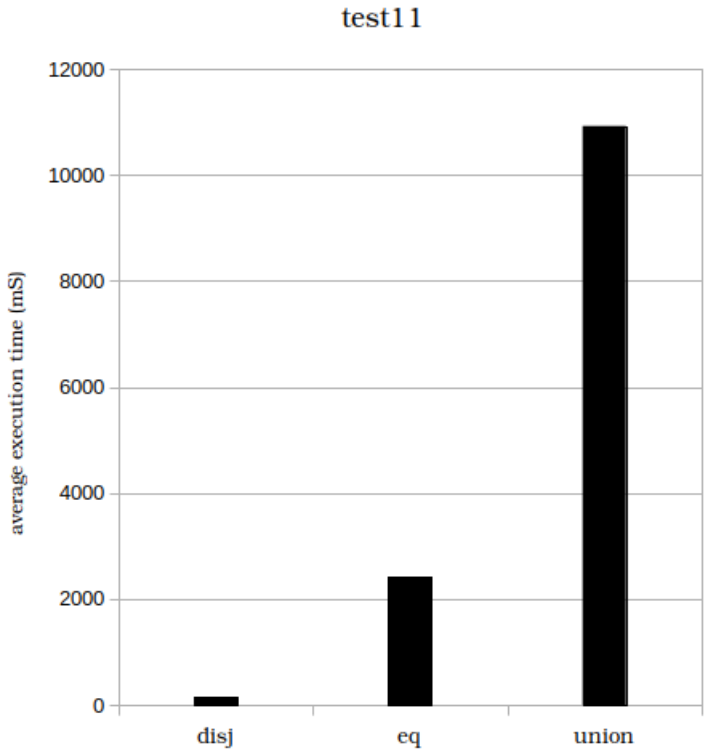
I risultati riportati dal file `times.txt` sono i seguenti:

```
Disj 162
Disj 154
Disj 154
Disj 151
Disj 145
Disj 138

Eq 2286
Eq 2367
Eq 2620
Eq 2031
Eq 2875
Eq 2308

Union 10371
Union 10645
Union 11310
Union 10645
Union 10921
Union 11645
```

Come evidente, i tempi sono molto simili per `disj`, più alti per `eq` e decisamente più alti per `union`, che impiega ora addirittura 10-11 secondi.



5.13.3 test6 - relazioni completamente specificate (piccole)

Per quanto riguarda le relazioni, si è pensato di procedere in modo analogo a quanto fatto per gli insiemi: un'analisi dei tempi in caso di relazioni di piccole dimensioni, e un'analisi nel caso di relazioni di grandi dimensioni.

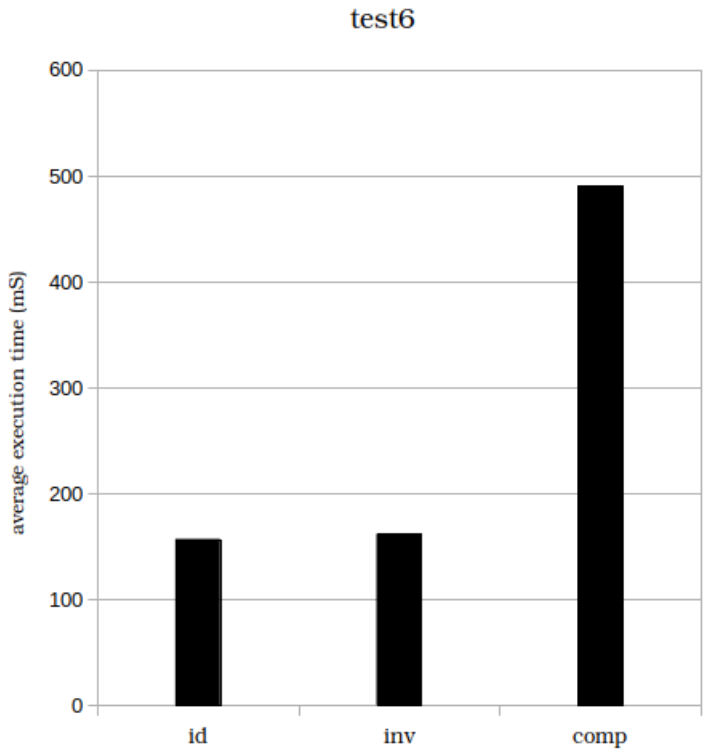
Per il primo caso, si è scelto di attivare i tempi in test6, contenente i vincoli `id`, `inv` e `comp`, con i seguenti risultati:

```
Id 171
Id 182
Id 162
Id 116
Id 131
Id 180

Inv 152
```

```
Inv 150
Inv 145
Inv 205
Inv 151
Inv 168

Comp 467
Comp 563
Comp 417
Comp 490
Comp 504
Comp 501
Comp 329
```



Come previsto, anche in questo test set il collo di bottiglia è rappresentato dal vincolo più oneroso da risolvere, ossia `comp`, con risultati molto simili a quelli di `union` in test1.

5.13.4 test12 - relazioni completamente specificate (grandi)

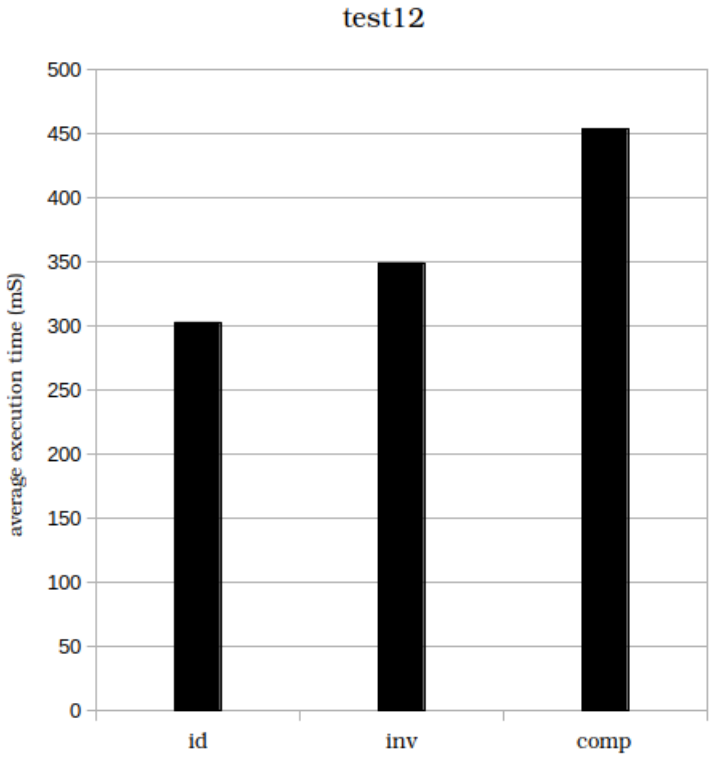
Per il caso di relazioni grandi, si è scelto invece di testare i vincoli presenti in test12.
Per fare un confronto equo con test11, si sono scelte relazioni completamente specificate di 20 elementi ground, come appunto per test11.

I risultati sono i seguenti:

```
Id 306
Id 297
Id 316
Id 294
Id 307
Id 298

Inv 358
Inv 289
Inv 404
Inv 337
Inv 317
Inv 391

Comp 389
Comp 410
Comp 439
Comp 500
Comp 487
Comp 497
```



Come si evince dai dati riportati, il vincolo `comp`, pur avendo arità 3, riesce a scalare molto meglio rispetto al vincolo `union`: i tempi restano infatti pressoché identici rispetto a quelli calcolati in test6.

5.13.5 test11(bis) - insiemi parzialmente specificati (grandi)

Come ultima analisi sui tempi d'esecuzione, si è scelto di utilizzare test11 con una lieve modifica: piuttosto che utilizzare insiemi completamente specificati di elementi ground, si è analizzato il caso di insiemi parzialmente specificati di elementi non ground, ottenuti tramite la sintassi:

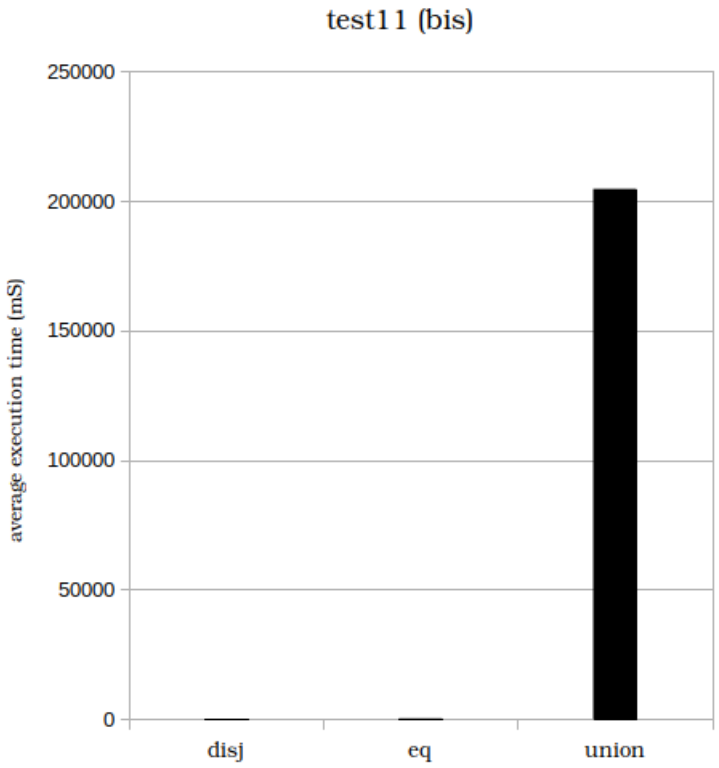
```
#LSetPartiallySpecifiedNotGround_20
```

I risultati sono i seguenti.

```
Disj 106
Disj 178
Disj 109
Disj 110
Disj 139
Disj 101

Eq 364
Eq 261
Eq 249
Eq 228
Eq 281
Eq 302

Union 204543
Union 212628
Union 200513
Union 197747
Union 210112
Union 202228
```



Come evidente, `eq` e `disj` non risentono assolutamente di questo cambiamento.
Il vincolo `union` invece, impiega notevolmente di più (circa 200 secondi, più di 3 minuti).
Da questo dato è facile dedurre come la gestione di insiemi parzialmente specificati sia più problematica per il solver rispetto alla gestione di insiemi completamente specificati.

6 Conclusioni e lavori futuri

In questo lavoro di tesi è stato affrontato il problema del testing dei vincoli della libreria Java JSetL. Il problema è stato affrontato realizzando uno strumento software ad hoc avente il compito di generare i test cases in maniera automatica. La realizzazione dello strumento ha quindi permesso di avere a disposizione un tool dalla duplice funzione.
Un primo aspetto utile è dato dalla possibilità di testare la correttezza di qualsiasi vincolo presente nella libreria (anche eventualmente di vincoli che verranno sviluppati in futuro e quindi non esistenti al momento della realizzazione dello strumento).
Un secondo aspetto utile è dato dalla possibilità di effettuare dei benchmark sistematici sull'esecuzione dei test set generati. Ciò permette di evidenziare quali vincoli riescano a scalare meglio all'aumentare delle dimensioni dei parametri: i dati ricavati dall'analisi di questi tempi forniscono un valido suggerimento che può indicare dove sia più opportuno iniziare a lavorare nel caso si voglia rendere la libreria nel suo complesso sempre più efficiente.

Per quanto riguarda eventuali sviluppi futuri, vi sono almeno due aspetti non trattati da questa tesi su cui si potrebbe lavorare.
Sicuramente il multi-threading è ancora tutto da esplorare: lo strumento genera al momento classi Java che sfruttano l'approccio di default di JUnit 4, ossia quello di un unico thread master il quale esegue sequenzialmente i diversi metodi di test. A partire dalla versione 5.3 di JUnit tuttavia, è possibile eseguire più metodi di test in parallelo tra loro (sebbene la feature, al momento della scrittura di questa tesi, sia ancora in fase sperimentale).
Si potrebbe quindi fare un confronto tra esecuzione sequenziale e parallela, capire se i vantaggi sono realmente tangibili oppure trascurabili, e preoccuparsi delle problematiche di data race tipiche di un approccio parallelo.

Un ulteriore miglioramento possibile potrebbe consistere nella creazione di un tool che sia in grado di visualizzare graficamente i tempi di esecuzione.
Questo ipotetico programma, ricevuto in input il file `times.txt`, potrebbe generare un istogramma in maniera automatica e, in base a soglie temporali preimpostate dall'utente, generare un report in cui vengano descritti quali sono quei vincoli che fungono da collo di bottiglia per il test set.

Ringraziamenti

Sono diverse le persone che mi hanno accompagnato fino al raggiungimento di questo importante traguardo, e mi sembra doveroso dedicare loro una pagina di questo lavoro.

Anzitutto desidero ringraziare il Prof. Gianfranco Rossi per avermi seguito durante tutto il mio percorso di tirocinio e di tesi, per essere sempre stato disponibile a chiarire ogni mio dubbio e per avermi sempre fatto sentire a mio agio.

Ringrazio di cuore i miei colleghi più stretti con cui ho condiviso questi tre anni:
Luca, per le nostre memorabili sessioni di studio e per tutta la sua disponibilità.
Massimo e Davide, per avermi sempre supportato e, soprattutto, sopportato.
Federico, per essermi sempre stato vicino quando ne ho avuto bisogno.
Un grazie speciale va inoltre ai miei amici di vecchia data Sebastiano e Luca, per tutti i bei momenti passati insieme.

Desidero infine ringraziare la mia famiglia: i miei genitori, per avermi dato l'opportunità di arrivare sino a questo punto senza mai farmi mancare nulla, e mia sorella, per aver sempre fatto il tifo per me.
A loro dedico questa tesi.

Riferimenti bibliografici

[1] Gianfranco Rossi, Elio Panegai, Elisabetta Poleo
JSetL: a Java library for supporting declarative programming in Java
Software Practice & Experience 2007; 37:115-149.

[2] Gianfranco Rossi, Roberto Amadini, Andrea Fois
JSetL User's Manual 3.0
July 4th, 2019
`http://www.clpset.unipr.it/jsetl/jsetl-3-0-manual.pdf`

[3] Ian Sommerville
Software Engineering
Pearson Education, 2011

[4] *JUnit Home Page*
`https://junit.org/junit4/`

[5] Pianfetti Maurizio
Strumenti per la generazione automatica di test strutturali e funzionali
Tesi di Laurea Magistrale in Ingegneria Informatica, Università di Genova,
22 Marzo 2013

[6] Cobianchi Michael
Trattamento di vincoli su insiemi e relazioni binarie nella libreria Java JSetL
Tesi di Laurea in Informatica, Università di Parma,
20 Dicembre 2018

[7] Fois Andrea
Estensioni ed uso degli Insiemi Intensionali Ristretti in JSetL
Tesi di Laurea in Informatica, Università di Parma,
12 Luglio 2018