UNIVERSITÀ DEGLI STUDI DI PARMA
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica

Contributo alla Specifica JSR-331 mediante un'Implementazione basata su JSetL

Relatore:
Chiar.mo Prof. Federico Bergenti

Candidato:
Fabio Biselli

Anno Accademico 2010/2011

Ai miei genitori,
Fausta e Romualdo

Indice

Introduzione

La *programmazione a vincoli* (*Constraint Programming* o *CP*) è un paradigma che offre strumenti per modellare e risolvere efficacemente problemi di soddisfacimento ed ottimizzazione con vincoli. Questa consiste nell'integrazione di vincoli all'interno di un linguaggio che funge da host. I primi linguaggi usati a tale scopo furono linguaggi di tipo logico, il più famoso è indubbiamente Prolog. Tali linguaggi differiscono fortemente da quelli imperativi come C, C++ o Java, poiché permettono al programmatore di specificare cosa fare e non come farlo.

In questo lavoro di tesi ci si occuperà di implementare delle specifiche che consentono di fornire al linguaggio Java alcuni vantaggi del paradigma dei linguaggi logici mediante l'implementazione dello standard JSR-331.

Java è un software multipiattaforma, ovvero progettato per funzionare su macchine che possono essere di diversa natura. Tale piattaforma ha come caratteristica peculiare il fatto di rendere possibile la scrittura e l'esecuzione di applicazioni che siano indipendenti dall'hardware sul quale poi sono eseguite.

Di fatto, la portabilità, è quell'aspetto del linguaggio che ne ha decretato il grande successo, insieme all'approccio object-oriented che consente un forte riuso del codice. Tali peculiarità sono valorizzate dall'introduzione delle API Java (Application Programming Interface), una collezione di componenti software già scritti e pronti all'uso.

Lo sviluppo delle API è delegato ad una comunità di sviluppo aperta, essendo il codice di Java open-source, denominata *JCP* (*Java Community Process*) che detiene la responsabilità dello sviluppo della tecnologia Java guidando l'implementazione e l'approvazione delle specifiche tecniche del linguaggio. Tali specifiche vengono richieste e descritte mediante i *JSR*.

Con l'acronimo *JSR* (*Java Specification Requests*) si indicano le descrizioni proposte o finali per le specifiche della piattaforma Java. In qualsiasi momento vi sono più processi di revisione ed approvazione in corso, ogni processo viene denominato con la sigla JSR seguita da un numero identificativo (vedi [1]).

Il lavoro di tesi svolto si riferisce alle proposte relative al JSR-331 in merito alla standardizzazione della programmazione a vincoli. Partendo dalle specifiche richieste da tale documento si è realizzata un'interfaccia

basata sulla libreria JSetL e sviluppata dal Dipartimento di Matematica dell'Università degli Studi di Parma.

Il lavoro di tesi è strutturato nel seguente modo:

Il Capitolo 1 introduce brevemente il concetto di programmazione con vincoli e di soluzione di un problema, si fornisce quindi un semplice esempio.

Il Capitolo 2 è dedicato al JSR-331, nel quale si riassumono i concetti principali del documento di specifica per quanto riguarda gli obiettivi e la struttura dello standard proposto.

Il Capitolo 3 descrive brevemente le funzionalità offerte dal solver JSetL per quanto riguarda i costrutti utilizzati per la realizzazione dell'implementazione concreta delle specifiche.

Il Capitolo 4 riguarda l'implementazione vera e propria della parte relativa alla definizione del problema. Ogni sezione del capitolo è incentrata su un concetto ed una classe specifica, nella quale verrà introdotta l'interfaccia, la classe realizzata e l'eventuale classe astratta o implementazione di base, ove utilizzata.

Il Capitolo 5, analogamente al precedente, descrive nel dettaglio l'implementazione della parte relativa alla risoluzione del problema.

Nel Capitolo 6 si darà spazio alle conclusioni del lavoro svolto e ai possibili lavori futuri.

Infine due appendici approfondiranno due aspetti affrontati durante lo sviluppo. L'appendice A riguarda un vincolo la cui trattazione non è stata banale, uno specifico vincolo di cardinalità richiesto dalla specifica. L'appendice B invece riguarda i test svolti ed i relativi risultati.

Capitolo 1

Programmazione a Vincoli

Come accennato nell'introduzione, la programmazione a vincoli offre un approccio dichiarativo al programmatore, consentendogli di specificare relazioni tra variabili sotto forma di vincoli. Oggigiorno tale paradigma rappresenta una provata tecnica di ottimizzazione e molti solver basati su di esso potenziano applicazioni presenti nei settori di pianificazione, configurazione, allocazione di risorse e supporto a decisioni in tempo reale. Tuttavia l'assenza di una standardizzazione continua a limitarne l'accettazione nel mondo dell'industria, ed è per ovviare a questa limitazione che è nata la specifica JSR-331.

Ma che cos'è effettivamente la programmazione a vincoli? Di cosa si occupa in pratica? In questo capitolo si darà una definizione formale di problema di soddisfacimento di vincoli e si vedranno i metodi più comuni di risoluzione. Si vedrà quindi un semplice esempio.

1.1 Definizione formale

Un *problema di soddisfacimento di vincoli* (o *CSP*) è definito da un insieme di variabili e da un insieme di vincoli. Ogni variabile ha un dominio non vuoto di possibili valori. Ogni vincolo coinvolge un sottoinsieme delle variabili e ne specifica le combinazioni di valori possibili.

Definizione 1.1. Un *problema di soddisfacimento di vincoli* (o CSP) è una tripla $\mathbf{P} = \langle \mathbf{V}, \mathbf{D}, \mathbf{C} \rangle$ tale che, per ogni $n > 0$ si ha che:

- $\mathbf{V} = \{X_1, X_2, \ldots, X_n\}$ è un insieme di variabili.
- $\mathbf{D} = D_1 \times D_2 \times \cdots \times D_n$ è una n-upla di domini tale che $D_i = \mathrm{dom}(X_i)$.
- $\mathbf{C} = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m\}$ è un insieme di vincoli di *arità* al più n definito su un sottoinsieme di $\mathbf{V}$.

Si ha quindi che se un vincolo $\mathcal{C}_i$ ha arità k allora esiste un insieme di indici $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ tale che $\mathcal{C}_i \subseteq D_{i_1} \times \cdots \times D_{i_k}$.

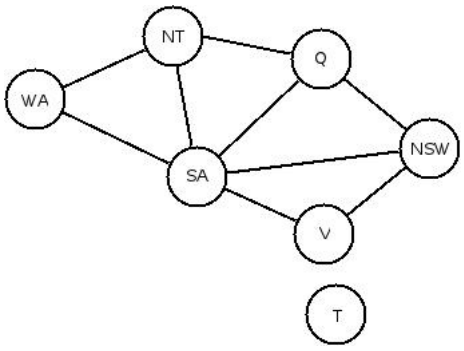

Figura 1.1: Esempio di grafo, colorazione della mappa australiana.

Definizione 1.2. Siano $v_1 \in D_1, \ldots, v_n \in D_n$ valori nei rispettivi domini, uno *stato* $\mathcal{S}$ del problema **P** è definito dall'assegnamento di valori ad alcune o a tutte le variabili del problema:

$$\mathcal{S} = \{X_i = v_i, X_j = v_j, \ldots\}\,.$$

Un assegnamento che non viola nessun vincolo è detto *consistente* o *legale*. Un assegnamento che coinvolge tutte le variabili del problema è detto *completo*.

Definizione 1.3. Una *soluzione* del problema è un assegnamento completo e consistente.

Alcuni CSP richiedono anche che la soluzione massimizzi una *funzione obiettivo*.

1.2 Risoluzione di un CSP

Per capire in cosa consiste la soluzione di un CSP può essere utile visualizzarlo come un *grafo di vincoli*, i nodi rappresentano le variabili del problema e gli archi corrispondono ai vincoli.

Esprimere un problema sotto forma di CSP apporta molti importanti benefici. Dato che la rappresentazione degli stati è conforme ad un modello accettato, consistendo sempre in un insieme di variabili con i loro vincoli e domini, la funzione successore ed il test obiettivo possono essere scritti in un modo generico che si applica ad ogni CSP. Inoltre è possibile sviluppare euristiche efficaci che non richiedano esperienza aggiuntiva del dominio.

Si può dare una *formulazione incrementale* di un CSP come se fosse un problema di ricerca:

- **Stato iniziale**: l'assegnamento vuoto {} nel quale nessuna variabile ha un valore assegnato.
- **Funzione successore**: assegna un valore ad una variabile in modo che non violi alcun vincolo ad essa associato.

- **Test obiettivo**: verifica che l'assegnamento sia completo.

- **Costo di cammino**: costante per ogni passo.

Poiché una soluzione deve essere un assegnamento completo avrà almeno profondità n, dove n è il numero di variabili coinvolte. Per questo, per risolvere i CSP sono molto utilizzati gli algoritmi di ricerca in profondità.

È quindi possibile definire una *formulazione a stato completo* del problema, nella quale ogni stato è un assegnamento completo che può soddisfare i vincoli oppure no. Questo approccio è utilizzato negli algoritmi di ricerca locale, nei quali viene assegnato un valore ad ogni variabile per lo stato iniziale e normalmente la funzione successore cambia il valore di una variabile alla volta.

1.3 Un semplice esempio: colorazione di una mappa

Si è definito formalmente un problema di soddisfacimento di vincoli e si è quindi spiegato brevemente cosa significa risolverlo. A questo punto vediamo un classico esempio: la colorazione di mappe geografiche.

La colorazione delle regioni di uno stato su una mappa geografica può essere facilmente espresso mediante un CSP, prendiamo come esempio la mappa dell'Australia (esempio trattato in [4]).

1.3.1 Definizione del problema

Innanzitutto occorre stabilire qual'è il problema. Colorare le regioni di un paese in modo che non si confondano significa che quelle adiacenti devono avere colori differenti. Supponiamo di avere a disposizione solo tre colori: rosso, verde e blu. Le regioni dell'Australia sono sette: Western Australia, Northern Territory, South Australia, Queensland, New South Wales, Victoria e l'isola della Tasmania. In figura 1.2 si notano le varie regioni sotto forma di nodi del grafo dei vincoli, gli archi rappresentano la relazione di confine tra regioni.

A questo punto abbiamo tutti gli elementi per ottenere un problema $\mathbf{P} = \langle \mathbf{V}, \mathbf{D}, \mathbf{C} \rangle$ tale che:

$$\mathbf{V} = \{W_A, N_T, S_A, Q, N_{SW}, V, T\},$$

$$\mathbf{D} = \{\text{rosso}, \text{verde}, \text{blu}\},$$

in cui i vincoli appartenenti a $\mathbf{C}$, descritti dagli archi del grafo, sono del tipo:

$$\mathcal{C}_1 \leftarrow W_A \neq N_T, \quad \mathcal{C}_2 \leftarrow W_A \neq S_A, \quad \ldots$$

1.3.2 Dalla definizione formale alla realizzazione in Java mediante JSR-331

Come già accennato nella sezione precedente la definizione formale di un CSP e la sua soluzione sono concetti abbastanza generici da poter essere applicati a qualunque sistema di risolutori o linguaggio di programmazione.

Tuttavia la sintassi può essere molto differente tra un linguaggio e l'altro, ma anche tra un risolutore e un altro scritti nel medesimo linguaggio, la specifica JSR-331 si prefigge di standardizzare tale sintassi mediante la definizione di un'interfaccia. Vediamo come può essere rappresentato il problema dell'Australia.

Listato 1.1: `Australia.java`.

```
public class Australia {
  static final String[] colors = { "red", "green", "blue"};
...
}
```

Viene creata la classe `Australia` in cui verrà definito il problema e la chiamata al risolutore di vincoli. L'array di stringhe rappresenta il dominio delle variabili che verranno create.

Listato 1.2: definizione delle variabili.

```
static public void main(String[] argv) {
  try {
    Problem p = new Problem("Australia");
    // Variabili.
    int n = colors.length-1;
    Var WA   = p.variable("Western Australia",0, n);
    Var NT   = p.variable("Northern Territory", 0, n);
    Var SA   = p.variable("South Australia",0, n);
    Var Q    = p.variable("Queensland",0, n);
    Var NSW  = p.variable("New South Wales",0, n);
    Var V    = p.variable("Victoria",0, n);
    Var T    = p.variable("Tasmania",0, n);
```

In questa parte di codice viene creato il problema `p` chiamato "Australia" e vengono quindi create le variabili legate al problema (costruite dal metodo `variable(nome, min, max)` della classe `Problem`).

Si noti che il dominio delle variabili è specificato nella chiamata del metodo `variable`, ovvero ogni variabile ha dominio $[0, n-1]$, in cui n è la lunghezza del vettore dei domini. Le variabili si riferiscono in pratica agli indici di tale vettore.

Listato 1.3: definizione dei vincoli.

```
    // Vincoli.
    p.post(WA, "!=", NT);
```

```
    p.post(WA, "!=", SA);
    p.post(NT, "!=", SA);
    p.post(NT, "!=", Q);
    p.post(SA, "!=", Q);
    p.post(SA, "!=", NSW);
    p.post(SA, "!=", V);
    p.post(Q, "!=", NSW);
    p.post(V, "!=", NSW);
```

La definizione dei vincoli avviene mediante il metodo `post(Var, String, Var)`, sempre invocato sul problema. In questo caso vengono specificati tutti i vincoli di disuguaglianza come specificato in figura 1.2.

Listato 1.4: ricerca della soluzione.

```
Solution solution = p.getSolver().findSolution();
if (solution != null) {
  for (int i = 0; i < p.getVars().length; i++) {
    Var var = p.getVars()[i];
    p.log(var.getName() + " - " +
    colors[solution.getValue(var.getName())]);
  }
}
else
  p.log("no solution found");
```

Dopo aver trovato la soluzione, viene stampata mediante un semplice algoritmo (che ovviamente non fa parte dello standard) e questo è il risultato ottenuto con JSetL:

Listato 1.5: la soluzione.

```
Western Australia - red
Northern Territory - green
South Australia - blue
Queensland - red
New South Wales - green
Victoria - red
Tasmania - red
```

Capitolo 2

Java Specification Request 331

JSR-331 è una *richiesta per specifiche* Java in fase di sviluppo sotto le regole redatte dal Java Community Process. Queste specifiche definiscono le API per la programmazione a vincoli.

La specifica JSR-331 risponde alla necessità di ridurre i costi associati all'incorporazione di risolutori di vincoli (come JSetL, Choco, etc.) ad applicazioni commerciali e non, operanti nel mondo reale. Esiste già un certo numero di fornitori di queste API, come già accennato possiamo ricordare JSetL, Choco, JaCoP ed altri. Tuttavia le differenze tra questi sono abbastanza significative da causare gravi difficoltà di utilizzo per gli sviluppatori di software.

2.1 Obiettivi del JSR-331

La standardizzazione della programmazione a vincoli si prefigge come scopo quello di rendere tale tecnologia più accessibile per gli sviluppatori. Avendo un'interfaccia unificata sarà possibile, per i programmatori, modellare il problema in modo tale da poter provare la soluzione con più CP solver. Questo minimizza la dipendenza da fornitori specifici, ma allo stesso tempo non limita la possibilità di quest'ultimi nel procedere con lo sviluppo del solver.

Gli obiettivi delle specifiche sono:

- facilitare l'inserimento della tecnologia basata sui vincoli nelle applicazioni Java;
- aumentare la comunicazione e la standardizzazione tra i vari fornitori di CP solver;
- incoraggiare il mercato delle applicazioni basate sulla programmazione a vincoli e dei suoi strumenti mediante la standardizzazione delle suddette API;

- facilitare l'integrazione di tecniche basate sui vincoli in altri JSR per supportare la programmazione dichiarativa;

- rendere le applicazioni Java più portabili tra vari fornitori di risolutori di vincoli;

- fornire un modello per l'implementazione ed il supporto di librerie di vincoli e strategie di ricerca per diverse applicazioni basate sulla programmazione a vincoli;

- supportare i fornitori di solver offrendo API che vadano incontro alle loro necessità e che siano di facile implementazione.

2.1.1 A chi è rivolta?

La specifica è rivolta principalmente a tre soggetti:

- aziende che utilizzano le CP API per sviluppare applicazioni di supporto a decisioni in ambito industriale;

- fornitori di risolutori di vincoli che intendono sviluppare e mantenere la propria implementazione delle CP API;

- ricercatori in ambito di programmazione a vincoli che vogliono fornire o arricchire librerie di vincoli standard, algoritmi di ricerca e problemi concreti che vengano mantenuti dalla CP community.

2.1.2 Scopo della specifica

Lo scopo della specifica JSR-331 è di definire un'interfaccia semplice da utilizzare, leggera e che costituisca uno standard per acquisire ed utilizzare risolutori di vincoli.

La specifica è mirata a piattaforme basate su Java ed è compatibile con JDK 1.5 o successivi.

L'ambito della specifica segue un approccio minimalista, con particolare cura alla facilità d'uso. Ricopre i più comuni concetti dei problemi con vincoli e della loro rappresentazione che è ormai già diventata una standardizzazione di fatto, adottata dai solver e negli articoli scientifici. Tale ambito è allo stesso tempo sufficientemente ampio da permettere agli sviluppatori di applicazioni l'utilizzo delle interfacce standard per modellare e risolvere tipici problemi di soddisfacimento di vincoli, all'interno dei domini più comuni in ambito aziendale, come la pianificazione, l'allocazione delle risorse e la configurazione.

I seguenti concetti ed oggetti rappresentano l'ambito iniziale delle specifiche:

- variabili vincolate dei tipi più utilizzati: interi, booleani, reali ed insiemi di interi;
- vincoli unari e binari ed espressioni definite con variabili vincolate;
- i più comuni vincoli globali;
- la possibilità di ottenere una soluzione, ogni soluzione oppure una soluzione ottimale, sotto certi limiti definiti dall'utente.

Ci si aspetta che le API vengano ampliate e così è anche specificato come aggiungere nuovi concetti, funzionalità e strategie relative ai CP nel momento in cui queste vengono comunemente adottate.

JSR-331 è focalizzato solo sull'interfaccia, per quanto riguarda le implementazioni non assume nessun approccio particolare. I seguenti concetti ed oggetti sono prerogativa esclusiva dei vari solver e strumenti specifici:

- meccanismi d'implementazione per i vari domini;
- implementazione dei più comuni vincoli binari e globali;
- meccanismi di propagazione di vincoli;
- backtracking e meccanismi di reversibilità.

2.2 Struttura

JSR-331 prescrive un insieme di operazioni fondamentali per definire e risolvere problemi di soddisfacimento di vincoli e problemi di ottimizzazione. La struttura consiste in tre principali componenti:

- specifiche (CP API);
- implementazione basata su differenti CP solver;
- *TCK* (*Tecnology Compatibility Kit*), un pacchetto di test, strumenti e documentazione che verrà utilizzato per verificare la conformità alle specifiche delle varie implementazioni.

Ogni implementazione del JSR-331 può implementare l'interfaccia direttamente oppure può estendere le classi della *common implementation* fornite dalle API. La common implementation può essenzialmente semplificare l'implementazione del JSR-331. Nel momento in cui lo standard diverrà più maturo e più implementazioni adotteranno vincoli e strategie di ricerca comuni, queste verranno gradualmente aggiunte alle librerie della common implementation.

2.2.1 Specifiche

Le specifiche per JSR-331 consistono in:

- un'interfaccia pura (`javax.constraints`);
- la common implementation (`javax.constraints.impl`).

- `javax.constraints`

  Interfacce Java pure, come `Problem` o `Solver`, che specificano i maggiori concetti e metodi per definire e risolvere problemi di soddisfacimento ed ottimizzazione di vincoli.

- `javax.constraints.impl`

  Classi Java come `AbstractProblem` e `AbstractSolver` che implementano (parzialmente o completamente) definizioni di problemi, concetti di risoluzione e metodi che non dipendono direttamente da uno specifico CP solver.

2.2.2 Implementazioni per JSR-331

Ogni implementazione della specifica JSR-331 è basata su un CP solver (come ad esempio JSetL) e deve fornire le definizioni per tutte le interfacce presenti nel pacchetto `javax.constraints`. Alcune classi possono essere implementate direttamente dall'interfaccia standard, altre si possono ottenere estendendo l'implementazione comune fornita dal pacchetto `javax.constraints.impl`.

JSR-331 richiede che ogni implementazione fornisca almeno la definizione delle classi presenti nei seguenti pacchetti:

- `javax.constraints.impl`

  Classi Java come `Problem` o `Solver` che forniscono una implementazione finale per la definizione di problemi, concetti e metodi. Queste classi possono essere derivate dall'implementazione comune.

- `javax.constraints.impl.constraint`

  Una libreria di vincoli che contiene le implementazioni di vincoli basilari e globali che sono di fatto basati sulle varie implementazioni dei CP solver.

- `javax.constraints.impl.search`

  Una libreria di strategie di ricerca che contiene le implementazioni basate sui solver concreti.

In aggiunta ogni implementazione può fornire propri vincoli (nativi) e proprie strategie di ricerca, assumendo che questi seguano le interfacce standard `javax.constraints.Constraint` e `javax.constraints.SearchStrategy`.

Il fatto che ogni implementazione per JSR-331 debba adottare gli stessi nomi per i package, per le principali classi e metodi permetterà agli sviluppatori di applicazioni di passare facilmente da un solver ad un altro (differenti implementazioni) senza dover cambiare nulla nel proprio codice. Possono di fatto scrivere applicazioni specifiche, motori basati su vincoli una sola volta utilizzando le CP API e quindi utilizzare i differenti solver cambiando solo i file `.jar` all'interno del classpath.

Nota. La possibilità di cambiare i solver senza modificare il codice delle applicazioni ha comunque una limitazione: fissando per convenzione i nomi dei package si ha come effetto collaterale quello di non poter unire le funzionalità di due o più solver differenti. La scelta di una determinata implementazione sottostante è definita dal solo file `.jar` nel classpath dell'applicazione.

2.2.3 Tecnology Compatibility Kit

Il *TCK* (*Tecnology Compatibility Kit*) è un pacchetto di documentazione, test e strumenti utilizzati per testare la correttezza delle implementazioni per le specifiche di JSR-331.

Il TCK consiste in due package:

- `org.jcp.jsr331.tests`

  Contiene i moduli JUnit che permettono all'utente di validare automaticamente la correttezza dell'implementazione con la specifica JSR-331.

- `org.jcp.jsr331.samples`

  Contiene esempi di CSP che forniscono test integrati per i più comuni vincoli e strategie di ricerca incluse in JSR-331, mostrano l'utilizzo della programmazione a vincoli per problemi reali.

Non tutti i concetti introdotti nel package `javax.constraints` sono richiesti per l'implementazione del solver al fine di essere validato. Il pacchetto `org.jcp.jsr331.tests` contiene solo quei test che sono normativi per la specifica (devono essere soddisfatti da ogni implementazione al fine di essere validate). Il file `AllTests.java` all'interno del package contiene tutti i test normativi.

Il pacchetto `javax.constraints.impl` fornisce le implementazioni di base per alcune interfacce e metodi opzionali. Ci sono due tipi di implementazioni opzionali:

1. implementazioni di default (non le più efficienti) che possono essere sovrascritte da una particolare implementazione per JSR-331;

2. semplici frammenti di codice che lanciano un'eccezione a runtime e che informano l'utente che quel metodo non è stato implementato dalla particolare implementazione.

Nota. Il package `org.jcp.jsr331.samples` è presente a soli fini dimostrativi e non tutto ciò che è ivi incluso deve essere supportato da ogni implementazione.

2.2.4 Modello di sviluppo

Il modello di sviluppo per le applicazioni finali che utilizzeranno le API JSR-331 richiederà che i seguenti file `.jar` siano inclusi nel classpath:

- `jsr331.jar`: include tutte le classi e le interfacce standard;

- `jsr331.<solver>.jar`: include tutte le classi della specifica implementazione;

- `<solver>.jar`: include tutte le classi del solver utilizzato su cui si basa l'implementazione.

Ad esempio lo sviluppo basato su JSetL richiederà:

- `jsr331.jar`

- `jsr331.jsetl.jar`

- `jsetl.jar`

2.3 Rappresentazione di un CSP

Come si è evidenziato nel capitolo 1 un CSP è definito da un insieme di variabili con il relativo dominio ed un insieme di vincoli; a queste variabili viene quindi ristretto il dominio per trovare una o più soluzioni.

JSR-331 definisce tutti i concetti Java per la *rappresentazione* e la *risoluzione* di un problema di soddisfacimento di vincoli. Suddivide quindi in maniera naturale un CSP in due parti fondamentali:

- la **definizione** del problema, rappresentata dall'interfaccia `Problem`;

- la **soluzione** del problema, rappresentata dall'interfaccia `Solver`.

Ogni concetto fondamentale di un CSP appartiene ad una delle due categorie, a livello utente è possibile presentare un CSP mediante la seguente suddivisione:

1. Problema:
   (a) Variabili vincolate.
   (b) Vincoli.
2. Risoluzione:
   (c) Strategia.
   (d) Soluzione.

Ogni CP solver utilizza nomi e rappresentazioni differenti per questi concetti, che comunque sono semanticamente invarianti per la maggior parte di essi. JSR-331 fornisce una nomenclatura unificata e specifiche dettagliate.

La definizione di un problema non conosce nulla a riguardo della sua risoluzione. Un'istanza della classe `Problem` può quindi esistere a prescindere dall'esistenza di un'istanza di `Solver`. Non vale il viceversa, poiché un'istanza della classe `Solver` può essere creata solo partendo da uno specifico problema. Durante la ricerca di una soluzione il solver può cambiare lo stato del problema (modifica dei domini delle variabili, semplificazione dei vincoli, etc). È quindi responsabilità degli specifici risolutori mantenere (o meno) i possibili stati del problema in base alla strategia di ricerca utilizzata.

Capitolo 3

La Libreria JSetL

In questo capitolo verrà descritta brevemente la libreria JSetL, soffermandosi principalmente sull'utilizzo, ovvero sui costrutti e i metodi utilizzati nell'implementazione concreta di JSR-331.

JSetL è una libreria Java che combina il paradigma di programmazione orientato agli oggetti con i concetti dei linguaggi *CLP* (*Constraint Logic Programming*), come variabili logiche, elenchi, unificazione, risoluzione di vincoli, non determinismo.

Tra le caratteristiche di interesse per l'implementazione JSR-331 troviamo le variabili logiche (`IntLVar` e `SetLVar`), i vincoli (`Constraint`) ed il meccanismo di risoluzione dei vincoli (`SolverClass`).

3.1 Variabili logiche

JSetL supporta la nozione di variabile logica come quella che si trova in Logica e nei linguaggi di programmazione funzionale. Le variabili logiche possono essere sia *inizializzate* che *non inizializzate*. Il valore di una variabile logica in JSetL può essere di qualsiasi tipo. Un valore può essere assegnato ad una variabile logica (non inizializzata) come risultato dell'elaborazione dei vincoli che la coinvolgono.

Variabile logica.
Una *variabile logica* è un'istanza della classe `LVar`, creata dalla seguente chiamata:

LVar varName = **new** LVar(ExtVarName, VarValue);

dove `varName` è il nome della variabile, `ExtVarName` è il nome opzionale esterno e `VarValue` è un parametro opzionale che ne rappresenta il valore.

Il nome esterno è una stringa che può essere utile quando occorre stampare la variabile ed i possibili vincoli che la coinvolgono.

Nota. Durante la fase di sviluppo dell'implementazione la stampa è stata una parte fondamentale per il debugging, sia per quanto riguarda le prove d'esecuzione che per i test con JUnit e con il TCK. Ovviamente il nome esterno non definisce una variabile (essendo opzionale) e sarebbe possibile creare variabili con il medesimo `ExtVarName` ma indipendenti. Tuttavia, per facilitare il riconoscimento all'occhio umano è buona norma definire nomi significativi, per quanto riguarda i nomi delle variabili temporanee interne si rimanda alla sezione 4.2.

3.2 Variabili logiche intere

Le variabili logiche intere sono quelle d'interesse al fine di implementare le classi `Var` e `VarBool` per l'attuale implementazione JSR-331.

In JSetL le variabili intere sono rappresentate dalla classe `IntLVar`, che di fatto estende la classe `LVar` ereditandone metodi e costruttori.

Variabile logica intera.
Una *variabile logica intera* è un'istanza della classe `IntLVar`, creata dalla seguente chiamata:

IntLVar varName = **new** IntLVar(name);

dove `varName` è il nome della variabile e `name` è il nome esterno opzionale.

3.2.1 Dominio

Ogni variabile logica intera ha un dominio ad essa associato, nel caso di `IntLVar` il dominio è rappresentato da un multi-intervallo di interi.

Il concetto di multi-intervallo è definito in [2] e, in breve, è costituito dall'unione disgiunta di intervalli di interi. Ogni volta che la variabile in questione viene modificata tramite vincoli che la coinvolgono il relativo dominio viene modificato di conseguenza.

3.2.2 Metodi utili

La classe `IntLVar` fornisce alcuni metodi di utilità, quelli più utilizzati nell'implementazione sono:

- `MultiInterval getDomain()`: restituisce il dominio della variabile intera.
- `Boolean isBound()`: restituisce `true` se il dominio è ristretto ad un solo elemento, `false` in caso contrario.

3.2.3 Operazioni aritmetiche

È possibile creare variabili vincolate intere anche partendo da una variabile d'invocazione (`this`), mediante le comuni operazioni aritmetiche. Siano X_0 la variabile d'invocazione `x0`, $\mathcal{C}_0$ i vincoli ad essa associati, X_1 la variabile data dal parametro `x1`, $\mathcal{C}_1$ i vincoli ad essa associati, X_2 la nuova variabile `x2` ottenuta dalle seguenti operazioni:

- `IntLVar sum(Integer k)`: crea una variabile logica intera vincolata `x2` tale che se `x2 = x0.sum(k)`, allora:

$$X_2 = X_0 + k \wedge \mathcal{C}_0.$$

- `IntLVar sum(IntLVar x1)`: crea una variabile logica intera vincolata tale che se `x2 = x0.sum(x1)`, allora:

$$X_2 = X_0 + X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `IntLVar sub(Integer k)`: crea una variabile logica intera vincolata tale che se `x2 = x0.sub(k)`, allora:

$$X_2 = X_0 - k \wedge \mathcal{C}_0.$$

- `IntLVar sub(IntLVar x1)`: crea una variabile logica intera vincolata tale che se `x2 = x0.sub(x1)`, allora:

$$X_2 = X_0 - X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `IntLVar mul(Integer k)`: crea una variabile logica intera vincolata tale che se `x2 = x0.mul(k)`, allora:

$$X_2 = X_0 \cdot k \wedge \mathcal{C}_0.$$

- `IntLVar mul(IntLVar x1)`: crea una variabile logica intera vincolata tale che se `x2 = x0.mul(x1)`, allora:

$$X_2 = X_0 \cdot X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `IntLVar div(Integer k)`: crea una variabile logica intera vincolata tale che se `x2 = x0.div(k)`, allora:

$$X_2 = \frac{X_0}{k} \wedge k \neq 0 \wedge \mathcal{C}_0.$$

- `IntLVar div(IntLVar x1)`: crea una variabile logica intera vincolata tale che se `x2 = x0.div(x1)`, allora:

$$X_2 = \frac{X_0}{X_1} \wedge X_1 \neq 0 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

3.2.4 Vincoli

La classe `IntLVar` fornisce metodi per la creazione dei comuni vincoli che rappresentano le relazione su interi ($<$, $=$, $\neq$, ...). Inoltre fornisce altri vincoli comunemente usati come *AllDifferent*, vincoli di dominio e di appartenenza.

Di seguito si elencano e descrivono brevemente i vincoli utilizzati nell'implementazione, siano X_0 la variabile d'invocazione `this`, $\mathcal{C}_0$ i vincoli ad essa associati, X_1 la variabile data dal parametro `x1` e $\mathcal{C}_1$ i relativi vincoli associati:

- `Constraint eq(Integer k)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 = k \wedge \mathcal{C}_0.$$

- `Constraint eq(IntLVar x1)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 = X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `Constraint neq(Integer k)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 \neq k \wedge \mathcal{C}_0.$$

- `Constraint neq(IntLVar x1)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 \neq X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `Constraint le(Integer k)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 \leq k \wedge \mathcal{C}_0.$$

- `Constraint le(IntLVar x1)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 \leq X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `Constraint lt(Integer k)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 < k \wedge \mathcal{C}_0.$$

- `Constraint lt(IntLVar x1)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 < X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `Constraint ge(Integer k)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 \geq k \wedge \mathcal{C}_0.$$

- `Constraint ge(IntLVar x1)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 \geq X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

- `Constraint gt(Integer k)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 > k \wedge \mathcal{C}_0.$$

- `Constraint gt(IntLVar x1)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow X_0 > X_1 \wedge \mathcal{C}_0 \wedge \mathcal{C}_1.$$

Sia `vars` un vettore di $n+1$ variabili logiche intere tale che X_0 rappresenti `vars[0]`, X_1 rappresenti `vars[1]`, etc. Il vincolo "AllDifferent" è descrivibile come:

- `Constraint AllDifferent(IntLVar[] vars)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

  $$\mathcal{C} \leftarrow X_0 \neq X_1 \wedge X_0 \neq X_2 \wedge \cdots \wedge X_0 \neq X_n \wedge X_1 \neq X_2 \wedge \cdots \wedge X_1 \neq X_n \wedge \cdots$$

  è possibile scrivere in modo più compatto il vincolo:

$$\mathcal{C} \leftarrow \left(\bigwedge_{0 \leq i < j \leq n} X_i \neq X_j \right) \wedge \bigwedge_{i=0}^{n} \mathcal{C}_i.$$

3.3 Variabili logiche su insiemi di interi

Le variabili logiche di interesse al fine di implementare la classe `VarSet` per l'attuale implementazione JSR-331 sono quelle di insiemi di interi.

In JSetL le variabili di insiemi di interi sono rappresentate dalla classe `SetLVar`, che estende la classe `LVar` ereditandone metodi e costruttori.

Variabile logica su insiemi di interi. Una variabile logica su *insiemi di interi* (più brevemente un *insieme di interi*) è un'istanza di `SetLVar`, creata dalla seguente chiamata:

SetLVar varName = **new** SetLVar(name);

dove `varName` è il nome della variabile e `name` è il nome opzionale esterno.

3.3.1 Dominio

Il dominio di un `SetLVar` è rappresentato da un'istanza di `SetInterval` che di fatto è un reticolo* di insiemi di interi (`Set<Integer>`).

Il dominio di una variabile di insiemi di interi può essere specificato alla creazione della variabile, ed è modificato automaticamente quando vengono risolti i vincoli ad essa associati.

Quando il dominio di una variabile si restringe fino ad essere rappresentato da un singoletto, la variabile viene detta *bound*, se invece il dominio si restringe all'insieme vuoto significa che i vincoli ad essa associati sono insoddisfacibili.

Sia `s` un'istanza della classe `MultiInterval` che rappresenta un insieme di interi S, la variabile logica che ha come dominio S è creata dalla seguente chiamata:

SetLVar varName = **new** SetLVar(s);

3.3.2 Metodi utili

La classe `SetLVar` fornisce alcuni metodi utili ereditati dalla classe `LVar`, adattati per supportare i tipi di ritorno `MultiInterval`, `SetLVar` e per i vincoli su insiemi e domini.

- `Constraints getConstraints()`: restituisce la congiunzione di tutti i vincoli associati alla variabile;
- `MultiInterval getDomain()`: restituisce il dominio della variabile;
- `Boolean isBound()`: restituisce `true` se il dominio è ristretto ad un solo elemento, `false` in caso contrario.

3.3.3 Operazioni insiemistiche

Come nel caso delle variabili intere, è possibile creare nuove variabili insiemistiche utilizzando metodi che riproducono le comuni operazioni sugli insiemi. Sia S_0 l'insieme rappresentante la variabile d'invocazione `s`, $\mathcal{C}_0$ i vincoli ad essa associati, siano S_1, S_2 gli insiemi rappresentanti i parametri `SetLVar x` e `y`, $\mathcal{C}_1$ e $\mathcal{C}_2$ i vincoli ad esse associati, si definiscono le seguenti operazioni:

- `SetLVar compl()`: crea un nuovo insieme di interi tale che, se `x = s.compl()` allora:
$$S_1 = S_0^c \wedge \neg\mathcal{C}_0;$$

*Un reticolo è un insieme parzialmente ordinato in cui ogni coppia di elementi ha sia un estremo inferiore che un estremo superiore.

- `SetLVar intersect(SetLVar y)`: crea un nuovo insieme di interi tale che, se `x = s.intersect(y)` allora:

$$S_1 = S_0 \cap S_2 \wedge \mathcal{C}_0 \wedge \mathcal{C}_2;$$

- `SetLVar union(SetLVar y)`: crea un nuovo insieme di interi tale che, se `x = s.union(y)` allora:

$$S_1 = S_0 \cup S_2 \wedge \mathcal{C}_0 \vee \mathcal{C}_2;$$

- `SetLVar diff(SetLVar y)`: crea un nuovo insieme di interi tale che, se `x = s.diff(y)` allora:

$$S_1 = S_0 \setminus S_2 \wedge \mathcal{C}_0 \wedge \neg\mathcal{C}_2.$$

3.4 Vincoli

JSetL fornisce vincoli per specificare condizioni su variabili logiche e insiemi. Questi vincoli sono manipolati da un *risolutore di vincoli* (un'istanza della classe `SolverClass`) che implementa le specifiche strategie di risoluzione.

Il *dominio dei vincoli* in JSetL è il dominio $\mathcal{SET}$ definito in [5], esteso con alcuni nuovi vincoli per la comparazione tra interi.

Vincolo atomico.
Un *vincolo atomico* in JSetL è un'espressione in una delle seguenti forme:

- e_1.`op`(e_2);
- e_1.`op`(e_2, e_3);

dove `op` è uno dei metodi predefiniti (`eq, neq, in, nin`, ...) ed e_1, e_2, e_3 sono espressioni il cui tipo dipende da `op`.

Il significato di questi metodi è associato in modo naturale al loro stesso nome, ovvero `eq` e `neq` stanno ad indicare l'uguaglianza e la disuguaglianza rispettivamente (dall'inglese equal e not equal), `in` e `nin` per l'appartenenza e la non appartenenza, etc.

Vincolo.
Un *vincolo* è un vincolo atomico o (ricorsivamente) un'espressione del tipo:

$$c_1.\texttt{and}(c_2.\texttt{and}(\cdots c_{n-1}.\texttt{and}(c_n)\cdots)$$

oppure

$$c_1.\texttt{and}(c_2).\ldots.\texttt{and}(c_n)$$

dove $c_1, c_2, \ldots, c_n$ sono vincoli atomici.

Il significato di c_1.`and`(c_2).....`and`(c_n) è la congiunzione logica $\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \cdots \wedge \mathcal{C}_n$, in cui i $\mathcal{C}_i$ sono le espressioni logiche rappresentanti i vincoli atomici c_i.

Constraint.
Un *Constraint* JSetL è un'istanza della classe `Constraint`, creata dalla seguente chiamata:

Constraint constraintName = **new** Constraint();

dove `constraintName` è il nome del vincolo.

3.4.1 Operazioni logiche

È possibile creare vincoli anche partendo da un vincolo d'invocazione (`this`), mediante le comuni operazioni logiche. Siano $\mathcal{C}_1$, $\mathcal{C}_2$ i vincoli rappresentanti le istanze dei `Constraint c1` e `c2`. Si definiscono le seguenti operazioni logiche:

- `Constraint c = c1.and(c2)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow \mathcal{C}_1 \wedge \mathcal{C}_2;$$

- `Constraint c = c1.or(c2)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow \mathcal{C}_1 \vee \mathcal{C}_2;$$

- `Constraint c = c1.orTest(c2)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow \mathcal{C}_1 \vee \mathcal{C}_2;$$

- `Constraint c = c1.notTest()`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow \neg\mathcal{C}_1;$$

- `Constraint c = c1.impliesTest(c2)`: crea un nuovo vincolo $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow \mathcal{C}_1 \Rightarrow \mathcal{C}_2.$$

Il vincoli `or` viene risolto in modo non deterministico[†], mentre `and` e i vincoli i cui nomi finiscono con la sottostringa "`Test`" (`impliesTest` ad esempio) vengono risolti in modo determistico[‡].

[†]Per risoluzione non deterministica si intende che il vincolo crea un punto di scelta copiando il contenuto del constraint store. In pratica aprendo più computazioni parallele.

[‡]Viceversa il metodo deterministico non crea punti di scelta e si limita a propagare il vincolo finchè non viene effettuato il labeling sulle variabili è un metodo più efficiente, ma non garantisce una soluzione senza labeling.

3.5 Meccanismi di risoluzione

I vincoli creati utilizzando la classe `Constraint` o mediante i metodi propri delle variabili logiche vengono passati ad un solver, un'istanza della classe `SolverClass`, che li aggiunge al proprio *constraint store*, il quale contiene tutti i vincoli attivi.

La risoluzione dei vincoli è attuata mediante la chiamata di uno dei metodi della classe `SolverClass` come `solve()` o `nextSolution()`.

3.5.1 Constraint store

Il `constraint store` di un'istanza del solver contiene tutti i vincoli attivi per il programma in esecuzione. JSetL fornisce alcuni metodi con i quali aggiungere nuovi vincoli ad uno specifico constraint store, visualizzarne il contenuto o rimuovere tutti i vincoli presenti.

Per aggiungere un vincolo `c` al constraint store `S` è sufficiente utilizzare il metodo `add` nel seguente modo:

```
S.add(c);
```

La collezione dei vincoli nello store è considerata come una congiunzione di vincoli atomici, se Γ è il contenuto del constraint store del solver `S`, con la chiamata `S.add(c)` si ottiene il vincolo $\Gamma \wedge \mathcal{C}$, dove $\mathcal{C}$ rappresenta `c`.

Esempio. Siano `x`, `y` e `z` variabili logiche, sia `solver` un'istanza della classe `SolverClass`.

```
solver.add(x.eq(3));      // x = 3
solver.add(y.neg(x));     // y != x
solver.add(x.eq(z));      // x = z
```

Il vincolo aggiunto allo store è $x = 3 \wedge y \neq x \wedge x = z$. Il medesimo risultato si sarebbe potuto ottenere con la seguente chiamata:

```
solver.add(x.eq(3).and(y.neq(x)).and(x.eq(z)));
```

3.5.2 Risoluzione dei vincoli

La risoluzione dei vincoli in JSetL è basata sulla riduzione di ogni congiunzione di vincoli atomici ad una forma semplificata denominata *solved form*.

Senza entrare nel dettaglio, si può affermare che una solved form è una congiunzione di vincoli irriducibili e che, se non vuota, rappresenta una soluzione per i vincoli aggiunti allo store.

Il metodo (inteso come funzione Java) principale per ottenere una soluzione da un'istanza `solver` di `SolverClass` è `solve`, il quale lancia un'eccezione se i vincoli non sono soddisfacibili, altrimenti trasforma il constraint store in modo non deterministico affinchè i vincoli assumano una forma semplificata.

3.5.3 Labeling

Come spesso si verifica nei risolutori di vincoli su domini finiti, il solver di JSetL non può considerarsi *completo* nel momento in cui i vincoli del constraint store coinvolgono istanze di `IntLVar` o `SetLVar`. Per controllare la soddisfacibilità dello store e per trovare una o tutte le soluzioni, in questo caso, può rendersi necessario introdurre strategie di ricerca, una di queste è il *labeling*.

Il labeling si effettua su variabili logiche, alle quali viene cercato di assegnare un valore del proprio dominio. Ovviamente considerare ogni possibile valore del dominio risulterebbe molto gravoso in termini di computazione ed efficienza, sono state introdotte quindi delle euristiche per ridurre lo spazio di ricerca:

- *Scelta delle variabili*: determina in quale ordine vengono scelte le variabili a cui si cerca di assegnare un valore.

- *Scelta dei valori*: determina quale valore assegnare alle variabili.

Di seguito si elencano le possibili euristiche attualmente presenti in JSetL:

Listato 3.1: euristiche dei valori.

```
public enum ValHeuristic {
        GLB,              // da sinistra a destra.
        LUB,              // da destra a sinistra.
        MID_MOST,         // dal centro.
        MEDIAN,           // dal valore medio.
        EQUI_RANDOM,      // metodi casuali.
        RANGE_RANDOM,
        MID_RANDOM
}
```

Listato 3.2: euristiche delle variabili.

```
public enum VarHeuristic {
        RIGHT_MOST,    // da destra a sinistra.
        LEFT_MOST,     // da sinistra a destra.
        MID_MOST,      // dal centro.
        MIN,           // dal dominio con valore minimo.
        MAX,           // dal dominio con valore massimo.
        FIRST_FAIL,    // dal dominio minimo.
        RANDOM         // casuale.
}
```

Per applicare un'euristica occorre aggiungerla sotto forma di vincolo all'interno del constraint store. Siano `x`, `y` variabili logiche, sia `vars` un array di variabili logiche (omogenee), le sintassi possibili sono le seguenti:

Listato 3.3: opzioni di labeling.

```
solver.add(x.label());

solver.add(y.label(ValHeuristic.LUB));

LabelingOptions lop = new LabelingOptions();
lop.val = ValHeuristic.MEDIAN;
lop.var = VarHeuristic.MID_MOST;
solver.add(label(vars, lop));
```

Come si vede è possibile utilizzare il metodo `label()` senza parametri, che utilizza un'euristica di default, oppure è possibile specificare un metodo per le scelte mediante la chiamata con parametro `label(ValHeuristic)`. L'ultima opzione è quella di utilizzare una struttura di supporto chiamata `LabelingOptions`, che consente di specificare entrambe le scelte per un array di variabili (ma è anche disponibile il metodo per una variabile singola).

Capitolo 4

Implementazione: Rappresentazione del Problema

In questo capitolo verranno descritte le classi Java implementate per la specifica JSR-331 nell'ambito della definizione di un problema.

Per ogni classe descritta verrà introdotta l'interfaccia fornita dalla specifica e quindi, con maggior dettaglio, l'implementazione che riguarda il solver JSetL.

Definizione del problema

Nella specifica JSR-331 la definizione del problema utilizza le seguenti interfacce:

- `Problem`;
- `Var`;
- `VarBool`;
- `VarReal`;
- `VarSet`;
- `Constraint`.

Nelle prossime sezioni verranno descritte le suddette classi (implementazioni) con i principali metodi, tra queste non è implementata la classe `VarReal` poiché JSetL non supporta, allo stato attuale del progetto, vincoli su variabili logiche reali.

4.1 Interfaccia `Problem`

La specifica prevede una generica interfaccia `Problem` che permetta agli utenti di creare ed accedere alle comuni entità del CSP. Un `problem`

funziona come una *factory* per la creazione di variabili e vincoli. Ogni variabile ed ogni vincolo appartengono ad uno ed un solo problema, ad esempio:

```
Problem p = ProblemFactory.newProblem("Test");
Var x = p.variable("X",1,10);
```

crea un'istanza `p` della classe `Problem` (definita da una specifica implementazione, JSetL nel nostro caso) e quindi, tramite `p`, viene creata una nuova variabile vincolata `x` di dominio $[1, 10]$ e con un nome esterno `X`. Il dominio di `x` è quindi composto da ogni intero tra 1 e 10, senza omissioni. La variabile è automaticamente aggiunta al problema.

4.1.1 Creare variabili

Tutti i metodi per creare variabili iniziano con la parola "`variable`", la nuova variabile creata con tale metodo viene automaticamente aggiunta al problema, ovvero viene inserita nel vettore delle variabili definito nel problema astratto (come vedremo in seguito) o nell'implementazione specifica JSetL.

Un metodo alternativo per creare variabili è quello di utilizzare un costruttore della classe `Var` (o `VarBool`, `VarSet`, ...) definito nella relativa implementazione, ad esempio:

```
Problem p = ProblemFactory.newProblem("Test");
Var x = new Var(p,"X",1,10);
```

in questo caso la variabile creata sarà sempre relativa al problema `p`, ma non verrà aggiunta alla lista delle variabili del problema, in pratica è una variabile di supporto. Per aggiungere la variabile al problema in un momento successivo è possibile usare il metodo `add`.

4.1.2 Creare ed aggiungere vincoli

Tutti i metodi per creare—ed aggiungere—vincoli iniziano con la parola "`post`". Anche in questo caso il vincolo creato viene automaticamente aggiunto alla lista dei vincoli del problema.

```
Problem p = ProblemFactory.newProblem("Test");
Var x = p.variable("X",1,10);
Var y = new Var(p,"Y",1,10);

p.post(x,"<",y);   // x < y.
```

In questo semplice esempio viene creata una variabile `x` legata al problema ed una di supporto `y`, viene quindi aggiunto al problema il vincolo $\texttt{x} < \texttt{y}$. Quando verrà generata una soluzione per `p` sia `x` che `y` dovranno soddisfare

il vincolo, ma ad esempio alla stampa della soluzione solo `x` verrà presa in considerazione.

Come per la creazione delle variabili anche i vincoli possono essere creati senza l'utilizzo di metodi factory dell'interfaccia `Problem`, utilizzando un costruttore della classe `Constraint` oppure di classi più specifiche che specializzano la classe `Constraint` (vedi sezione 4.8).

4.1.3 Metodi di uso generale

L'interfaccia `Problem` specifica anche metodi generici per la stampa, per ottenere la versione, il solver ed altri. Se ne elencano alcuni.

- **public** String getImplVersion()

  Restituisce la versione corrente dell'implementazione concreta di JSR-331, JSetL nel nostro caso.

- **public** Solver getSolver()

  Restituisce un'istanza del `Solver` associato al problema di invocazione che verrà utilizzato per risolvere il problema. Se un solver non è già definito questo metodo ne crea uno nuovo e lo associa al problema.

- **public void** log(String text)

  Stampa il testo passato come parametro sul display di default (come definito nell'implementazione).

- **public** Var scalProd(**int**[] values, Var[] vars)

  Crea una nuova variabile (`Var`) vincolata ad essere il prodotto scalare dell'array di valori interi e delle variabili date.

- **public** Var element(**int**[] values, Var indexVar)

  Crea una nuova variabile vincolata che sia un elemento dell'array `values` con un indice definito da una variabile vincolara `indexVar`.

4.2 Classe `Problem`

La classe `Problem` implementa l'interfaccia standard `Problem` estendendo la classe `AbstractProblem` dell'implementazione comune.

```
public class Problem extends AbstractProblem {
```

Questo approccio permette di ereditare tutti i metodi astratti puri da implementare e, dove necessario, è possibile ridefinire i metodi non astratti.

4.2.1 Classe AbstractProblem

`AbstractProblem` è una classe astratta fornita dall'implementazione comune (`javax.constraints.impl`) che implementa l'interfaccia `Problem`:

```
abstract public class AbstractProblem implements Problem {
```

definendo ogni metodo ed attributo di utilità generica. Non è una classe astratta pura, poiché fornisce molte implementazioni di base, sia per gli attributi che per i metodi.

Nota. La classe `Problem` definita sopra non è chiaramente quella che implementa la classe astratta, la prima infatti è per esteso la classe:

`javax.constraints.impl.Problem`,

che è parte dell'implementazione, mentre la seconda é parte delle interfacce:

`javax.constraints.Problem`.

Attributi

Tra gli attributi più importanti si evidenziano:

```
String name;
ArrayList<Var> vars;
ArrayList<VarBool> varBools;
ArrayList<Constraint> constraints;
Solver solver;
```

La stringa `name` rappresenta il nome del problema, `vars` è la lista delle variabili logiche intere inserite nel problema, analogamente `varBools` è la lista delle variabili booleane e `constraints` la lista dei vincoli del problema.

Come si può notare nell'implementazione di base mancano le liste per `VarReal` e `VarSet`. Per quanto riguarda le variabili reali, allo stato attuale dello sviluppo di JSR-331 non sono supportate, mentre per le variabili insiemistiche viene fornita un'implementazione che sfrutta le variabili intere. JSetL tuttavia fornisce direttamente il supporto alle variabili insiemistiche e quindi si è naturalmente deciso di non sfruttare l'implementazione comune fornita per queste variabili, come si vedrà più avanti nel capitolo.

Metodi di uso generale

I metodi di utilità generica o che non hanno bisogno di un'implementazione specifica come quella fornita da JSetL o Choco, vengono implementati direttamente all'interno della classe `AbstractProblem`. Tra quelli più utilizzati si evidenziano:

- **public** Var add(Var var)

  Aggiunge la variabile intera passata come parametro alle variabili del problema e restituisce la variabile stessa.

- **public** Var add(VarBool var)

  Aggiunge la variabile booleana passata come parametro alle variabili del problema e restituisce la variabile stessa.

- **public void** remove(String name)

  Rimuove la variabile passata come parametro dalle variabili del problema.

- **public** Var[] variableArray(String name, **int** min, **int** max, **int** size)

  Crea ed aggiunge al problema un array di variabili intere il cui nome è specificato dalla stringa `name` a cui è concatenato l'indice della variabile ("`name`-i"). Il dominio di ogni variabile è $[\min, \max]$ e la dimensione dell'array è definita dal parametro `size`.

- **public** Var[] getVars()

  Restituisce l'array di variabili intere associate al problema.

Altri metodi definiti nella classe astratta vanno comunque ridefiniti nell'implementazione, poiché rappresentano funzionalità specifiche dei solver concreti. Tra questi metodi si elencano:

- **public** VarSet variableSet(String name, **int** min, **int** max)

  Crea una variabile insiemistica di interi e la aggiunge al problema;

- **public** Constraint postAllDifferent(Var[] vars)

  Dato un vettore di variabili intere, le vincola ad essere tutte differenti;

- **public** Constraint postCardinality(Var[] vars, Var cardVar, String oper, Var var)

  Crea un particolare vincolo di cardinalità tra l'array di variabili passate come parametro e una data relazione.

4.2.2 Implementazione

Una volta definita la classe, come visto all'inizio della sezione 4.2, sono stati inseriti tutti i metodi astratti dell'interfaccia `Problem` non definiti nell'implementazione di base.

Si descrivono ora gli attributi ed i metodi definiti, basati sul solver JSetL.

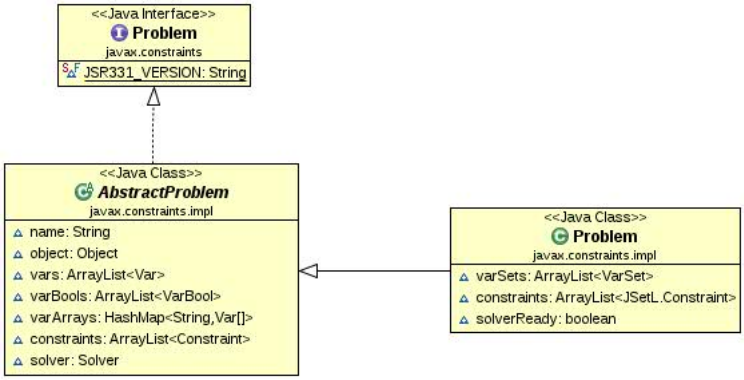

Figura 4.1: Class Diagram di `Problem`.

Attributi

Come accennato in precedenza, l'implementazione di base fornisce attributi di classe per la gestione di variabili intere, booleane e per i vincoli, ma non prevede nulla per quelle insiemistiche. Sono stati quindi aggiunti, sulla base del medesimo meccanismo di `AbstractProblem`, attributi per la gestione di queste variabili.

Listato 4.1: attributi di `Problem`.

```
ArrayList<VarSet> varSets;

private static final int OPER_EQ = 1, OPER_UNKNOWN = 0,
    OPER_NEQ = 2, OPER_LT = 3, OPER_LEQ = 4, OPER_GT = 5,
    OPER_GEQ = 6;

private static int counter = 0;
```

L'attributo `varSets` rappresenta la lista delle variabili insiemistiche appartenenti al problema.

Sono state poi definite delle costanti utili per identificare gli operatori ($=$, $\leq$, $\geq$, ...) mediante il costrutto `switch` (che richiede come parametro un intero) e delle variabili globali utilizzate per contare variabili interne al fine di assegnare un nome univoco alle stesse.

Costruttori

I costruttori richiesti sono due: uno senza parametro ed uno con parametro `String` che ne rappresenta il nome.

Listato 4.2: costruttore con parametro.

```
public Problem(String name) {
        super(name);
```

```
        varSets = new ArrayList<VarSet>();
}
```

In entrambi i casi viene chiamato il costruttore della classe astratta di base con la sola differenza che quello senza parametro assegna un valore di default al nome:

```
        super("_P"+(counter++));
```

e quindi viene inizializzata la lista delle variabili proprie della classe, ovvero quelle insiemistiche che si sono ridefinite nella nostra implementazione.

Metodo `postElement`

L'interfaccia `Problem` specifica anche metodi per creare vincoli che hanno a che fare con elementi di array di interi o di variabili vincolate intere. Se una variabile logica intera `indexVar` ha la funzione di indice all'interno di un array `v`, il risultato di un'operazione `v[indexvar]` è un'altra variabile vincolata. Poiché Java non supporta l'overloading dell'operatore "`[]`" l'interfaccia standard utilizza questo metodo.

Vi sono quattro possibili definizioni di questo metodo:

```
  public Constraint postElement(int[], Var, String, int);
  public Constraint postElement(int[], Var, String, Var);
  public Constraint postElement(Var[], Var, String, int);
  public Constraint postElement(Var[], Var, String, Var);
```

Vediamo nel dettaglio solo uno di questi metodi poiché in tutti e quattro i casi si utilizza la classe di supporto `Element` che rappresenta il vincolo specificato mediante i parametri del costruttore e che sarà definito in 4.8.2.

Listato 4.3: `postElement`, array di interi.

```
public Constraint postElement(
                int[] array,
                javax.constraints.Var indexVar,
                String oper,
                int value) {
        if (indexVar.getMin() > array.length - 1 || indexVar
            .getMax() < 0)
                throw new RuntimeException("elementAt: 
                    invalid index variable");
        Element result = new Element(array, indexVar, oper,
            value);
        post(result);
        return result;
}
```

Innanzitutto viene controllato che il dominio della variabile indice sia compatibile con il range dell'array, altrimenti viene lanciata un'eccezione.

Viene quindi istanziata la variabile `result` di tipo `Element` con i medesimi parametri del metodo invocato. Questa variabile rappresenta il vincolo:

array[indexVar] oper value,

che viene infine aggiunto al problema mediante il metodo `post`.

Metodo `post`

Un vincolo non ha effetto fino a che non viene inserito nel problema. Per fare ciò si utilizza il metodo `post`, che ha le seguenti dichiarazioni:

```
public Constraint post(int[], Var[], String, int);
public Constraint post(int[], Var[], String, Var);
public Constraint post(Var[], String, int);
public Constraint post(Var[], String, Var);
public Constraint post(Var, String, int);
public Constraint post(Var, String, Var);
public Constraint post(Constraint);
```

Ogni metodo `post` ha il medesimo approccio: costruisce il vincolo basandosi sul metodo `linear` (si entrerà nel dettaglio nei prossimi paragrafi), lo pubblica nel solver JSetL mediante la chiamata della specifica `post(Constraint)` ed infine lo aggiunge ai vincoli del problema con il metodo `add`.

Listato 4.4: `post`.

```
public Constraint post(
                int[] array,
                javax.constraints.Var[] vars,
                String oper,
                int value) {
        Constraint result = linear(array, vars, oper, value)
            ;
        post(result);
        add(result);
        return result;
}

public Constraint post(
                Var var,
                String oper,
                int value) {
        Constraint result = linear(var, oper, value);
        post(result);
        add(result);
        return result;
}
```

Nel listato 4.4 si possono notare due metodi `post` standard, uno per le variabili singole (il secondo) ed uno per array di variabili. Entrambi utilizzano il metodo `linear` e restituiscono il nuovo vincolo creato.

Listato 4.5: `post(Constraint)`.

```
public void post(javax.constraints.Constraint constraint) {
        add(constraint);
        addJSetLConstraints((JSetL.Constraint) constraint.
            getImpl());
}
```

Questo è invece il metodo su cui tutti gli altri si basano per aggiungere il vincolo creato al `Problem`. Viene invocata la funzione `add` della classe base per aggiungere il vincolo JSR-331. Quindi mediante il metodo `addJSetLConstraints` si inserisce il vincolo specifico di JSetL all'array di supporto.

Anche la classe `Constraint` ha un metodo `post`, per cui è possibile creare un vincolo e quindi renderlo attivo con la seguente sintassi:

```
Constraint c = p.linear(x, ``<='', y);
c.post();
```

Metodi `linear`

I vincoli che hanno a che fare con il confronto di espressioni vincolate utilizzano gli operatori di confronto standard:

Stringa	Semantica
<	minore stretto,
<=	minore o uguale,
=	uguale,
>=	maggiore o uguale,
>	maggiore stretto,
!=	diverso.

I metodi che implementano il confronto sono chiamati `linear` e vengono usati, come accennato in precedenza, nelle definizioni di `post`. È anche possibile per l'utente utilizzare direttamente `linear`, non sono infatti metodi privati, e sono utili perché non aggiungono direttamente un vincolo e le variabili al problema.

Vi sono quattro differenti metodi `linear`:

```
public Constraint linear(int[], Var[], String, int);
public Constraint linear(int[], Var[], String, Var);
public Constraint linear(Var, String, int);
public Constraint linear(Var, String, Var);
```

i primi due hanno a che fare con vettori di variabili vincolate, mentre gli altri ne gestiscono una sola. In entrambi i casi si effettua il confronto con un intero o un'altra variabile vincolata.

Listato 4.6: `linear`, con array di variabili.

```
public Constraint linear(
                int[] array,
                javax.constraints.Var[] vars,
                String oper, int value) {
        if (array.length != vars.length || array.length ==
            0)
                throw new RuntimeException(
                        "Coefficent and variable length must
                            be equal and not zero.");
        Var scalprod = (Var) scalProd(array, vars);
        return new Linear(scalprod, oper, value);
}
```

Il metodo con i vettori, dopo aver verificato che le lunghezze siano compatibili e non nulle, genera il prodotto scalare tra i due array tale che se $v_0, v_1, \ldots, v_{n-1}$ sono le variabili contenute in `vars` e $a_0, a_1, \ldots, a_{n-1}$ sono i coefficienti interi contenuti in `array`, la variabile vincolata s rappresentata da `scalprod` è data da:

$$s = a_0 \cdot v_0 + a_1 \cdot v_1 + \cdots + a_{n-1} \cdot v_{n-1}.$$

Una volta creata la variabile temporanea `scalprod` viene calcolato e ritornato il vincolo mediante il costruttore della classe `Linear`.

Listato 4.7: `linear`, con variabile singola.

```
public Constraint linear(
                javax.constraints.Var var,
                String oper,
                int value) {
        if (var == null)
                throw new RuntimeException("Parameters must
                    not be null.");
        return new Linear(var, oper, value);
}
```

Nel metodo con la singola variabile viene controllato che `var` non sia nulla, nel qual caso viene lanciata un'eccezione, anche quindi viene ritornata la costruzione di un nuovo vincolo specifico di tipo `Linear`, discusso nella sezione 4.8.1.

Metodo `scalProd`

Questo metodo pubblico permette di creare una nuova variabile vincolata intera che rappresenta il prodotto scalare tra gli elementi degli array passati

come parametro. Come si è già visto per il metodo `linear` il risultato che si ottiene è del tipo:

$$s = a_0 \cdot v_0 + a_1 \cdot v_1 + \cdots + a_{n-1} \cdot v_{n-1}.$$

`scalProd` è spesso utilizzato all'interno di altri metodi, ma è comunque definito pubblicamente poiché rappresenta un'operazione comune per le variabili.

Listato 4.8: `scalProd`.

```
public Var scalProd(int[] values, Var[] vars) {
  .
  .
  IntLVar[] intVars = new IntLVar[vars.length];
  if (values[0] != 0)
    intVars[0] = ((Var) vars[0]).getIntLVar().mul(values[0])
        ;
  else intVars[0] = new IntLVar(0,0);
  for (int i = 1; i < values.length; i++) {
    if (values[i] !=0) {
      IntLVar tmp = new IntLVar(((Var) vars[i]).getIntLVar()
          .mul(values[i]));
      intVars[i] = intVars[i-1].sum(tmp);
    }
    else intVars[i] = intVars[i-1];
  }
  return new Var(this, intVars[vars.length-1]);
}
```

Inizialmente il metodo controlla le lunghezze degli array (parte omessa) e se non compatibili o se nulli lancia una `Runtime Exception`.

Quindi viene creato un array ausiliario di `IntLVar` di lunghezza pari a quella dei due vettori `vars` e `values`. Questo vettore viene utilizzato per calcolare i risultati parziali, ovvero nell'i-esimo elemento è contenuto il prodotto scalare dei primi $i+1$ elementi.

$$\begin{array}{rcl} \texttt{vars} & = & [v_0, v_1, \ldots, v_i, \ldots, v_{n-1}] \\ \texttt{values} & = & [a_0, a_1, \ldots, a_i, \ldots, a_{n-1}] \\ \hline \texttt{intVars[i]} & = & a_0 \cdot v_0 + a_1 \cdot v_1 + \cdots + a_i \cdot v_i \end{array}$$

Alla fine del processo si ottiene un array il cui ultimo elemento rappresenta proprio il prodotto scalare dei due vettori passati come parametro. A questo punto viene creata e restituita una nuova variabile intera JSR-331 mediante un opportuno costruttore.

Metodo `allDifferent`

L'interfaccia `Problem` definisce un modo semplice per creare ed inserire nel problema uno dei più comuni vincoli globali conosciuto come AllDifferent:

public Constraint postAllDifferent(Var[] vars);

È definito un altro sinonimo più compatto:

public Constraint postAllDiff(Var[] vars);

Questi metodi creano, inseriscono nel problema e restituiscono un nuovo vincolo per cui ogni variabile dell'array `vars` debba avere un valore diverso dalle altre. La definizione del metodo è la seguente:

Listato 4.9: `allDifferent`.

```
public Constraint postAllDifferent(Var[] vars) {
        if (vars.length == 0)
                throw new RuntimeException("Variable array
                    must not be empty.");
        Constraint result = new AllDifferent(vars);
        post(result);
        return result;
}
```

Questo metodo crea semplicemente un nuovo vincolo appoggiandosi al costruttore della classe `AllDifferent` che verrà definito nella sezione 4.8.5.

Metodi `postCardinality`

La specifica JSR-331 include tra i metodi pubblici alcune funzioni che creano vincoli inerenti alla cardinalità di array di variabili intere. Questi vincoli detti di cardinalità (`Cardinality Constraints`), che tuttavia non hanno nulla a che fare con le variabili insiemistiche, sono quattro:

```
public Constraint postCardinality(Var[], int, String, int);
public Constraint postCardinality(Var[], int, String, Var);
public Constraint postCardinality(Var[], Var, String, int);
public Constraint postCardinality(Var[], Var, String, Var);
```

Poiché sono vincoli abbastanza particolari, la loro trattazione più dettagliata viene fatta a parte, nella sezione 4.8.3 per quanto riguarda la classe `Cardinality` e nell'appendice A per l'approccio e la trattazione del problema d'implementazione di tale vincolo.

Metodi `postGlobalCardinality`

L'interfaccia `Problem` specifica anche metodi utili per creare vincoli di cardinalità globale noti come *GCC* (*Global Cardinality Constraints*) che rappresentano non uno, ma più vincoli di cardinalità allo stesso tempo. La specifica prevede che il livello d'implementazione di questi possa essere comune o specifico, ovvero è fornita un'implementazione nella classe `AbstractProblem`. È quindi possibile implementare o meno questi metodi.

La lista dei metodi, limitata alle variabili intere è la seguente:

```
public Constraint postGlobalCardinality(Var[], int[], Var[])
    ;
public Constraint postGlobalCardinality(Var[], int[], int[],
    int[]);
```

In entrambi i casi il metodo sfrutta la costruzione di un vincolo specifico implementato mediante la classe `GlobalCardinality` che verrà definito in 4.8.4.

Listato 4.10: `postGlobalcardinality`.

```
public Constraint postGlobalCardinality(
                Var[] vars,
                int[] values,
                Var[] cardinalityVars) {
        Constraint c = new GlobalCardinality(vars, values,
            cardinalityVars);
        c.post();
        return c;
}
```

In questo esempio si può notare che il costruttore utilizza esattamente i parametri d'invocazione della funzione. Il vincolo creato viene quindi inserito nel problema.

4.3 Interfaccia comune: `CommonBase`

Prima di parlare nello specifico delle componenti del problema (inteso come CSP) e quindi di variabili e vincoli, occorre parlare dell'interfaccia comune. Come per l'interfaccia `Problem` anche le interfacce `Var`, `VarBool`, `VarSet` e `Constraint` sono fornite di un'implementazione astratta (non pura) che fornisce attributi, costrutti e metodi di base. Questa è la classe `CommonBase`.

4.3.1 Attributi

Gli attributi della classe `CommonBase` sono quattro:

```
        Problem problem;
        String  name;
        Object  impl;
        Object  businessObject;
```

L'attributo `problem` rappresenta il problema a cui l'oggetto del CSP è associato, la stringa `name` è il nome dato all'oggetto. Si parla di oggetto perché questa classe implementa gli attributi di base di ogni oggetto di un CSP, ed è chiaro che ogni variabile intera, booleana o insiemistica possa

avere un nome, e *debba* sicuramente avere un problema associato e una parte implementativa.

La parte implementativa è rappresentata dall'attributo `impl`, che per l'appunto è un'istanza `Object`, ovvero l'oggetto generico Java. Ogni implementazione specifica (come JSetL) deve, se supportato, appoggiarsi all'attributo `impl` per sfruttare la propria implementazione.

L'ultimo attributo della classe è `businessObject`, che può essere utilizzato come attributo di supporto per le specifiche implementazioni.

4.3.2 Costruttori

Con la classe `CommonBase` vengono forniti due costruttori, uno senza parametro ed uno con parametro stringa che ne rappresenta il nome.

Listato 4.11: costruttori di `CommonBase`.

```
public CommonBase(Problem problem) {
        this(problem,"");
}

public CommonBase(Problem problem, String name) {
        this.problem = problem;
        this.name = name;
        impl = null;
        businessObject = null;
}
```

Il primo costruttore definito nel listato 4.11 si limita a chiamare il secondo passando come parametro la stringa vuota. Quest'ultimo imposta il problema ed il nome passati come parametro ai relativi attributi `problem` e `name`, infine `impl` e `businessObject` vengono impostati a `null`.

L'importanza di questi costruttori è dovuta al fatto che ogni classe derivata dalla suddetta dovrà utilizzarli. Questo implica che ogni classe (`Var`, `VarBool`, `VarSet` e `Constraint`) dovrà fornire tali costruttori di base ed utilizzare quindi il metodo `setImpl` per definire la propria implementazione dell'oggetto costruito.

4.3.3 Metodi

La classe `CommonBase` fornisce alcuni metodi comuni per qualsiasi tipo di variabile o vincolo. Se ne descrivono brevemente alcuni.

- **public** Problem getProblem()

  Restituisce il problema a cui l'oggetto d'invocazione è legato.

- **public void** setName(String name)

  Imposta il nome dell'oggetto.

- **public** String getName()

  Restituisce il nome dell'oggetto d'invocazione.

- **public void** setImpl(Object impl)

  Imposta l'implementazione concreta di uno specifico solver per l'oggetto d'invocazione.

- **public** Object getImpl()

  Restituisce l'implementazione concreta (dello specifico solver) per l'oggetto d'invocazione.

- **public void** setObject(Object obj)

  Aggiunge un oggetto ausiliari.

- **public** Object getObject()

  Restituisce l'oggetto ausiliario.

4.4 Classe `Var`

Questa classe implementa le variabili intere vincolate `Var` per la specifica JSR-331 estendendo la classe `AbstractVar` (e quindi `CommonBase`).

Come visto nella descrizione dell'interfaccia comune vengono forniti tutti gli attributi e metodi utili ed è quindi sufficiente implementare i metodi dell'interfaccia `AbstractVar` ed i costruttori di `CommonBase`, mappando quindi le funzionalità richieste per la classe `Var` con quelle fornite da JSetL (tramite `IntLVar`).

Listato 4.12: metodi di Var

```
public int getMin();
public int getMax();
public boolean isBound();
public boolean contains(int value);
public Var plus(int value);
public Var minus(int value);
public Var plus(Var x);
public Var minus(Var x);
public Var multiply(int value);
public Var multiply(Var x);
public Var divide(int value);
public Var divide(Var x);
```

4.4.1 Costruttori

Sono stati implementati vari costruttori per la classe `Var` atti a soddisfare diverse esigenze, in primis quelle richieste dalla specifica e quindi altre suggerite dallo sviluppo del progetto.

Il meccanismo è simile per tutti i costruttori: si richiama il costruttore della classe base (mediante il costrutto `super`) con il parametro `Problem` (e a volte il nome), quindi si imposta l'oggetto `impl` con l'implementazione concreta.

Listato 4.13: un costruttore di `Var`.

```
public Var(Problem problem) {
        super(problem);
        String name = problem.getFreshName();
        setImpl(new IntLVar(name));
        setName(name);
}
```

In questo esempio si vede il costruttore con il solo parametro `problem`. Viene chiamato il costruttore `AbstractVar(problem)` mediante il costrutto `super(problem)`, quindi viene generato un nuovo nome mediante la funzione di supporto `getFreshName`* della classe `Problem`.

A questo punto viene impostato l'attributo `impl` con una nuova istanza della classe `IntLVar` creata con il costruttore con parametro stringa, che ne rappresenta il nome, infine questo viene settato anche per la variabile `Var`.

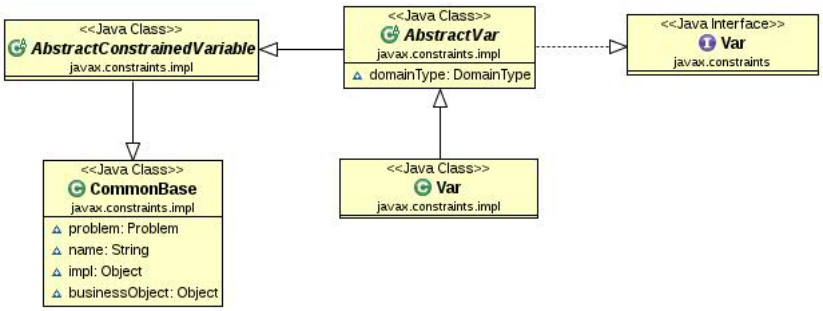

Figura 4.2: Class Diagram di `Var`.

4.4.2 Metodi di uso generale

I primi quattro metodi dell'elenco 4.12 sono metodi di utilità generica e se ne dà una descrizione dettagliata.

*La funzione ausiliaria `getFreshName` si limita a creare un nome per le variabili interne che sia unico, al fine di rendere più chiara la fase di debugging, non è trattata nel dettaglio.

Metodo getMin

Il metodo `getMin` restituisce il più piccolo intero del dominio della variabile d'invocazione. È definito nel seguente modo:

Listato 4.14: `getMin`.

```
public int getMin() {
        MultiInterval domain = ((IntLVar) getImpl()).
            getDomain();
        return domain.getGlb();
}
```

Dapprima viene presa l'implementazione concreta con il metodo `getImpl`, che in questo caso restituisce l'istanza di `IntLVar` relativa alla variabile d'invocazione (`this`). A questo punto, mediante il metodo `getDomain`, dalla variabile si ottiene il dominio sotto forma di `MultiInterval`. Viene quindi restituito il minimo valore di `domain` con il metodo `MultiInterval.getGlb`.

Metodo getMax

Il metodo `getMax` restituisce il più grande intero del dominio della variabile d'invocazione. È definito nel seguente modo:

Listato 4.15: `getMax`.

```
public int getMin() {
        MultiInterval domain = ((IntLVar) getImpl()).
            getDomain();
        return domain.getGlb();
}
```

Analogamente a `getMin` questo metodo sfutta le funzionalità di `IntLVar` e `MultiInterval`.

Metodo isBound

Il metodo `isBound`, come suggerisce il nome, è di tipo booleano, ovvero ha due possibili valori di ritorno: `true` se il dominio della variabile vincolata si riduce ad un singoletto, `false` altrimenti.

Listato 4.16: `isBound`.

```
public boolean isBound() {
        return ((LVar) getImpl()).isBound();
}
```

Questo metodo si appoggia semplicemente al relativo metodo di JSetL per le variabili intere.

Metodo contains

Anche questo è un metodo booleano, restituisce `true` se e solo se un dato intero appartiene al dominio della variabile.

Listato 4.17: `contains`.

```
public boolean contains(int value) {
        MultiInterval domain = ((IntLVar) getImpl()).
            getDomain();
        return domain.contains(value);
}
```

Come nei metodi precedenti viene prima richiamato il dominio della variabile JSetL e in seguito viene ritornato il medesimo metodo sui multi-intervalli `MultiInterval.contains(int)`.

4.4.3 Operazioni aritmetiche

La classe `AbstractVar` prevede l'implementazione dei comuni operatori aritmetici per le variabili intere: "+", "·", "−", "÷".

Per ogni operatore è definita una funzione con parametro intero ed una che abbia come parametro un'altra variabile intera, questo permette di ottenere delle espressioni del tipo:

$$X + k, \quad X + Y,$$

dove X e Y rappresentano delle variabili intere vincolate e k un intero.

La lista dei metodi aritmetici è definita dalle ultime otto dichiarazioni del listato 4.12, poiché si basano tutte sullo stesso principio ne verranno definite nel dettaglio solo due, una con parametro intero ed una con parametro variabile.

Listato 4.18: `plus`, con parametro intero.

```
public Var plus(int value) {
        Problem p = (Problem) getProblem();
        Var x = new Var(p, p.getFreshName());
        x.setImpl(((IntLVar) getImpl()).sum(value));
        ((IntLVar)x.getImpl()).setName(x.getName());
        // To constraint the new variable.
        Constraint c = new Constraint(p,
                ((IntLVar) x.getImpl()).eq(((IntLVar)
                    getImpl()).sum(value)));
        p.post(c);
        return x;
}
```

Il metodo crea una nuova variabile `x` di tipo `Var` che poi verrà restituita come risultato, a questa viene assegnato un nuovo nome e una nuova

implementazione, vincolata ad essere il risultato della somma della variabile d'invocazione e dell'intero `value`.

A questo punto si potrebbe dire che il lavoro sia finito e si possa restituire il risultato, tuttavia si commetterebbe un errore. La nuova variabile deve infatti essere vincolata ed è quindi necessario aggiungere al solver JSetL il vincolo $X = T + k$ dove X è la variabile creata, T è quella di invocazione e k l'intero `value`.

Questa aggiunta (che, nel codice, è definita subito dopo il commento) è stata dovuta non tanto per la correttezza del metodo per quanto riguarda le variabili del problema, ma per le variabili di supporto i cui vincoli non verrebbero aggiornati all'interno del solver JSetL.

Listato 4.19: `multiply`, con parametro variabile.

```
public Var multiply(javax.constraints.Var x) {
        Problem p = (Problem) getProblem();
        Var a = new Var(p, p.getFreshName());
        a.setImpl(((IntLVar) getImpl()).mul(((Var) x).
            getIntLVar()));
        ((IntLVar)a.getImpl()).setName(a.getName());
        // To constraint the new variable.
        Constraint c = new Constraint(p,
            ((IntLVar) a.getImpl()).eq(((IntLVar) getImpl()
                ).mul((IntLVar) x.getImpl())));
        p.post(c);
        return a;
}
```

Come si può notare il metodo con parametro variabile è sostanzialmente identico, l'unica differenza è data dalla chiamata della funzione `getImpl` per il parametro `x` all'interno del costruttore del nuovo vincolo inserito.

4.5 Classe `VarBool`

La classe `VarBool` rappresenta le variabili booleane. Queste si possono considerare un caso particolare delle variabili intere, con il dominio ristretto ai valori $[0, 1]$. In cui 0 rappresenta il valore `false` e 1 il valore `true`.

JSR-331 fornisce un'implementazione di base per questa classe che estende `Var`, tuttavia si è scelto di ridefinirla (anche se in modo del tutto analogo) per avere una gestione più diretta e per facilitare eventuali modifiche future.

```
public class VarBool extends Var implements VarBool {
```

4.5.1 Costruttori

I costruttori della classe chiamano quelli della classe base `Var` e quindi impostano l'implementazione con una variabile dal dominio $[0, 1]$

Listato 4.20: un costruttore di `VarBool`

```
public VarBool(Problem problem, String name) {
        super(problem, name);
        setImpl(new IntLVar(name, 0, 1));
}
```

4.6 Classe `VarSet`

La specifica JSR-331 introduce un'interfaccia basilare per le variabili insiemistiche vincolate. Al contrario delle variabili intere, quando un'istanza di questa classe è considerata bound significa che coincide ad un singolo insieme di valori interi.

Il pacchetto `javax.constraints.impl` fornisce la classe `BasicVarSet` che di fatto implementa l'interfaccia `VarSet`. Tuttavia poiché JSetL ha le proprie classi che implementano insiemi, e soprattutto insiemi di interi mediante la classe `SetLVar`, si è deciso di non considerare l'implementazione di base fornita e proseguire con l'approccio utilizzato per la classe `Var`.

```
public class VarSet extends AbstractConstrainedVariable
    implements javax.constraints.VarSet {
```

La classe estende `AbstractConstrainedVariable` e quindi `CommonBase` che come si è già più volte sottolineato, fornisce tutti gli strumenti necessari. Inoltre `VarSet` implementa direttamente l'interfaccia `VarSet`, al contrario di quanto visto per la classe `Var`.

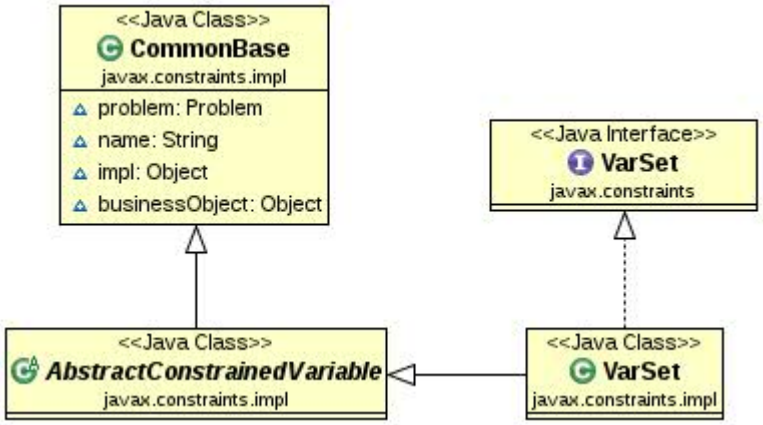

Figura 4.3: Class Diagram di `VarSet`.

Il dominio di una variabile insiemistica consiste in due insiemi:

- Required Set: un insieme di interi i cui valori appartengono tutti alla variabile (lower bound);
- Possibile Set: un insieme di interi in cui almeno un valore appartiene alla variabile (upper bound).

Il Required Set è sempre un sottoinsieme del Possibile Set. Ad esempio, se una variabile insiemistica rappresenta i giorni lavorativi della settimana, i valori possibili sono: $\{1, 2, 3, 4, 5, 6, 7\}$, mentre quelli richiesti dovranno essere un sottoinsieme di cardinalità 5, ad esempio $\{1, 2, 3, 4, 5\}$. È permesso rimuovere elementi solo dai valori possibili ed aggiungerli solo a quelli richiesti. La cardinalità di una variabile insiemistica vincolata è una variabile intera vincolata. È possibile definire intersezione ed unione di variabili insiemistiche.

Se chiamiamo V una generica variabile insiemistica, L e U due insiemi di interi che rappresentano rispettivamente Required Set e Possibile Set, tale che $L \subseteq U$, allora il dominio di V si può esprimere nel seguente modo:

$$\mathrm{dom}(V) = \{L, U\}.$$

Tutte queste caratteristiche, specificate nel documento di riferimento di JSR-331 [1], sono già presenti nella classe `SetLVar` di JSetL, ed è stato quindi semplice e naturale sfruttarne le caratteristiche.

4.6.1 Costruttori

I costruttori richiesti dall'interfaccia sono solo quelli definiti in `CommonBase`, infatti `VarSet` non ne prevede dei propri. I costruttori standard sono:

```
public VarSet(Problem);
public VarSet(Problem, String);
```

mentre quelli definiti in un secondo momento, per venire incontro alle necessità implementative sono:

```
public VarSet(Problem, int[], String);
public VarSet(Problem, Set<Integer>, Set<Integer>, String);
public VarSet(Problem, MultiInterval, String);
public VarSet(Problem, SetLVar);
```

Come per la classe `Var` ogni costruttore utilizza il medesimo approccio: viene chiamato il costruttore della classe base mediante il costrutto `super` e quindi viene impostato l'oggetto `impl` con una chiama al relativo costruttore di `SetLVar`.

Si danno alcuni esempi.

Listato 4.21: un costruttore standard di VarSet

```
public VarSet(Problem problem, String name) {
        super(problem);
        setImpl(new SetLVar(name));
        setName(name);
}
```

Il costruttore in questione si comporta esattamente come quello definito per la classe Var.

Listato 4.22: costruttore con lista di interi.

```
public VarSet(Problem problem, int[] values, String name) {
        super(problem, name);
        MultiInterval s = new MultiInterval();
        for (int i = 0; i < values.length; i++)
                s.add(values[i]);
        setImpl(new SetLVar(name,s));
}
```

In questo costruttore, che ha come parametro una lista di interi ed il nome, viene creato un multi-intervallo a cui vengono aggiunti uno ad uno tutti gli elementi dell'array values. Quindi viene chiamato il costruttore che ha come parametro un MultiInterval fornito da JSetL.

Listato 4.23: costruttore con un multi-intervallo.

```
public VarSet(Problem problem, MultiInterval lb, String name
    ) {
        super(problem, name);
        setImpl(new SetLVar(name, lb, MultiInterval.universe
            ()));
}
```

Questo costruttore ausiliario sfrutta la classe JSetL MultiInterval di cui si è già spesso parlato e che è definita in [2]. In questo caso viene chiamato un costruttore di SetLVar che utilizza due multi-intervalli, il primo rappresenta i valori opzionali, mentre il secondo quelli richiesti.

Sia L l'insieme che rappresenta il multi-intervallo lb, $\mathcal{U}$ l'insieme universo (di tutti gli interi rapprensentabili dall'implementazione), allora la variabile insiemistica creata V ha dominio:

$$\mathrm{dom}(V) = \{L,\mathcal{U}\}.$$

4.6.2 Metodi di uso generale

L'interfaccia VarSet prevede alcuni metodi di utilità generica, tra cui si evidenziano:

```
public boolean isBound();
public Set<Integer> getValue() throws Exception;
public void setValue(Set<Integer> set) throws Exception;
public Set<Integer> getRequiredSet();
public Set<Integer> getPossibleSet();
public boolean isPossible(int value);
public boolean isRequired(int value);
public void remove(int value) throws Exception;
public void require(int val) throws Exception;
public boolean contains(Set<Integer> setOfValues);
public void setEmpty(boolean flag);
public Var getCardinality();
```

Alcuni di questi metodi si avvalgono della controparte nell'implementazione concreta JSetL e quindi non se ne darà la definizione. Tra questi si ha `isBound`, `getValue` e `setValue`.

I metodi che hanno a che fare con il dominio invece seguono un altro approccio, dovendosi basare sulla funzione `getDomain` che restituisce il dominio della variabile JSetL `SetLVar`. Questo dominio è un'istanza della classe `SetInterval` e quindi i metodi sono basati su questa classe oltre che sui multi-intervalli già citati.

Listato 4.24: metodi getter per il dominio.

```
public Set<Integer> getRequiredSet() {
        return ((SetLVar) getImpl()).getDomain().getGlb();
}

public Set<Integer> getPossibleSet() {
        return ((SetLVar) getImpl()).getDomain().getLub();
}
```

Come anticipato la chiave dei metodi `getRequiredSet` e `getPossibleSet` è una funzione della classe `SetInterval` che rappresenta il dominio di una variabile insiemistica. Con `getGlb` viene restituito un multi-intervallo (che implementa un `Set<Integer>`) rappresentante il lower-bound della variabile. Invece `getLub` restituisce l'upper-bound. I due oggetti mappano perfettamente l'insieme required e possible definiti in `VarSet`.

Listato 4.25: un intero è possibile o richiesto?.

```
public boolean isPossible(int value) {
        return ((SetLVar) getImpl()).getDomain().getLub().
            contains(value);
}

public boolean isRequired(int value) {
        return ((SetLVar) getImpl()).getDomain().getGlb().
            contains(value);
```

```
}
```

Nelle funzioni `isPossible` e `isRequired`, dopo aver ottenuto l'insieme required o possible viene utilizzato il metodo `contains` che restituisce un booleano a seconda che l'intero passato come parametro sia contenuto nel multi-intervallo di invocazione o meno.

Listato 4.26: inserire e rimuovere elementi.

```
public void remove(int value) throws Exception {
        ((SetLVar) getImpl()).getDomain().getLub().remove(
            value);
}

public void require(int value) throws Exception {
        ((SetLVar) getImpl()).getDomain().getGlb().add(value
            );
}
```

Nella descrizione inziale della classe si è detto che è permesso rimuovere elementi solo dai valori possibili ed aggiungerli solo a quelli richiesti. I metodi `remove` e `require` implementano quanto detto, non sono state quindi fornite funzioni che aggiungano elementi all'upper-bound o ne rimuovano dal lower-bound.

Listato 4.27: inclusione insiemistica.

```
public boolean contains(Set<Integer> setOfValues) {
        return ((SetLVar) getImpl()).getDomain().getLub().
            containsAll(setOfValues);
}
```

Il metodo booleano `contains` controlla se ogni elemento dell'insieme dato sia contenuto nel dominio della variabile d'invocazione. Utilizza il metodo `containsAll` sui multi-intervalli.

Listato 4.28: getter per la cardinalità.

```
public Var getCardinality() {
        Var result = new Var((Problem) getProblem(), ((
            SetLVar) getImpl()).card());
        return result;
}
```

Il metodo `getCardinality` si avvale del metodo `card` presente nella classe `SetLVar`, che restituisce una variabile vincolata intera rappresentante la cardinalità dell'insieme. Questa variabile viene poi utilizzata per costruire una nuova variabile intera `Var` che viene quindi restituita dal metodo.

4.6.3 Operazioni insiemistiche

`VarSet` prevede le operazioni insiemistiche di unione ed intersezione. Queste operazioni sono già presenti in `SetLVar` ed è stato quindi semplice implementarle. JSetL prevede inoltre le operazioni di complementazione e differenza insiemistica al contrario dello standard, tuttavia non è stata fornita una implementazione per queste.

Listato 4.29: unione insiemistica.

```
public VarSet union(javax.constraints.VarSet varSet) {
        Problem p = (Problem) getProblem();
        SetLVar tmp = ((SetLVar) varSet.getImpl());
        VarSet result = new VarSet(p);
        result.setImpl(((SetLVar)getImpl()).union(tmp));
        // To constraint the new variable.
        Constraint c = new Constraint(p,
                ((SetLVar) result.getImpl()).eq(((SetLVar)
                    getImpl()).union(tmp)));
        p.post(c);
        return result;
}
```

Listato 4.30: intersezione insiemistica.

```
public VarSet intersection(javax.constraints.VarSet varSet)
    {
        Problem p = (Problem) getProblem();
        SetLVar tmp = ((SetLVar) varSet.getImpl());
        VarSet result = new VarSet(p);
        result.setImpl(((SetLVar)this.getImpl()).intersect(
            tmp));
        // To constraint the new variable.
        Constraint c = new Constraint(p, ((SetLVar) result.
            getImpl()).eq(
                        ((SetLVar)getImpl()).intersect(tmp))
                            );
        p.post(c);
        return result;
}
```

In entrambi i metodi si dichiara una variabile temporanea di tipo `SetLVar` che rappresenti la variabile passata come parametro e quindi si crea la variabile di tipo `VarSet` che diventerà il risultato dell'operazione. A `result` viene assegnato il risultato dell'operazione tra la variabile temporanea e la variabile d'invocazione concreta. Anche in questo caso, come per le operazioni aritmetiche sulle variabili intere, occorre aggiornare i vincoli per la nuova variabile creata. Il sistema utilizzato è quindi il medesimo.

4.7 Classe Constraint

L'ultima classe che si tratta a riguardo della definizione del problema è quella che rappresenta il vincolo generico. JSR-331 specifica i vincoli più comuni che definiscono relazioni tra variabili. Questi vincoli possono essere ottenuti tramite l'interfaccia `Problem` ed ognuno di essi è un' istanza di questa classe.

4.7.1 Implementazione

`Constraint` implementa i vincoli JSR-331 estendendo la classe astratta dell'implementazione.

```
abstract public class AbstractConstraint extends CommonBase
    implements javax.constraints.Constraint {
```

Come si può vedere `AbstractConstraint` a sua volta estende `CommonBase`, già descritta nella sezione 4.3. Questo vuol dire che ogni attributo e metodo utile è ereditato dalla classe e verrà quindi sfruttato nell'implementazione. Si hanno poi i metodi definiti nella classe astratta: `post`, `and`, `or`, `negation` e `implies`; a parte il primo sono tutti stati ridefiniti poiché JSetL fornisce la propria implementazione.

4.7.2 Costruttori

Analogamente a quanto visto per gli altri elementi del problema (le variabili) i costruttori di questa classe richiedono come parametro il problema. Viene poi fornito un costruttore che aggiunge il nome ed infine uno ausiliario per l'implementazione JSetL.

Listato 4.31: i costruttori.

```
public Constraint(Problem problem) {
        super(problem);
        setImpl(new JSetL.Constraint());
}

public Constraint(Problem problem, String name) {
        super(problem, name);
        setImpl(new JSetL.Constraint());
}

public Constraint(Problem problem, JSetL.Constraint
    constraint) {
        super(problem);
        setImpl(constraint);
}
```

I tre costruttori sopra definiti utilizzano il medesimo approccio di ogni classe che si basa su `CommonBase`: chiamano il costruttore base e quindi impostano l'implementazione concreta di JSetL.

4.7.3 Vincoli logici

La specifica JSR-331 prevede quattro metodi che rappresentano i comuni operatori logici per creare vincoli: $\wedge$, $\vee$, $\neg$ e $\Rightarrow$.

Siano $\mathcal{C}_1$ e $\mathcal{C}_2$ generici vincoli, allora le seguenti espressioni sono a loro volta vincoli:

$$\mathcal{C}_1 \wedge \mathcal{C}_2, \quad \mathcal{C}_1 \vee \mathcal{C}_2, \quad \neg\mathcal{C}_1, \quad \mathcal{C}_1 \Rightarrow \mathcal{C}_2.$$

Congiunzione logica

La congiunzione logica è implementata tramite il metodo `and` ed ha un parametro di tipo `Constraint`:

Listato 4.32: `and`.

```
public Constraint and(Constraint c) {
        JSetL.Constraint constraint = ((Constraint) c).
            getConstraint();
        Constraint result = new Constraint(getProblem(),
            this.getConstraint().and(constraint));
        return result;
}
```

Il metodo semplicemente crea un nuovo vincolo mediante il costruttore ausiliario creato per JSetL. Utilizza il metodo `and` della classe `Constraint` di JSetL.

Disgiunzione logica

Anche la disgiunzione logica ha un vincolo come parametro ed è implementata dal seguente codice:

Listato 4.33: `or`.

```
public Constraint or(Constraint c) {
        JSetL.Constraint constraint = ((Constraint) c).
            getConstraint();
        Constraint result = new Constraint(getProblem(),
            this.getConstraint().or(constraint));
        return result;
}
```

Il metodo è praticamente identico al primo, con l'unica differenza che nel costruttore è utilizzato il metodo `or`.

Negazione

La Negazione, contrariamente agli altri metodi, non ha argomenti, per il resto è identico come approccio: viene creato un nuovo vincolo mediante il metodo `notTest`:

Listato 4.34: `negation`.

```
public Constraint negation() {
        Constraint result = new Constraint(getProblem(),
            this.getConstraint().notTest());
        return result;
}
```

Implicazione

L'implicazione segue di pari passo i primi due metodi descritti: ha un parametro di tipo `Constraint` e crea un nuovo vincolo mediante il metodo `impliesTest` di JSetL.

Listato 4.35: `implies`.

```
public Constraint implies(Constraint c) {
        JSetL.Constraint constraint = ((Constraint) c).
            getConstraint();
        Constraint result = new Constraint(getProblem(),
            this.getConstraint().impliesTest(constraint));
        return result;
}
```

Nota. Nella classe `JSetL.Constraint` erano presenti i soli metodi `or` e `and` che implementavano la disgiunzione e congiunzione logica (la disgiunzione viene propagata in modo non deterministico). Durante lo sviluppo dell'interfaccia per la specifica si è reso necessaria l'introduzione di metodi per la negazione e l'implicazione.

Sono stati quindi forniti i metodi `impliesTest`, e `notTest` che realizzano un'implementazione deterministica delle relative operazioni logiche.

Nell'approccio di JSetL per la soluzione dei vincoli il non determinismo garantisce il risultato a scapito dell'efficienza; tuttavia sono stati introdotti altri sistemi per raggiungere la correttezza e la completezza, come il labeling visto nel capitolo 3.

L'approccio nello sviluppo dell'implementazione JSR-331 basata su JSetL verte sul labeling di ogni variabile inserita nel problema e questo garantisce un risultato quando si cerca una soluzione. Per questo motivo l'utilizzo di metodi deterministici non è un problema, anzi in alcuni test si è rivelato un approccio molto più efficiente.

4.8 Vincoli specifici

Durante il testing mediante TCK e con l'esecuzione di alcuni esempi inclusi nella cartella `sample` dello stesso si è resa necessaria l'implementazione di alcune classi non incluse nel documento di specifica:

- `AllDifferent.java`
- `And.java`
- `Cardinality.java`
- `Element.java`
- `GlobalCardinality.java`
- `IfThen.java`
- `Linear.java`
- `Neg.java`
- `Or.java`

Ogniuna di queste estende la classe `Constraint` e ne ridefinisce i costruttori in modo tale che rappresenti un ben definito vincolo. Alcune di queste classi sono utilizzate nelle definizioni dei vari metodi della classe `Problem` per la creazione e l'inserimento di vincoli relativi al problema, altre invece ne sfruttano le peculiarità.

Poiché ogni classe che implementa vincoli specifici estende `Constraints` non saranno date tutte le definizioni di classe, se ne fornisce una a fine esemplificativo:

```
public class Linear extends Constraint {
```

Verranno quindi di seguito, per ogni vincolo, descritti i costruttori più significativi.

4.8.1 Classe `Linear`

Questa classe rappresenta il più classico vincolo sulle variabili logiche intere, ovvero il vincolo lineare. JSR-331 prevede quattro possibili vincoli come descritto nella sezione 4.2.2.

I costruttori che hanno come parametri array di interi e variabili funzionano esattamente come il metodo `scalProd` definito all'interno della classe `Problem`. L'unica differenza è che una volta computata la variabile rappresentante il prodotto scalare questa viene vincolata al valore o alla variabile passata come parametro mediante l'operatore dato, esattamente

come verrà descritto di seguito. Non se ne darà quindi una descrizione dettagliata.

Per quanto riguarda i costruttori con variabili singole si dà la definizione di uno solo dei metodi poiché pressoché identici. Entrambi hanno come primo parametro una variabile `var` o `var1`, come secondo una stringa `oper` e come terzo ed ultimo una variabile `var2` o un intero `value`. La stringa `oper` rappresenta l'operatore di confronto che viene gestito dalla funzione `getOperator`, su cui non ci si soffermerà.

Listato 4.36: costruttore `Linear` con variabile e intero.

```
public Linear(javax.constraints.Var var, String oper, int
    value) {
        super(var.getProblem());
        IntLVar v = ((Var) var).getIntLVar();
        switch(getOperator(oper)) {
        case 1: {
                // Case = "equals".
                JSetL.Constraint constraint = v.eq(value);
                setImpl(constraint);
        } break;
        case 2: {
                // Case != "not equals".
                JSetL.Constraint constraint = v.neq(value);
                setImpl(constraint);
        } break;
        .
        .
        .
        case 6: {
                // Case >= "greater equals".
                JSetL.Constraint constraint = v.ge(value);
                setImpl(constraint);
        } break;
        default: throw new UnsupportedOperationException();
        }
}
```

Inizialmente viene chiamato il costruttore della classe base con l'istanza della classe `Problem` della variabile passata come parametro. Quindi viene estratta la variabile intera JSetL di tipo `IntLVar` mediante la funzione ausiliaria `getIntLVar` (che non fa altro che chiamare il metodo `getImpl` dell'implementazione comune). Quindi, a seconda del caso, viene creato un vincolo JSetL opportuno (nel listato, per leggibilità, sono stati omessi alcuni casi) utilizzato per essere impostato come implementazione del vincolo creato mediante il metodo `setImpl`.

Ecco un'istruzione JSetL per la creazione del vincolo:

```
JSetL.Constraint constraint = v.eq(value);
```

Se nessun caso dovesse essere catturato mediante lo `switch` verrebbe lanciata un'eccezione di operazione non supportata.

Come accennato in precedenza la versione del metodo con due variabili (`var1` e `var2`) è di poco differente, infatti inizialmente viene estratta la seconda variabile JSetL e viene quindi trattata come quella intera, poiché JSetL fornisce i medesimi metodi per interi e variabili. Ecco la riga aggiunta prima dello statement `switch`:

```
IntLVar value = ((Var) var2).getIntLVar();
```

4.8.2 Classe `Element`

`Element` definisce vincoli per la trattazione di array di variabili intere o array di variabili logiche intere. JSR-331 richiede quattro possibili vincoli, poiché simili se ne descriveranno solo due: uno per l'array di variabili ed uno per quello di interi.

Di fatto questo è un vincolo lineare che coinvolge gli elementi dell'array dato, in termini degli indici individuati dal dominio della variabile logica indice. Ad esempio nel caso:

Element(**int**[] array, javax.constraints.Var indexVar, ''='', **int** value)

se `array[indexVar]` è una variabile vincolata che ha come dominio gli interi del vettore `array` indicizzati da `indexVar`, il vincolo risultante è del tipo:

array[indexVar] = value.

I quattro costruttori definiti in questa classe sfruttano tutti la medesima meccanica per costruire il vincolo, se ne descriverà pertanto solo uno.

Listato 4.37: `Element` per array di interi.

```
public Element(
                int[] array,
                javax.constraints.Var indexVar,
                String oper,
                javax.constraints.Var var) {
        super(indexVar.getProblem());
        if (indexVar.getMin() > array.length - 1 || indexVar
            .getMax() < 0)
                throw new RuntimeException("elementAt: 
                    invalid index variable");
        Problem p = (Problem) var.getProblem();
        JSetL.Constraint element = new JSetL.Constraint();
        IntLVar z = new IntLVar(p.getFreshName());
```

```
        IntLVar index = (IntLVar) indexVar.getImpl();
        SolverClass solver = ((Solver) p.getSolver()).
            getSolverClass();
        ListOps listOps = new ListOps(solver);
        List<Integer> list = new ArrayList<Integer>();
        for (int i = 0; i < array.length; i++)
                list.add(i,array[i]);
        LList l = new LList("array", list);
        element.and(listOps.ithElem(l, index, z));
        if (oper.equals("=")) {
          element.and(z.eq((IntLVar)var.getImpl()));
        .
        .
```

Questo costruttore ha come parametro un array di interi, una variabile logica (`indexVar`) utilizzata come indice, una stringa rappresentante la relazione con cui si vuole vincolare la nuova variabile risultante `array[indexVar]` ed infine la variabile vincolante. L'algoritmo è semplice, viene costruito un nuovo vincolo risultato `element` e una variabile ausiliaria `z` che rappresenta la sopracitata variabile `array[indexVar]`. Quindi si sfrutta un vincolo utente JSetL chiamato `ListOps` che ha come metodo `ithElem`, questo implementa esattamente il concetto richiesto, ovvero data una lista di interi (o variabili) ed un indice i, vincola una data variabile ad essere equivalente all'i-esimo elemento di tale lista. La variabile in questione è la nuova variabile ausiliaria. Viene quindi aggiornato il vincolo `element` con il vincolo lineare rappresentato dalla stringa `oper` e dalla variabile `var`. Infine il vincolo creato viene impostato nell'implementazione mediante `setImpl`.

4.8.3 Classe `Cardinality`

La classe `Cardinality` modella il vincolo di cardinalità (vedi appendice A per una trattazione completa), in sintesi tratta un array di variabili logiche imponendo ad un dato numero di esse il soddisfacimento di un vincolo lineare.

La specifica richiede quattro versioni per la classe come visto nella sezione relativa alle funzioni `postCardinality` della classe `Problem`. Si darà la descrizione di un solo costruttore, poiché utilizzano tutti il medesimo approccio, cambiando solo le variabili coinvolte.

Listato 4.38: `Cardinality`.

```
public Cardinality (Var[] vars, int cardValue, String oper,
    Var var) {
        super(vars[0].getProblem());
        Problem p = (Problem) vars[0].getProblem();
        IntLVar value = ((javax.constraints.impl.Var) var).
            getIntLVar();
```

```
        JSetL.Constraint cardinality = new JSetL.Constraint
            ();
        IntLVar v = new IntLVar("c"+cardValue, cardValue);
        IntLVar k = new IntLVar("_card"+cardValue);
        SolverClass solver = ((Solver) p.getSolver()).
            getSolverClass();
        Occurrence listOps = new Occurrence(solver);
        Vector<IntLVar> varsList = new Vector<IntLVar>();
        for (int i = 0; i < vars.length; i++)
                varsList.add((IntLVar) vars[i].getImpl());
        cardinality.and(listOps.occurrence(varsList, v, k));
        switch(p.getOperator(oper)) {
        .
        .
        case 3: {
            // Case < "less".
            cardinality.and(k.lt(value));
        } break;
        .
        .
        Constraint result = new Constraint(p, cardinality);
        ((Solver) p.getSolver()).add(result);
        setImpl(result.getImpl());
}
```

La costruzione del vincolo di cardinalità si basa su un vincolo JSetL definito dall'utente[†] chiamato `Occurrence` (vedi A.3.1 per una definizione completa) che di fatto, data una lista di variabili logiche intere, lega un'altra variabile intera ad essere il numero di occorrenze degli elementi di tale lista che assumono un dato valore.

Ogni costruttore di `Cardinality` utilizza un'istanza di `Solver` poiché i vincoli definiti da utente in JSetL richiedono di essere associati ad una `SolverClass`, nel caso di cui sopra la variabile `solver`. Altre variabili di supporto sono `v` e `k` che rappresentano rispettivamente il valore a cui le variabili devono equivalere ed il numero di occorrenze delle variabili nell'array che soddisfano il vincolo di equivalenza.

Operativamente nei costruttori si utilizza un vincolo JSetL `cardinality` che rappresenta la congiunzione dei due vincoli su cui si basa l'algoritmo. Il primo è un'istanza di `Occurrence` che vincola `k` come scritto sopra, quindi il secondo è un vincolo lineare che lega la variabile `k` mediante l'operatore `oper` al valore `value`. Il vincolo lineare è determinato dalla stringa `oper` e come per altre funzioni viste in precedenza si utilizza uno `switch` ed il metodo `getOperator` della classe `Problem`.

Il vincolo JSetL così creato viene quindi utilizzato per impostare l'implementazione dell'oggetto `this`.

[†]I vincoli definiti dall'utente in JSetL sono trattati in [7].

4.8.4 Classe `GlobalCardinality`

JSR-331 specifica due vincoli di cardinalità globale, entrambi trattano un array di variabili ed utilizzano più vincoli `Cardinality`. Il primo vincola un numero esatto di variabili nell'array dato, mentre il secondo ne limita il range.

Cardinalità esatta

La prima versione che si presenta è quella con cardinalità esatta, poiché tale vincolo viene utilizzato in quello con range. Formalmente il vincolo:

```
public GlobalCardinality(Var[] vars, int[] values, Var[]
    card)
```

è tale che, per ogni indice `i` il numero delle volte in cui il valore `values[i]` occorre nell'array `vars` è *esattamente* `card[i]`.

Listato 4.39: corpo del costruttore.

```
super(vars[0].getProblem());
if (card.length != values.length) {
        throw new RuntimeException("
            GlobalCardinality error: arrays values
            and cardinalityVars do not have same
            size");
}
Problem p = (Problem) vars[0].getProblem();
Constraint result = new Constraint(p);
for (int i = 0; i < values.length; i++) {
        Cardinality tmp = new Cardinality(vars,
            values[i], "=", card[i]);
        result.and(tmp);
}
setImpl(result.getImpl());
```

Inizialmente il costruttore chiama quello della classe base, quindi verifica che le dimensioni dei vettori siano compatibili, altrimenti lancia un'eccezione. Viene creato un vincolo di supporto `result` che rappresenta la congiunzione di ogni vincolo di cardinalità coinvolto. Nel ciclo `for` viene creata un'istanza di `Cardinality` temporanea con gli indici relativi come sopra descritto, viene quindi aggiornato `result`. Infine viene impostata l'implementazione del vincolo creato mediante `setImpl`.

Cardinalità limitata

In questa versione il vincolo:

```
public GlobalCardinality(Var[] vars, int[] values, int[]
    cardMin, int[] cardMax)
```

è tale che, per ogni indice `i` il numero delle volte in cui il valore `values[i]` occorre nell'array `vars` è compreso tra `cardMin[i]` e `cardMax[i]` inclusi.

Listato 4.40: inizializzazione delle variabili.

```
        super(vars[0].getProblem());
        Problem p = (Problem) vars[0].getProblem();
        int min = cardMin[0];
        int max = cardMax[0];
        for(int i = 0; i < cardMin.length; i++){
                if(cardMin[i] > cardMax[i]) {
                        throw new RuntimeException("
                            GlobalCardinality error:
                            cardMin["+i+"] <= cardMax["+i+"
                            ]");
                }
                if (cardMin[i] < min)
                        min = cardMin[i];
                if (cardMax[i] > max)
                        max = cardMax[i];
        }
        Var[] cardinalityVars = new Var[values.length];
        for (int i = 0; i < cardinalityVars.length; i++) {
                cardinalityVars[i] = new javax.constraints.
                    impl.Var(p, min, max);
                cardinalityVars[i].setName("card"+i);
        }
```

Il costruttore chiama quello della classe base, quindi inizializza le variabili di supporto chiamate `cardinalityVars` di tipo intero. Queste hanno tutte un dominio compreso tra il più piccolo degli interi appartenenti all'array `cardMin` ed il più grande di `cardMax`.

Listato 4.41: costruzione del vincolo.

```
        GlobalCardinality result = new GlobalCardinality(
            vars,values,cardinalityVars);
        for (int i = 0; i < cardinalityVars.length; i++) {
                p.post(cardinalityVars[i],">=",cardMin[i]);
                p.post(cardinalityVars[i],"<=",cardMax[i]);
        }
        setImpl(result.getImpl());
}
```

A questo punto viene creato un nuovo vincolo di tipo `GlobalCardinality` in cui il vettore di variabili vincolanti (`cardinalityVars`) è costituito da variabili unbound (con dominio ristretto per ridurre l'albero delle soluzioni). Ogni elemento di indice `i` viene quindi vincolato rispetto ai corrispondenti valori degli array `cardMin` e `cardMax`.

4.8.5 Classe `AllDifferent`

Questa classe implementa il famoso vincolo AllDifferent, ovvero dato un array di variabili logiche, questo impone che ogniuna abbia un valore diverso.

Listato 4.42: `postAllDiff`.

```
public AllDifferent(javax.constraints.Var[] vars) {
        super(vars[0].getProblem());
        IntLVar[] vec = new IntLVar[vars.length];
        for (int i = 0; i < vars.length; i++)
                vec[i] = ((Var) vars[i]).getIntLVar();
        JSetL.Constraint alldiff = IntLVar.allDifferent(vec)
             ;
        setImpl(alldiff);
}
```

Il costruttore crea un array di supporto (`vec`) di `IntLVar` al quale assegna il riferimento alla parte di implementazione concreta dell'array di variabili intere passate come parametro. Quindi crea un nuovo vincolo JSetL mediante il metodo `allDifferent` della classe `IntLVar` che di fatto vincola ogni variabile ad essere differente dalle altre.

A questo punto viene impostata l'implementazione del vincolo mediante il solito metodo `setImpl`.

4.8.6 Classi `And`, `Or`, `Neg`, `IfThen`

Le classi `And`, `Or`, `Neg` e `IfThen` rappresentano i classici vincoli di congiunzione, disgiunzione, negazione ed implicazione. La loro trattazione viene fatta in un unico paragrafo poiché utilizzano lo stesso approccio, ovvero di fatto generano un vincolo che sfrutta la relativa funzione della classe base. Si da solo la definizione della classe `Or` a scopo esemplificativo.

Listato 4.43: Classe `Or`.

```
public class Or extends Constraint {

    public Or(Constraint c1, Constraint c2) {
            super(c1.getProblem());
            Constraint result = (Constraint) c1.or(c2);
            setImpl(result.getImpl());
    }

}
```

Come è evidente dal listato di cui sopra, questo genere di classi è molto conciso, si limita a definire un costruttore con il numero di parametri pari a quello della funzione che utilizza per la costruzione del vincolo.

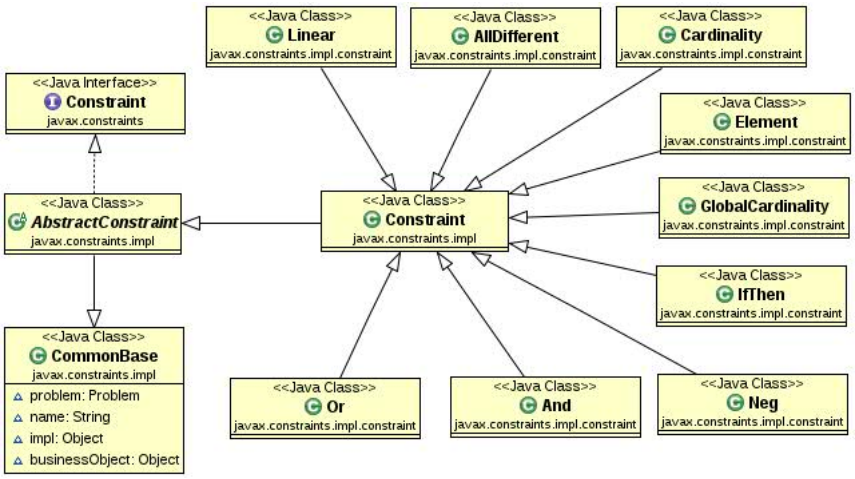

Figura 4.4: Class Diagram delle classi che implementano i vincoli.

Capitolo 5

Implementazione: Rappresentazione della Soluzione

In questo capitolo verranno descritte le classi Java implementate per la specifica JSR-331 nell'ambito della risoluzione di un problema (CSP).

Per ogni classe descritta verrà introdotta l'interfaccia fornita dalla specifica e quindi, con maggior dettaglio, l'implementazione che riguarda il solver JSetL.

Risoluzione del problema

Per rappresentare la risoluzione di un qualsiasi problema, la specifica JSR-331 utilizza le seguenti interfacce:

- `Solver`;
- `SearchStrategy`;
- `Solution`;
- `SolutionIterator`.

Nelle prossime sezioni verranno descritte le suddette classi (implementazioni) con i principali metodi.

5.1 Interfaccia `Solver`

Per rappresentare la parte di risoluzione di un qualsiasi CSP, la specifica JSR-331 utilizza l'interfaccia `Solver`. Il solver permette ad un utente di risolvere un problema cercando una soluzione soddisfacibile od ottimale. Ecco un esempio:

Listato 5.1: una risoluzione di `problem`.

```
problem.log("=== Find One solution:");
Solver solver = problem.getSolver();
Solution solution = solver.findSolution();
if (solution != null)
  solution.log();
else
  problem.log("No Solutions");
```

In questo semplice esempio il solver cerca una soluzione utilizzando una strategia di ricerca di default. La specifica definisce anche un'interfaccia per la strategia di ricerca come si vedrà nella sezione 5.4.

5.1.1 Come ottenere le soluzioni

Lo scopo principale dell'interfaccia `Solver` è quello di dare all'utente gli strumenti, dato un problema, per trovarne le soluzioni. Vengono quindi forniti i seguenti metodi:

```
public Solution findSolution();
public Solution findSolution(ProblemState restoreOrNot);
public Solution findOptimalSolution(Objective objective, Var
    objectiveVar);
public Solution findOptimalSolution(Var objectiveVar);
public Solution[] findAllSolutions();
```

I primi due metodi cercano una soluzione del problema, utilizzano la strategia di default o l'ultima strategia definita. Restituiscono la soluzione trovata (se esiste) o `null`. In entrambi i casi se una soluzione non è stata trovata lo stato del problema viene ristabilito.

Il metodo `findOptimalSolution` cerca una soluzione che massimizzi o minimizzi una variabile obiettivo data. La versione con un solo parametro (definita nell'implementazione comune) si limita a chiamare quella con due parametri in cui l'obiettivo è minimizzare la variabile data. Nel metodo con due parametri il primo può avere due valori: `Objective.MINIMIZE` oppure `Objective.MAXIMIZE`.

L'ultima funzione cerca di trovare tutte le soluzioni del problema. Restituisce un vettore di soluzioni oppure `null` se non ve ne sono. L'utente deve comunque prestare attenzione a non sovraccaricare la memoria poiché il numero di soluzioni di un problema può essere enorme.

5.1.2 Definire una strategia

La seconda funzionalità dell'interfaccia verte sulla definizione di una strategia di ricerca. È possibile definire delle euristiche sulla scelta dei valori e delle variabili a cui assegnare un valore, ad esempio:

```
SearchStrategy strategy = solver.getSearchStrategy();
strategy.setVars(s);
strategy.setVarSelectorType(VarSelectorType.RANDOM);
strategy.setValueSelectorType(ValueSelectorType.RANDOM);
```

In questo esempio si definisce una strategia che coinvolge le variabili di un array `s` in cui la scelta delle variabili e dei valori è casuale.

I metodi forniti dall'interfaccia per quanto concerne la strategia sono:

```
public void setSearchStrategy(SearchStrategy strategy);
public SearchStrategy getSearchStrategy();
public SearchStrategy newSearchStrategy();
public void addSearchStrategy(SearchStrategy strategy);
```

I primi due sono i classici metodi setter e getter che rispettivamente impostano o restituiscono una strategia. `newSearchStrategy` crea una nuova istanza della classe `SearchStrategy`, mentre l'ultima funzione aggiunge una data strategia al solver. Ovviamente un solver può avere più strategie che coinvolgono diverse variabili e l'interfaccia predispone gli strumenti per realizzare questa proprietà.

5.2 Classe `Solver`

La classe `Solver` implementa l'interfaccia `javax.constraints.Solver` estendendo la classe `AbstractSolver` dell'implementazione comune.

```
public class Solver extends AbstractSolver {
```

Questo approccio permette di ereditare tutti i metodi astratti puri da implementare e, dove necessario, ridefinire i metodi non astratti.

5.2.1 Classe `AbstractSolver`

JSR-331 fornisce un'implementazione astratta non pura, ovvero in cui sono implementati concetti e metodi comuni. Innanzitutto si possono notare le seguenti strutture:

```
    public enum ProblemState {
        RESTORE,
        DO_NOT_RESTORE
    }
```

Questa struttura è utilizzata per controllare lo stato del problema dopo l'esecuzione di una risoluzione.

```
    public enum Objective {
        MINIMIZE,
        MAXIMIZE
```

```
    }
```

Questa viene utilizzata per specificare il tipo di ottimizzazione richiesta dal metodo `findOptimalSolution`. Si passa ora ad analizzare la classe astratta dell'implementazione comune.

`AbstractSolver` è una classe astratta fornita dall'implementazione comune (`javax.constraints.impl`) che implementa l'interfaccia `Solver`:

```
abstract public class AbstractSolver implements Solver {
```

definendo ogni metodo ed attributo di utilità generica. Non è una classe astratta pura, poiché fornisce molte implementazioni di base, sia per gli attributi che per i metodi.

Attributi

La classe fornisce un buon numero di attributi per modellare la risoluzione di un problema, si elencano i più utilizzati (durante lo sviluppo dell'interfaccia per JSetL):

```
Problem problem;
protected Vector<SearchStrategy> searchStrategies;

Vector<Solution> solutions;
int      maxNumberOfSolutions;
int      timeLimit;
```

Il primo attributo (`problem`) rappresenta il problema a cui è legato il solver, come si è già detto non vi può essere una soluzione senza un problema, quindi ogni istanza della classe dovrà avere un problema ad essa associato.

Il vettore `searchStrategies` di elementi della classe `SearchStrategy`, come suggerisce il nome, contiene tutte le strategie di ricerca associate al solver.

Un altro vettore utile è rappresentato da `solutions` che memorizza le soluzioni trovate, questo è sfruttato soprattutto dal metodo `findAllSolutions` che ha appunto lo scopo di trovare ogni soluzione valida.

Gli ultimi attributi evidenziati sono due interi e rappresentano il massimo numero di soluzioni (`maxNumberOfSolutions`) ed il limite di tempo per i vari metodi che cercano le soluzioni, espresso in millisecondi (`timeLimit`).

Metodi di uso generale

I metodi di utilità generica o che non hanno bisogno di un'implementazione specifica come quella fornita da JSetL o Choco, vengono implementati direttamente all'interno della classe `AbstractSolver`. Tra quelli più utilizzati si evidenziano:

- **public void** setMaxNumberOfSolutions(**int** number);

  Imposta un limite per il numero di soluzioni che possono essere trovate dal metodo `findAllSolutions` o che possono essere considerate durante l'esecuzione del metodo `findOptimalSolution`. Il valore di default per il numero massimo è -1, che stà a significare nessun limite.

- **public int** getMaxNumberOfSolutions();

  Restituisce il corrente numero massimo di soluzioni.

- **public void** setOptimizationTolerance(**int** tolerance);

  Specifica una tolleranza per il metodo `findOptimalSolution`. Se la differenza tra la nuova soluzione trovata e quella ottimale corrente è minore o uguale alla tolleranza, non viene aggiornata la soluzione ottimale. Di default la tolleranza è 0.

- **public int** getOptimizationTolerance();

  Restituisce la tolleranza corrente.

- **public void** setTimeLimit(**int** milliseconds);

  Imposta un limite di tempo in millisecondi per l'esecuzione totale dei differenti metodi di ricerca delle soluzioni (metodo `find`). Di default non è impostato alcun limite.

- **public int** getTimeLimit();

  Restituisce il tempo limite in millisecondi. Se non è specificato un tempo limite il valore di ritorno è -1.

5.2.2 Implementazione

Una volta definita la classe, come visto all'inizio della sezione 5.1 parlando dello sviluppo dell'interfaccia `Problem`, si sono inserite tutte le definizioni dei metodi astratti dell'interfaccia `Solver` non definite nell'implementazione di base. Si descrivono ora gli attributi ed i metodi definiti, basati sul solver JSetL.

Attributi

La specifica JSR-331, contrariamente a quanto visto per le variabili ed i vincoli, non prevede che uno specifico CP solver implementi una classe apposita per un entità solver. Questo approccio è ragionevole e non è quindi prevista una struttura come `CommonBase` in cui sia possibile impostare una implementazione concreta mediante `setImpl`.

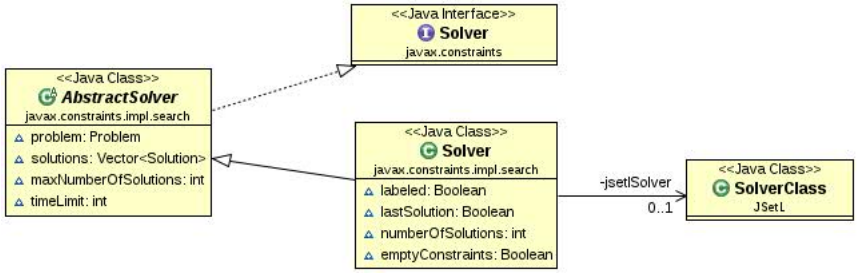

Figura 5.1: Class Diagram di `Solver`.

Tuttavia JSetL fornisce una classe (`SolverClass`) che implementa alcune delle proprietà richieste dall'interfaccia `Solver`, quindi viene utilizzato come attributo `private` un'istanza di `SolverClass`. Si elencano di seguito gli attributi aggiunti rispetto alla classe di base:

Listato 5.2: attributi di Solver.

```
        private SolverClass jsetlSolver;
        Boolean lastSolution = false;
        int numberOfSolutions;

        protected JSetL.Constraint[] jsetlConstraints;
        Boolean emptyConstraints = false;
```

Come già detto, il primo attributo rappresenta il solver concreto di JSetL. Segue un attributo booleano: `lastSolution` che indica se nel processo di ricerca si è arrivati all'ultima soluzione valida per il corrente solver `jsetlSolver`. L'attributo intero rappresentante il corrente numero di soluzioni trovate.

I due attributi separati sono d'ausilio per il passaggio dei vincoli dal problema a cui il solver è legato al solver JSetL.

Costruttori

Vi è un solo costruttore richiesto, che prende come parametro il problema su cui si costruisce il solver.

Listato 5.3: costruttore di base.

```
public Solver(Problem problem) {
        super(problem);
        jsetlSolver = new SolverClass();
        numberOfSolutions = 0;
        getProblemConstraints();
        setProblemConstraints();
}
```

Qui viene chiamato il costruttore della classe astratta di base con il solito costrutto `super(problem)`, quindi viene istanziato l'attributo `jsetlSolver` mediante il suo costruttore. Il numero di soluzioni è impostato a 0. Una volta inizializzati i membri propri della classe, mediante i metodi privati `getProblemConstraints` e `setProblemConstraints` vengono caricati i vincoli presenti nel problema (al momento dell'istanziazione del solver) e quindi salvati nello *store* JSetL.

Listato 5.4: metodi privati per caricare i vincoli.

```
private void setProblemConstraints() {
        if (emptyConstraints)
                return;
        for (JSetL.Constraint c : jsetlConstraints)
                jsetlSolver.add(c);
}

private void getProblemConstraints() {
        jsetlConstraints = ((Problem) getProblem()).
            getJSetLConstraints();
        if (jsetlConstraints == null)
                emptyConstraints  = true;
        else emptyConstraints = false;
}
```

Metodo `newSearchStrategy`

`newSearchStrategy` è un metodo che consente all'utente di creare una nuova strategia di ricerca per il solver specifico, `Solver` in questo caso funziona praticamente come una factory.

Listato 5.5: `newSearchStrategy`.

```
        public SearchStrategy newSearchStrategy() {
                return new SearchStrategy(this);
        }
```

Questo metodo restituisce semplicemente la chiamata ad un costruttore della classe `SearchStrategy`.

Risoluzione del problema

Questa parte sella sezione è dedicata alla ricerca delle soluzioni, verranno qui definiti i metodi `findSolution`, `findOptimalSolution` e `findAllSolutions`.

Metodo findSolution

Il metodo `findSolution` di cui viene richiesta l'implementazione nelle specifiche è quello con parametro (`ProblemState`). Tale parametro definisce se, dopo la ricerca, lo stato del problema vada ristabilito o meno.

Listato 5.6: `findSolution`.

```
public Solution findSolution(ProblemState restoreOrNot) {
  applyHeuristic();
  Solution solution = null;
  if(restoreOrNot == ProblemState.RESTORE) {
      .
      .
      .
  } else if(restoreOrNot == ProblemState.DO_NOT_RESTORE) {
      try {
        jsetlSolver.solve();
        solution = new Solution(this,
                        numberOfSolutions++);
      } catch (Failure e) {
          solution = null;
          e.printStackTrace();
        }
  }
  return solution;
}
```

Il metodo applica le euristiche sulle variabili del problema con `applyHeuristic`, che verrà discusso in seguito, quindi procede con la ricerca della soluzione.

A questo punto la computazione ha un punto di scelta, ovvero se lo stato del problema va ristabilito al termine della ricerca o meno.

Il caso `DO_NOT_RESTORE`, più semplice del primo, applica semplicemente il metodo `solve` al solver JSetL, che se troverà una soluzione verrà salvata nella variabile `solution`, altrimenti sarà lanciata un'eccezione e catturata subito dopo, impostando a `null` la variabile stessa. Il costruttore di `Solution`, come vedremo nella sezione 5.6, salverà ogni variabile del problema internamente.

Listato 5.7: `findSolution` il caso `RESTORE`.

```
    if(restoreOrNot == ProblemState.RESTORE) {
            if(jsetlSolver.check()) {
                    lastSolution = false;
                    try {
                          solution = new Solution(this,
                                          numberOfSolutions++);
                    } catch (Failure e) {
                            e.printStackTrace();
                            return null;
```

```
                }
            }
            else return null;
    }
```

Il parametro `RESTORE` è utilizzato principalmente dal metodo `hasNext` della classe `SolutionIterator`. Concettualmente il suo significato può essere espresso come il ripristino dello stato del problema fino all'ultimo punto di scelta, per poter generare quindi la soluzione successiva. Per implementare questo concetto si è optato per utilizzare la funzione `check` del solver JSetL che consente di generare una soluzione senza svuotare il `constraint store`.

Metodo `findOptimalSolution`

Come suggerisce il nome, questo metodo ricerca una soluzione ottimale per il problema. Per fare ciò l'algoritmo dovrebbe esplorare l'intero spazio delle soluzioni e memorizzare quella più vicina all'obiettivo. In realtà si è adottato un sistema per ridurre tale spazio, propagando vincoli sulla variabile obiettivo, analogamente all'approccio adottato dall'implementazione comune. La descrizione del metodo verrà spezzata in più parti, poiché è piuttosto lungo e composto da più istruzioni correlate.

Listato 5.8: `findOptimalSolution`, inizializzazione.

```
public Solution findOptimalSolution(Objective objective, Var
    objectiveVar) {
        addObjective(objectiveVar);
        long startTime = System.currentTimeMillis();
        if (objectiveVar.getName().isEmpty())
                objectiveVar.setName("Objective");
        if (getProblem().getVar(objectiveVar.getName()) ==
            null) {
                getProblem().add(objectiveVar);
        }
        javax.constraints.Var obj = objectiveVar;
        JSetL.Constraint c = new JSetL.Constraint();
        IntLVar target = (IntLVar) obj.getImpl();
        Boolean maximize = false;
        int bestValue = Integer.MAX_VALUE /2;
        if(objective.equals(Objective.MAXIMIZE)) {
                maximize = true;
                bestValue = - Integer.MAX_VALUE /2;
                c.and(target.gt(bestValue));
        } else  c.and(target.lt(bestValue));
        Solution solution = null;
```

Inizialmente il metodo deve occuparsi di definire l'obiettivo e la variabile coinvolta. Se la variabile non ha un nome glie ne viene assegnato uno di default, quindi si verifica che questa appartenga alle variabili del problema, contrariamente viene aggiunta.

L'obiettivo della funzione può essere di minimizzare o massimizare il valore della variabile data, a questo propostio viene creato un valore pessimo `bestValue` che rappresenta il minimo o il massimo valore rappresentabile a seconda del parametro `Objective` passato. Il nome della variabile intera `bestValue` potrebbe essere fuorviante, il valore infatti è quello pessimo per l'obiettivo ma è considerato l'attuale valore migliore. Ad ogni passo se il nuovo valore `newValue` è migliore, `bestValue` verrà sostituito da questo.

A supporto di questo approccio viene utilizzato un vincolo JSetL `c` relativo alla variabile obiettivo. Se tale variabile deve essere massimizzata il valore `target` deve essere maggiore del miglior valore attuale, viceversa dovrà essere minore.

Listato 5.9: `findOptimalSolution`, il ciclo.

```
while(jsetlSolver.check(c)) {
        if(getMaxNumberOfSolutions() > 0 &&
                        numberOfSolutions >=
                            getMaxNumberOfSolutions())
                break;
        if(getTimeLimit() > 0) {
                if (System.currentTimeMillis() - startTime >
                      getTimeLimit())
                        break;
        }
        numberOfSolutions++;
        try {
            solution = new Solution(this,
                    numberOfSolutions++);
        } catch (Failure e) {
                e.printStackTrace();
                return null;
        }
        .
        .
        .
```

Terminata l'inizializzazione si passa al ciclo per la ricerca vera e propria. In questa parte del codice si evidenziano le condizioni di ingresso e di uscita dal ciclo.

Le istruzioni all'interno del `while` continuano fintanto che il vincolo propagato `c` nel solver JSetL mediante la funzione `check`, è soddisfacibile. Internamente al ciclo si trova quindi una nuova soluzione e viene incrementato di conseguenza il numero delle soluzioni.

Le possibili condizioni di uscita sono le seguenti:

1. `check` ritorna `false`, ovvero non esistono soluzioni con l'attuale propagazione del vincolo `c`;
2. è impostato un numero limite di soluzioni ed è stato raggiunto o superato;
3. è impostato un tempo limite ed è stato raggiunto o superato.

Listato 5.10: `findOptimalSolution`, verifica obiettivo.

```
int newValue;
if(maximize) {
        newValue = solution.getMin(obj.getName());
        if(bestValue < newValue)
                bestValue = newValue;
        c.and(target.gt(bestValue));
}
else {
        newValue = solution.getMax(obj.getName());
        if(bestValue > newValue)
                bestValue = newValue;
        c.and(target.lt(bestValue));
}
```

Questa è la parte, interna al ciclo, che verifica l'obiettivo ed aggiorna l'eventuale soluzione ottima, inoltre aggiorna il vincolo che verrà poi propagato nella successiva esecuzione della `check`. Ad ogni passaggio viene memorizzato il valore della corrente soluzione nell'intero `newValue`, a seconda che l'obiettivo sia massimizzare o minimizzare, se il nuovo valore è migliore di `bestValue`, questo viene sostituito.

Ogni volta viene aggiornato il vincolo relativo alla variabile obiettivo: se occorre massimizzare il vincolo sarà del tipo `target` > `newValue`, altrimenti del tipo `target` < `newValue`.

Listato 5.11: `findOptimalSolution`, termine.

```
if (solution != null)
   log("Optimal solution is found. Objective: " +solution.
        getValue(objectiveVar.getName()));
   return solution;
}
```

Metodo `findAllSolutions`

L'ultimo metodo implementato per la ricerca delle soluzioni è inerente alla ricerca di tutte le possibili soluzioni del problema. Come visto per

`findSolution` questo metodo applica prima un'euristica per le variabili ed i valori, poi si occupa della ricerca.

Listato 5.12: `findAllSolutions`, inizializzazione.

```
public Solution[] findAllSolutions() {
  ArrayList<Solution> array = new ArrayList<Solution>();
  applyHeuristic();
  long startTime = System.currentTimeMillis();
  if(jsetlSolver.check() && (this.getMaxNumberOfSolutions()
      > 0 || this.getMaxNumberOfSolutions() == -1)) {

          .
          .  // Ricerca di tutte le soluzioni.
          .

  }
  else return null;
  .
  .
  .
```

Inizialmente viene creato un nuovo `ArrayList` che conterrà le soluzioni trovate nel processo di risoluzione. Vengono applicate le euristiche su ogni variabile del problema e quindi viene inizializzato il timer per il controllo tempo limite.

A questo punto il metodo è pronto ad iniziare la ricerca, mediante l'istruzione `if` viene controllato che il problema sia soddisfacibile (`check`) e il numero di soluzioni sia coerente con la richiesta. All'interno del ciclo è definita la ricerca delle soluzioni.

Listato 5.13: `findAllSolutions`, ricerca.

```
        do {
            if (getTimeLimit() > 0) {
                if (System.currentTimeMillis() - startTime >
                getTimeLimit())
                    break;
            }
            Solution solution;
            try {
                solution = new Solution(this,
                numberOfSolutions++);
            } catch (Failure e) {
                solution = null;
                e.printStackTrace();
            }
            array.add(solution);
        } while(jsetlSolver.nextSolution() &&
                ((this.getMaxNumberOfSolutions() >
```

```
                numberOfSolutions) ||
                this.getMaxNumberOfSolutions() == -1));
```

Questa parte di codice, interna al ciclo, genera una soluzione mediante il solito costruttore quindi l'aggiunge al vettore delle soluzioni. Si entra nel ciclo solo se una soluzione è ammissibile.

Le condizioni di uscita del ciclo sono le seguenti:

1. non esiste una nuova soluzione e quindi l'ultima valida è effettivamente quella aggiunta nell'array;
2. si è superato il numero di soluzioni massime e quindi l'ultima soluzione trovata era quella con numero massimo;
3. si è raggiunto o superato il tempo limite.

A questo punto, poiché il metodo richiede che venga restituito un array di soluzioni (e non un `ArrayList`), questo viene creato e restituito:

Listato 5.14: `findAllSolutions`, calcolo del risultato.

```
  Solution[] result = new Solution[array.size()];
  for (int i = 0; i < result.length; i++) {
    result[i] = array.get(i);
  }
  return result;
}
```

Metodi ausiliari

Questa parte della sezione è dedicata alle funzioni ausiliarie all'interno della classe, tra i metodi più rilevanti si evidenziano:

```
private void applyHeuristic();
public void addJSetLConstraint(JSetL.Constraint c);
```

Ogni metodo sopra elencato è importante perché influenza il solver aggiungendo vincoli al problema.

Metodo `applyHeuristic`

Questo metodo aggiunge al solver (`jsetlSolver`) i vincoli che hanno a che fare con le euristiche di scelta delle variabili e dei valori che si vogliono assegnare.

Listato 5.15: `applyHeuristic`.

```
private void applyHeuristic() {
  Vector<javax.constraints.SearchStrategy> strategies =
```

```
      getSearchStrategies();
  if (strategies.isEmpty()) {
    SearchStrategy strategy = (SearchStrategy)
        getSearchStrategy();
    strategy.label();
    strategy.labelSets();
  }
  else {
      for (int i = 0; i < strategies.size(); i++) {
        SearchStrategy strategy = strategies.elementAt(i);
        ((SearchStrategy) strategy).label();
        ((SearchStrategy) strategy).labelSets();
      }
   }
}
```

Il metodo carica semplicemente le strategie di ricerca in un `Vector`. Se nessuna strategia è stata precedentemente definita ne viene creata una di default mediante il metodo `getSearchStrategy`, quindi vengono chiamati i metodi di labeling della classe `SearchStrategy`. Se sono state caricate più di una strategia, per ognuna di esse viene effettuato il labeling.

Metodo `addJSetLConstraint`

Il metodo in questione è utilizzato per aggiungere un vincolo JSetL direttamente al solver `jsetlSolver`.

Listato 5.16: `addJSetLConstraint`.

```
public void addJSetLConstraint(JSetL.Constraint c) {
       jsetlSolver.add(c);
}
```

5.3 Interfaccia `SearchStrategy`

La specifica JSR-331 utilizza il concetto di Search Strategy per permettere ad un utente di scegliere tra differenti algoritmi di ricerca forniti dalle differenti implementazioni. Le strategie di ricerca sono utilizzate dai metodi dei solver che si occupano di trovare una soluzione.

Una strategia di ricerca dovrebbe conoscere tutte le variabili di cui si dovrà assegnare un valore e potrebbero essere necessari selettori esterni per le variabili ed i valori.

Ogni solver dovrebbe fornire almeno una strategia di ricerca di default da utilizzare nei costruttori del `Solver`.

5.3.1 Lista d'esecuzione

La lista d'esecuzione delle strategie di ricerca consente agli utenti di mescolare le strategie per i diversi tipi di variabili e di controllare l'ordine di assegnamento dei valori. Ad esempio, per la pianificazione e per problemi di allocazione delle risorse, un utente può decidere prima di programmare tutte le attività e poi assegnare le risorse alle attività già programmate. Ma può anche decidere di assegnare prima le risorse e solo allora di programmare le attività in base alla disponibilità delle risorse.

È stato quindi fornito un sistema per cui l'utente può specificare diverse strategie per insiemi di variabili differenti. Il metodo della classe `Solver`

SearchStrategy newSearchStrategy();

restituisce una nuova istanza per la strategia di default della particolare implementazione JSR-331 in uso. L'utente può quindi specificare su questa strategia delle euristiche per le variabili ed i valori e quindi aggiungere la nuova strategia modificata alla lista d'esecuzione.

Di seguito si mostra un semplice esempio.

Listato 5.17: due differenti strategie.

```
Solver solver = problem.getSolver();
SearchStrategy typeStrategy = solver.getSearchStrategy();
typeStrategy.setVars(types);
SearchStrategy countStrategy = solver.newSearchStrategy();
countStrategy.setVars(counts);
countStrategy.setVarSelectorType(VarSelectorType.MIN_DOMAIN)
    ;
solver.addSearchStrategy(countStrategy);
solution = solver.findSolution();
```

Sono presenti altri metodi utili per specificare strategie di ricerca senza dover esplicitamente creare nuove istanze di `SearchStrategy`. Il metodo `addSearchStrategy` ad esempio supporta differenti combinazioni di parametri `Var[]`, `VarSelector` e `ValueSelector`. Il codice di cui sopra può essere riscritto in un modo più conciso:

```
Solver solver = problem.getSolver();
solver.getSearchStrategy().setVars(types);
solver.addSearchStrategy(counts, VarSelectorType.MIN_DOMAIN)
    ;
solution = solver.findSolution();
```

5.3.2 Variable Selector

JSR-331 specifica un insieme di selettori di variabili standard che possono essere usati dall'utente per personalizzare la strategia. Queste euristiche sono

definite dall'interfaccia standard `VariableSelector` utilizzando il seguente tipo enumerativo:

```
static public enum VarSelectorType {
    INPUT_ORDER,
    MIN_VALUE,
    MAX_VALUE,
    MIN_DOMAIN,
    MIN_DOMAIN_MIN_VALUE,
    MIN_DOMAIN_RANDOM,
    RANDOM,
    MIN_DOMAIN_MAX_DEGREE,
    MIN_DOMAIN_OVER_DEGREE,
    MIN_DOMAIN_OVER_WEIGHTED_DEGREE,
    MAX_WEIGHTED_DEGREE,
    MAX_IMPACT,
    MAX_DEGREE,
    MAX_REGRET,
    CUSTOM
}
```

Non tutte queste euristiche devono essere fornite da ogni implementazione concreta di JSR-331.

5.3.3 Value Selector

JSR-331 specifica un insieme di selettori di valori standard che possono essere usati dall'utente per personalizzare la strategia. Queste euristiche sono definite dall'interfaccia standard `ValueSelector` utilizzando il seguente tipo enumerativo:

```
static public enum ValueSelectorType {
    IN_DOMAIN,
    MIN,
    MAX,
    MIN_MAX_ALTERNATE,
    MIDDLE,
    MEDIAN,
    RANDOM,
    MIN_IMPACT,
    CUSTOM
}
```

Non tutte queste euristiche devono essere fornite da ogni implementazione concreta di JSR-331.

Euristiche in JSetL

Prima di passare alla descrizione della classe `SearchStrategy` che fornisce l'implementazione per le strategie di ricerca, occorre sottolineare che

l'approccio alle strategie di selezione delle variabili e dei valori è differente da quello ipotizzato nella specifica.

La specifica presuppone che le variabili siano selezionate prima, per poi essere passate ad un risolutore che vi assegni un valore; infatti all'interno del `package javax.constraints.impl.search.selectors` fornito dalla specifica sono definiti vari selettori predefiniti.

JSetL si basa su un sistema molto differente, sfruttando il solver ed aggiungendo vincoli di labeling, come già accennato nel capitolo 3 e come verrà sottolineato poco più avanti. Per questa ragione, le euristiche di cui sopra sono state mappate su quelle definite in JSetL per il labeling.

Nota Bene. Di contro, questo approccio rende impossibile per l'utente definire una propria strategia come invece la specifica descritta in [1] richiederebbe.

5.4 Classe SearchStrategy

La classe `SearchStrategy` implementa l'interfaccia standard estendendo la classe `AbstractSearchStrategy` dell'implementazione comune.

```
public class SearchStrategy extends AbstractSearchStrategy {
```

Questo approccio permette di ereditare tutti i metodi astratti puri da implementare e, dove necessario, ridefinire i metodi non astratti.

5.4.1 Classe AbstractSearchStrategy

La classe astratta `AbstractSearchStrategy` estende la classe `CommonBase` descritta nella sezione 4.3, poiché come accennato precedentemente, la specifica si aspetta che ogni implementazione concreta disponga di una classe che rappresenti la strategia.

JSetL non ha questa caratteristica, tuttavia si è deciso di utilizzare la classe astratta poiché fornisce molti attributi e metodi utili.

Attributi

Gli attributi della classe `AbstractSearchStrategy` sono sei:

```
        Solver solver;
        protected Var[] vars;
        protected VarReal[] varReals;
//      protected VarSet[] varSets;
        protected VarSelector varSelector;
        protected ValueSelector valueSelector;
        SearchStrategyType type;
```

Il primo è il riferimento al solver a cui la strategia è legata. Seguono quindi gli array delle variabili della strategia, come si può notare l'array per le variabili insiemistiche è commentato. Durante lo studio della specifica si è cambiato approccio per quanto riguarda le variabili insiemistiche.

Gli attributi `varSelector` e `valueSelector` specificano le euristiche della strategia, mentre `type` ne definisce il tipo.

Nota. Per quanto riguarda l'implementazione concreta basata su JSetL due attributi verranno ignorati: `varReals` e `type`. Il primo perché attualmente non sono supportate variabili reali, mentre il secondo per la differenza di apporccio nell'applicazione delle euristiche, di fatto ogni strategia sarà di tipo `SearchStrategyType.DEFAULT`.

Metodi utili

I metodi di utilità generica o che non hanno bisogno di un'implementazione specifica come quella fornita da JSetL, vengono implementati direttamente all'interno della classe `AbstractSearchStrategy`. Tra quelli più utilizzati si evidenziano:

- **public** Solver getSolver();

  Restituisce il solver associato alla strategia di ricerca.

- **public** Var[] getVars();

  Restituisce un array di variabili associate alla strategia d'invocazione.

- **public void** setVars(Var[] vars);

  Associa un array di variabili alla strategia d'invocazione.

- **public** ValueSelector getValueSelector();

  Restituisce l'euristica di selezione del valore da assegnare alle variabili.

- **public** VarSelector getVarSelector();

  Restituisce l'euristica di selezione delle variabili.

- **public void** setValueSelector(ValueSelector valueSelector);

  Imposta l'euristica di selezione del valore da assegnare alle variabili.

- **public void** setVarSelector(VarSelector varSelector);

  Imposta l'euristica di selezione delle variabili.

Altri metodi, come ad esempio `setVars` sulle variabili insiemistiche, sono stati omessi poiché non utilizzati dall'implementazione basata su JSetL oppure perché sovrascritti dalla stessa, e verranno quindi definiti nella prossima sezione.

5.4.2 Implementazione

Come già sottolineato nel capitolo, JSetL ha un approccio diverso da quello previsto dalla specifica, l'implementazione quindi non si avvale appieno delle funzionalità della classe `AbstractSearchStrategy` e della `CommonBase`. Inoltre, utilizzando un approccio diverso anche per l'implementazione delle variabili insiemistiche si è dovuto specializzare ulteriormente la classe con attributi e metodi.

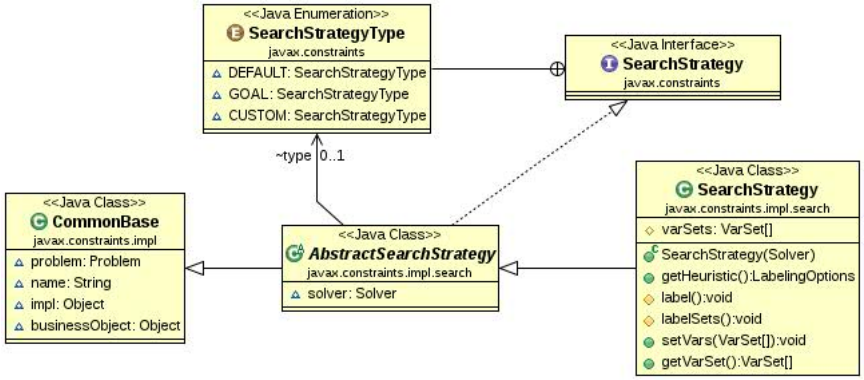

Figura 5.2: Class Diagram di `SearchStrategy`.

Attributi

Poiché nel problema (nella classe `Problem`) è stato aggiunto un array di variabili insiemistiche legate al problema, anche nella classe `SearchStrategy` è stato aggiunto tale array.

```
    protected VarSet[] varSets;
```

L'attributo `varSets` rappresenta le variabili insiemistiche associate alla strategia.

Costruttori

Vi è un solo costruttore di classe, che prende come unico argomento il solver a cui è associato:

```
public SearchStrategy(Solver solver) {
        super(solver);
        varSets = ((Problem) getProblem()).getSetVars();
}
```

Questo chiama il costruttore della classe base, quindi inizializza i due array mediante la funzione `getProblem` che restituisce il problema a cui la strategia è associata e quindi da questo ottiene le variabili con il relativo metodo `getSetVars`.

Euristiche

In questa parte della sezione verranno specificati i metodi che codificano ed applicano le euristiche di scelta per assegnare valori alle variabili e per scegliere l'ordine di assegnamento delle stesse.

Metodo `getHeuristic`

Questa funzione è una vera e propria mappa per le euristiche, ovvero associa ogni `VarSelector` o `ValueSelector` della specifica ad un'euristica di JSetL. La funzione è divisa in due parti, la prima si occupa delle variabili, la seconda dei valori.

Listato 5.18: `getHeuristic`, le variabili.

```
public LabelingOptions getHeuristic() {
        LabelingOptions lop = new LabelingOptions();
        VarSelector varSelector = getVarSelector();
        if (varSelector != null) {
                VarSelectorType varType = varSelector.
                    getType();
                switch(varType) {
                case INPUT_ORDER:
                        lop.var = VarHeuristic.LEFT_MOST;
                        break;
                case MIN_VALUE:
                        lop.var = VarHeuristic.MIN;
                        break;
                case MAX_VALUE:
                        lop.var = VarHeuristic.MAX;
                        break;
                case RANDOM:
                        lop.var = VarHeuristic.RANDOM;
                        break;
                case MIN_DOMAIN:
                        lop.var = VarHeuristic.FIRST_FAIL;
                        break;
                default:
                        lop.var = VarHeuristic.LEFT_MOST;
                        break;
                }
        }
        else lop.var = VarHeuristic.LEFT_MOST;
```

Innazitutto viene inizializzata un'istanza della classe `LabelingOptions`, questa è una classe di supporto JSetL utile per definire vincoli di labeling sulle variabili, ovvero le euristiche.

Quindi, mediante il metodo della classe base `getVarSelector`, viene memorizzato il selettore per le variabili. Con lo `switch` infine, a seconda del

caso viene impostato il valore di `lop.var` opportuno. Se nessuno dei casi combacia con quelli identificati viene imposto di default un'euristica di tipo leftmost, come richiesto dalla specifica.

Listato 5.19: `getHeuristic`, i valori.

```
    ValueSelector valueSelector = getValueSelector();
    if (valueSelector != null) {
        ValueSelectorType valueType = valueSelector.getType
            ();
        switch(valueType) {
        case MIN:
                lop.val = ValHeuristic.GLB;
                break;
        case MAX:
                lop.val = ValHeuristic.LUB;
                break;
        case MIDDLE:
                lop.val = ValHeuristic.MID_MOST;
                break;
        case MEDIAN:
                lop.val = ValHeuristic.MEDIAN;
                break;
        case RANDOM:
                lop.val = ValHeuristic.EQUI_RANDOM;
                break;
        default:
                lop.val = ValHeuristic.GLB;
                break;
        }
        return lop;
```

Nella parte finale dell'algoritmo si compiono gli stessi passaggi, ma per la scelta dei valori. Quindi viene restituito il `LabelingOptions` ottenuto.

Metodo `label`

Il metodo `label` è il primo metodo implementato e descritto che si occupa di applicare un'euristica. Questo opera sulle variabili intere della strategia.

Listato 5.20: `label()`.

```
protected void label() {
        if (vars == null || vars.length == 0)
                return;
        SolverClass sc = ((Solver) getSolver()).
            getSolverClass();
        IntLVar[] vec = new IntLVar[vars.length];
        for (int i = 0; i < vars.length; i++) {
                vec[i] = (IntLVar) vars[i].getImpl();
```

```
        }
        sc.add(IntLVar.label(vec, getHeuristic()));
}
```

Dopo aver controllato che il vettore delle variabili intere non sia nullo (contrariamente il metodo viene interrotto senza errori) viene memorizzato un riferimento al solver JSetL e viene inizializzato un vettore di `IntLVar` con le implementazioni concrete delle variabili intere della strategia.

A questo punto viene caricata la strategia mendiante il metodo `getHeuristic` vista sopra, ed aggiunto al solver JSetL un vincolo speciale di labeling creato mediante la funzione `label(IntLVar[], LabelingOptions)` definita in [2] e di cui si è accennato nel capito 3.

Metdo `labelSets`

`labelSets` si comporta come il metodo per le variabili intere, cambiano solo i tipi, tecnica e metodi sono i medesimi, si da la definizione per completezza.

Listato 5.21: `labelSets()`.

```
protected void labelSets() {
        if (varSets == null || varSets.length == 0)
                return;
        SolverClass sc = ((Solver) getSolver()).
            getSolverClass();
        SetLVar[] vec = new SetLVar[varSets.length];
        for (int k = 0; k < vars.length; k++) {
                vec[k] = (SetLVar) varSets[k].getImpl();
        }
        sc.add(SetLVar.label(vec, getHeuristic()));
}
```

5.5 Interfaccia `Solution`

L'interfaccia standard `Solution` specifica le soluzioni che possono essere generate dai metodi della classe `Solver` e dagli iteratori. Questa interfaccia è completamente implementata nella common implementation definita dalla classe `BasicSolution`, ogni implementazione concreta può estenderla mediante la sottoclasse `Solution`.

Un'istanza della soluzione contiene le copie di tutte le variabili legate al problema che sono state usate da una strategia di ricerca che ha creato tale soluzione.

Metodi

I metodi e le funzioni definiti nell'interfaccia sono i seguenti:

- **public** Var[] getVars();

  Restituisce le variabili della soluzione legate al problema.

- **public** Var getVar(String name);

  Restituisce la variabile con il nome dato dalla stringa `name`. Lancia un'eccezione se la variabile non esiste.

- **public int** getValue(String name);

  Restituisce il valore della variabile con il nome dato dalla stringa `name`. Lancia un'eccezione se la variabile non esiste.

- **public boolean** isBound();

  Restituisce `true` se ogni variabile della soluzione è bound (ovvero il suo dominio è un singolo valore), `false` altrimenti.

- **public boolean** isBound(String name);

  Restituisce `true` se la variabile con il nome dato è "bound", `false` altrimenti.

- **public int** getSolutionNumber();

  Restituisce il numero associato alla soluzione.

- **public void** setSolutionNumber(**int** number);

  Assegna un valore intero al numero associato alla soluzione.

- **public void** log();

  Questo metodo stampa la soluzione, è una variante della medesima definita nell'interfaccia `Problem`.

- **public** Solver getSolver();

  Restituisce il solver associato alla soluzione.

5.6 Classe `Solution`

La classe `Solution` implementa l'interfaccia standard `Solution` estendendo la classe `BasicSoltion` dell'implementazione comune.

```
public class Solution extends BasicSolution {
```

Questo approccio permette di ereditare tutti i metodi della classe base, che non è astratta, ma una vera e propria implementazione di base della soluzione. Verrà quindi brevemente descritta.

5.6.1 Classe BasicSolution

`BasicSolution` rappresenta l'implementazione di base della soluzione di un problema, viene fornita nella specifica dal file `BasicSolution.java` contenuto nel package `javax.constraints.impl.search`. È una classe base, non estende quindi nessun'altra classe

```
public class BasicSolution implements Solution {
```

Questa classe implementa totalmente quanto specificato nell'interfaccia, pertanto non vi sarebbe il bisogno di specializzarla. Tuttavia poiché nella implementazione basata su JSetL sono stati introdotti alcuni aspetti che si discostano dalle specifiche individuate dallo standard, si è dovuto trattare in modo specializzato quest'ultimi.

Attributi

Gli attributi della classe base sono i seguenti:

```
        Solver          solver;
        int             solutionNumber;
        ResultInt[]     intResults;
        ResultReal[]    realResults;
        ResultSet[]     setResults;
```

`solver` rappresenta, come di consueto, il solver da cui la soluzione è stata generata. L'intero `solutionNumber` ne rappresenta il numero progressivo.

Gli utlimi tre attributi meritano un discorso più dettagliato, questi infatti sono degli array di un tipo definito nello stesso file della classe `BasicSolution` e di fatto rappresentano internamente il risultato.

Classe ResultInt

`ResultInt` rappresenta una delle variabili intere risolta dal solver.

```
class ResultInt {
        String varName;
        int value;
        inte min;
        int max;
        boolean bound;
}
```

Ha alcuni attributi che ne rappresentano il nome, il valore attuale (se unico), il valore minimo e quello massimo, quindi un booleano che indica se la variabile risolta è bound, ovvero ha un unico valore possibile. Viene fornita anche una funzione `toString`.

Classe `ResultReal`

Questa classe è praticamente identica a quella definita per le variabili intere (cambiano ovviamente i tipi) e, poiché JSetL non supporta i vincoli sui reali, verrà omessa la descrizione.

Classe `ResultSet`

Anche in questo caso verrà omessa la descrizione della classe, poiché l'approccio adottato dallo standard è molto differente dall'implementazione concreta che JSetL fornisce per gli insiemi. Tuttavia a differenza della struttura sui reali, questa verrà ridefinita nel file della classe `Solution`.

Metodi

Si evidenziano alcuni metodi non presenti tra quelli già menzionati nelle interfacce che manipolano le sopracitate classi di risultati:

- **private int** getIndexOfInt(String name);

  Restituisce l'indice della variabile risolta di nome `name`. In caso la variabile non esista lancia un'eccezione.

- ResultInt createResult(Var var);

  Questa importante funzione, presa una variabile intera (risolta dal solver), genera e restituisce un istanza della classe `ResultInt`.

Al momento della realizzazione dell'implementazione concreta lo standard non fornisce il supporto alle variabili reali ed insiemistiche. Ovvero nel costruttore di `BasicSolution` non è ancora implementato l'algoritmo per generare variabili risolte di tipo `ResultReal` e `ResultSet`. Analogamente non è stata introdotta la funzione `createResult` per le suddette variabili. Come si vedrà nella prossima sezione questi meccanismi per le variabili insiemistiche sono state aggiunte nella specializzazione `Solution`.

5.6.2 Implementazione

La parte implementativa dell'interfaccia `Solution` verte esclusivamente sul supporto delle variabili insiemistiche. Per raggiungere tale obiettivo si è ridefinita la classe `ResultSet`, si è aggiunto un attributo come nella classe base, si è definita la funzione `createResult` per variabili insimistiche e si è specializzato il costruttore.

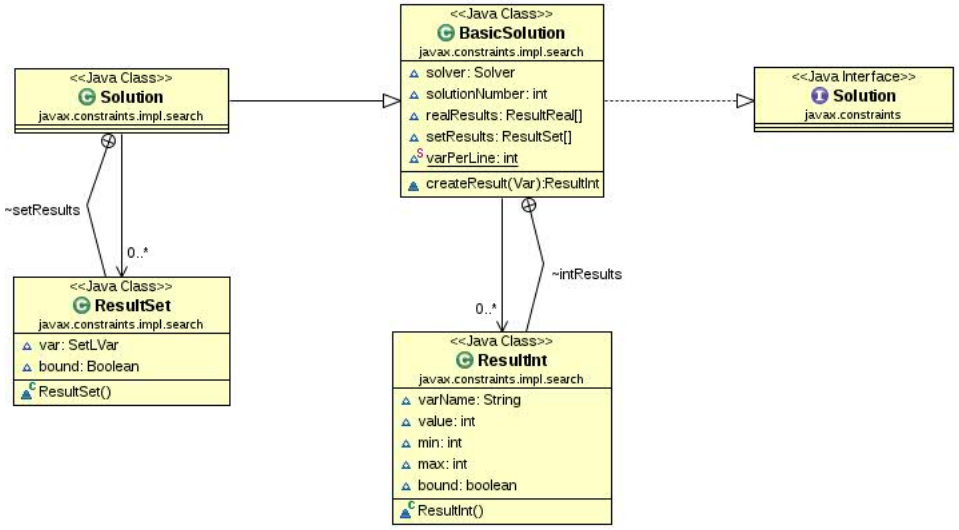

Figura 5.3: Class Diagram di `Solution`.

Attributi

L'unico attributo di classe è `setResults`, un array di tipo `ResultSet`. Questo attributo si è dovuto aggiungere causa l'impossibilità di accedere alla controparte della classe base. Come questa, conterrà un array di variabili risolte di tipo insiemistico.

Classe `ResultSet`

Come anticipato nella sezione precedente, la classe ausiliaria `ResultSet` è stata ridefinita nel seguente modo:

Listato 5.22: `ResultSet`.

```
class ResultSet {
        SetLVar var;
        Boolean bound;

        public String toString() {
                return var.toString();
        }

        public String getName() {
                return var.getName();
        }
}
```

Questa classe è stata creata sul modello di quelle implementate nel file dell'implementazione comune `BasicSolution`, ma sfrutta la classe `SetLVar`.

Metodo `createResult`

Anche la funzione `createResult` è stata implementata seguendo l'approccio di quella per variabili intere nella classe base.

Listato 5.23: `createResult` per variabili insiemistiche.

```
private ResultSet createResult(VarSet var) {
        ResultSet result = new ResultSet();
        if(var.isBound())
                result.bound = true;
        else result.bound = false;
        result.var = (SetLVar) ((VarSet) var).getImpl();
        return result;
```

Data una variabile insiemistica `var`, questo metodo crea una nuova istanza della classe `ResultSet` e quindi ne assegna i valori. Se la data variabile è bound l'attributo `bound` della nuova variabile risolta viene impostato a `true`, altrimenti a `false`. Quindi nel campo `var` viene copiata l'implementazione concreta relativa alle variabili insimistiche.

Costruttore

La parte più delicata dell'implementazione è il costruttore. La descrizione di questo è stata lasciata per ultima poiché sfrutta il tipo ed il metodo definiti sopra.

Listato 5.24: `createResult` per variabili insiemistiche.

```
public Solution(Solver solver, int solutionNumber) throws
    Failure {
        super(solver, solutionNumber);
        Vector<javax.constraints.SearchStrategy>
            searchStrategies =
            ((AbstractSolver)solver).getSearchStrategies();
        Vector<VarSet> strategyVars = new Vector<VarSet>();
        for (javax.constraints.SearchStrategy strategy :
            searchStrategies) {
                VarSet[] vars = ((SearchStrategy) strategy).
                    getVarSet();
                if (vars == null)
                        return;
                for (VarSet var : vars) {
                        if (!strategyVars.contains(var))
                                strategyVars.add(var);
                }
        }
        setResults = new ResultSet[strategyVars.size()];
        Iterator<VarSet> iter = strategyVars.iterator();
        int i = 0;
```

```
        while(iter.hasNext()) {
                VarSet var = iter.next();
                setResults[i++] = createResult(var);
        }
}
```

Analogamente a quanto visto finora si è seguito l'approccio dell'implementazione di base. Brevemente, il costruttore chiama quello della classe base, quindi applica gli stessi costrutti per creare le variabili risolte ed inserirle nell'array dei risultati (specializzato).

Si evidenziano due fasi: nella prima vengono caricate tutte le strategie e per ogni strategia si salvano le variabili insimistiche coinvolte; nella seconda fase viene creato l'array dei risultati che viene poi riempito tramite la funzione `createResult`.

5.7 Interfaccia `SolutionIterator`

L'interfaccia standard `SolutionIterator` consente all'utente di trovare e navigare tra diverse soluzioni ed eseguire differenti azioni specifiche sulle soluzioni trovate.

L'utilizzo, come si intende nella specifica, è presentato in questo esempio:

```
SolutionIterator iter = solver.solutionIterator();
while(iter.hasNext()) {
  Solution solution = iter.next();
  ...
}
```

I metodi previsti dall'interfaccia sono solo due: `hasNext` che controlla se c'è una nuova soluzione e `next` che genera la nuova soluzione.

5.7.1 Classe `BasicSolutionIterator`

L'implementazione comune fornisce una classe di base che implementa l'interfaccia `SolutionIterator`. Questa non è stata presa in considerazione per l'implementazione concreta JSetL e non verrà quindi descritta.

5.8 Classe `SolutionIterator`

La classe `SolutionIterator` implementa l'interfaccia `SolutionIterator` direttamente, senza appoggiarsi a classi di base.

```
public class SolutionIterator implements javax.constraints.
    SolutionIterator {
```

Attributi

Si definiscono i seguenti attributi:

```
        Solver solver;
        int solutionNumber;
        Boolean hasNext;
        Boolean checked = false;
        private boolean firstcall = true;
```

Come di consueto per le classi che rappresentano la soluzione, il primo attributo è il `solver` legato alla soluzione, quindi segue il numero identificativo della stessa. `hasNext` è un valore booleano che specifica se è presente una nuova soluzione, `checked` invece è utilizzato per verificare che la funzione `hasNext` sia stata invocata prima della funzione `next`. L'ultimo attributo verifica che la funzione `hasNext` sia o meno invocata la prima volta.

Costruttore

L'unico costruttore richiesto è quello con parametro di tipo `Solver`:

Listato 5.25: il costruttore di `SolutionIterator`.

```
public SolutionIterator(Solver s) {
        solver = (Solver) s;
        hasNext = true;
        solutionNumber = 0;
}
```

Questo imposta l'attributo `solver` con il solver che ha generato la soluzione, imposta di default l'attributo `hasNext` a `true` ed il numero di soluzione a 0.

Metodo `hasNext`

Il metodo `hasNext` controlla che esista una soluzione successiva all'interno dello stato del solver. Per fare ciò cerca una soluzione con il metodo `findSolution` con parametro `ProblemState.RESTORE`, in modo tale da permettere al solver di mantenere i punti di scelta ed iterare il procedimento.

Listato 5.26: `hasNext`.

```
public boolean hasNext() {
        checked = true;
        if (firstcall) {
                firstcall = false;
                Solution solution = solver.findSolution(
                    ProblemState.RESTORE);
                if (solution == null) {
```

```
                    hasNext = false;
                    return false;
                }
                return true;
            }
            return solver.hasNext();
```

Inizialmente imposta `checked` al valore `true`, quindi se è la prima chiamata `firstcall` viene impostata a `true` quindi cerca la soluzione. Se la soluzione trovata non è nulla restituisce `true`, altrimenti imposta l'attributo `hasNext` a `false` e restituisce `false`.

Nel caso in cui la chiamata al metodo non è la prima, il risultato viene demandato all'omonima funzione della classe `Solver` che semplicemente sfrutta la funzione `nextSolution` del solver JSetL.

Nota Bene. È importante sottolineare che l'unico caso in cui viene usata la funzione `findSolution` con il parametro `ProblemState.RESTORE` all'interno dell'implementazione comune è nel relativo metodo `hasNext` della classe `BasicSolutionIterator`. Questo perché, secondo l'approccio utilizzato nella specifica JSR-331, `ProblemState` modella il controllo per il *Backtracking*.

In JSetL il backtracking è insito nel solver e, per sfruttarlo, si utilizza la funzione `nextSolution`, senza bisogno di un punto di scelta. Per questo motivo, nella classe `Solver` il metodo `findSolution` utilizza `nextSolution` nel caso il parametro passato sia `RESTORE`.

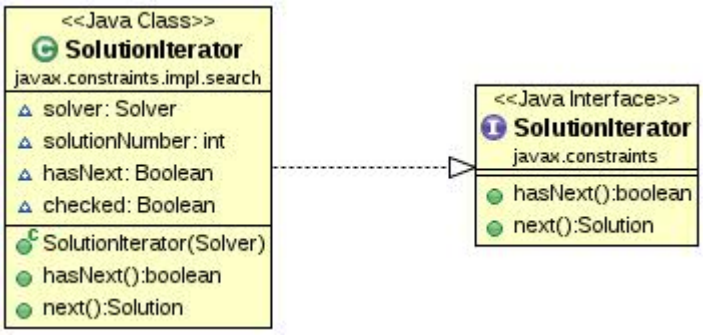

Figura 5.4: Class Diagram di `SolutionIterator`.

Metodo `next`

Questo metodo crea la soluzione trovata dal metodo `hasNext` e la restituisce. È quindi necessario che ogni volta che questo viene chiamato sia stato precedentemente invocato `hasNext`. A questo proposito viene sfruttato l'attributo `checked`: inizialmente il medoto controlla che questo

sia `true` (ovvero `hasNext` è stato utilizzato), quindi prima dell'uscita lo reimposta a `false`, in modo tale che `hasNext` debba essere richiamato nuovamente in seguito.

Listato 5.27: `next`.

```
public Solution next() {
        if (!checked)
                throw new RuntimeException("Cannot use
                    SolutionIterator.next() " + "before
                    checking the hasNext() returned true");
        Solution solution;
        try {
                solution = new Solution(solver,
                    solutionNumber);
                solution.setSolutionNumber(solutionNumber);
                solutionNumber++;
        } catch (Failure e) {
                e.printStackTrace();
                hasNext = false;
                return null;
        }
        checked = false;
        return solution;
}
```

All'interno del metodo viene creata la soluzione mediante il costruttore fornito dalla classe `Solution` che automaticamente genera le variabili risolte. Viene quindi aggiornato il numero di soluzione.

Capitolo 6

Conclusioni e lavori futuri

Il lavoro di tesi descritto verte sull'implementazione delle specifiche individuate dal JSR-331 mediante il pacchetto JSetL. Il modello di sviluppo utilizzato per il progetto può essere catalogato come *in cascata* in cui si sono eseguite più iterazioni delle seguenti fasi:

i Studio del documento di specifica [1].

ii Valutazione delle funzionalità offerte da JSetL.

iii Codifica.

iv Test e validazione.

Prima di passare alla descrizione delle fasi indicate occorre sottolineare che tutto il lavoro non si sarebbe potuto svolgere senza l'appoggio del Dr. Jacob Feldman, il quale, oltre ad un contatto diretto e grande disponibilità, ha fornito l'accesso al repository del progetto e uno spazio dedicato all'implementazione JSetL. Dal repository, mediante *SVN*, si è potuto scaricare tutto il materiale necessario per iniziare l'implementazione, inoltre il repository è stato uno strumento fondamentale per mantenere il progetto aggiornato.

Per quanto riguarda il punto i si è trattato di analizzare il documento di specifica fornito. Sono stati molto utili, al fine di comprendere meglio i requisiti dei vari metodi e delle classi, anche gli esempi presenti nel TCK (vedi cap. 2).

La valutazione di JSetL (punto ii) è stata facilitata dai numerevoli documenti (articoli, tesi, ...) presenti al Dipartimento di Matematica, nonché dal contatto diretto con persone che hanno lavorato attivamente al progetto, su tutti il prof. Gianfranco Rossi, promotore del progetto e sempre disponibile.

La fase di codifica ha portato alla realizzazione vera e propria delle classi descritte nei capitoli precedenti. Indubbiamente lo sforzo maggiore è stato

quello di mappare i requisiti di input e di output richiesti sulle funzionalità di JSetL in modo corretto ed efficace. In questa fase, oltre alla codifica delle classi, si sono realizzati anche i test di unità, ovvero delle prove d'esecuzione (white box) scritte appositamente per verifiche interne.

L'ultima fase riguarda il TCK che di fatto rappresenta l'unico test di conformità alle specifiche. Come accennato nel capitolo 2 il TCK è fornito direttamente dalle specifiche JSR-331 ed un'implementazione, per essere conforme allo standard, deve permettere a questi test di terminare con successo. Nelle varie iterazioni del processo di sviluppo si è utilizzato questo pacchetto per validare le unità introdotte e per poter quindi passare alla realizzazione del modulo successivo o rivalutare quello non soddisfacente. Oltre al TCK si è sviluppato un ulteriore pacchetto di test per verificare l'implementazione prodotta allo scopo di coprire il più possibile il codice. Tale attività è stata svolta nelle ultime iterazioni del processo, nel momento in cui il TCK ha dato un buon numero di successi.

Il lavoro svolto, essendo completamente modulare e documentato oltre che dal presente testo anche mediante una ricca documetazione JavaDoc, risulta facilmente estendibile e modificabile.

6.1 Sviluppi futuri

La specifica JSR-331, al momento dell'inizio del progetto, era in fase di valutazione da parte del Java Community Process. Durante la stesura del presente testo la specifica è stata accettata ed è passata da uno stato di richiesta allo stato di release finale. Tuttavia, come annunciato nel blog ([11]) del Dr. Jacob Feldman, questo rappresenta una milestone del progetto di standardizzazione e JSR-331 sarà sviluppato ulteriormente. Per tale motivo uno dei lavori futuri sarà quello di mantenere allineata l'implementazione basata su JSetL con le specifiche. Si può notare che l'implementazione fornita è solo una delle possibili, sarebbe possibile analizzare più in dettaglio il pacchetto JSetL e le sue funzionalità al fine di migliorare l'implementazione attuale. È anche doveroso sottolineare che durante lo sviluppo del progetto sono emerse funzionalità richieste che si è dovuto aggiungere e quindi anche il JSR-331 rappresenta uno spunto per migliorare JSetL stesso. Inoltre essendo JSetL un progetto didattico, è soggetto naturalmente a frequenti cambiamenti e sarà quindi importante studiare la possibilità di utilizzare nuove funzionalità. Infine si evidenzia la possibilità di produrre un report, in fase di stesura, nella forma di un Rapporto Tecnico del Dipartimento di Matematica, che possa guidare nell'implementazione delle specifiche anche altri solver.

Appendice A

Il Vincolo di Cardinalità

A.1 Preliminari sul vincolo di cardinalità

L'interfaccia `Problem` della specifica JSR-331 introduce dei metodi convenzionali per la creazione di vincoli che abbiano a che fare con la cardinalità di certi valori in un array di variabili vincolate. Questi vincoli contano quanto spesso certi valori sono assegnati alle variabili nell'array dato. La variabile di cardinalità risulta quindi vincolata al numero di quelle variabili dell'array che siano a loro volta vincolate da uno specifico valore.

Vediamo quindi un metodo della classe `Problem` che genera un vincolo di cardinalità:

Constraint postCardinality(Var[] vars, **int** cardValue, String oper, **int** value)

Questo metodo crea, inserisce nel solver e restituisce un nuovo vincolo di cardinalità tale che "Card(`vars`, `cardValue`) `oper` `value`".

Card(`vars`, `cardValue`) denota una variabile vincolata ad essere uguale al numero di tutti quegli elementi nell'array `vars` che sono a loro volta uguali a `cardValue`.

Per esempio se la stringa `oper` rappresenta il simbolo "$<$" ciò significa che la variabile Card(`vars`, `cardValue`) deve essere strettamente minore di `value`.

Definizione A.1. Sia V la famiglia indiciata da I_V delle variabili vincolate appartenenti all'array `vars`, $n \in \mathbb{N}$ la dimensione dell'array `vars`, $a \in \mathbb{Z}$ il valore corrispondente alla variabile intera `cardValue`, si definisce la famiglia S tale che:

$$\forall i \in I_V, \quad \forall v_i \in V, \quad v_i \in S \Leftrightarrow v_i = a.$$

Si noti che $I_V = \{0, 1, \ldots, n-1\}$ e che la cardinalità di I_V è esattamente n. In questo modo si può mappare ogni elemento dell'array `vars` con un elemento della famiglia indiciata V mediante i rispettivi indici, che coincidono.

Definizione A.2. Sia S la famiglia definita in A.1 ed indiciata da I_S, $\odot$ l'operatore descritto dalla stringa `oper` e $b \in \mathbb{Z}$ il valore corrispondente alla variabile intera `value`, si definisce il *vincolo di cardinalità* $\mathcal{C}$ tale che:

$$\mathcal{C} \leftarrow |I_S| \odot b.$$

Se n è la lunghezza del vettore `vars`, allora il vincolo $\mathcal{C}$ appena definito si può anche esplicitare nel seguente modo:

$$\mathcal{C} \leftarrow (v_0 = a \Leftrightarrow v_0 \in S) \wedge \cdots \wedge (v_{n-1} = a \Leftrightarrow v_{n-1} \in S) \wedge |I_S| \odot b. \quad \text{(A.1)}$$

A.2 Una prima implementazione del vincolo di cardinalità

Possiamo utilizzare nell'implementazione il concetto di insieme di indici, sfruttando la classe `VarSet` che modella proprio insiemi di interi, restringendone il dominio ai soli valori non negativi.

Se n è la lunghezza del vettore `vars`, allora il vincolo $\mathcal{C}$ può quindi essere riscritto come segue:

$$\mathcal{C} \leftarrow (v_0 = a \Leftrightarrow 0 \in I_S) \wedge \cdots \wedge (v_{n-1} = a \Leftrightarrow n-1 \in I) \wedge |I_S| \odot b. \quad \text{(A.2)}$$

Vediamo la costruzione del vincolo $\mathcal{C}$ nell'implementazione JSR-331 basata su JSetL. Inizialmente costruiamo l'insieme degli indici di dominio $[0, n-1]$ ed il vincolo vuoto.

Listato A.1: postCardinality()

```
// S rappresenta l'insieme degli indici.
SetLVar S = new SetLVar("_Card" + counterSet++, new
    MultiInterval(), new MultiInterval(0, vars.length-1));
// Il vincolo C.
JSetL.Constraint cardinality = new JSetL.Constraint();
```

Passiamo quindi ad inizializzare `cardinality` con il vincolo sulla cardinalità dell'insieme degli indici, la funzione `getOperator(oper)` restituisce un codice relativo all'operatore opportuno e mediante lo statement `switch` viene richiamata la relativa funzione ed aggiornato il vincolo $\mathcal{C}$.

Listato A.2: postCardinality()

```
switch(getOperator(oper)) {
  case 1: {
    // Case = "equals".
    cardinality.and(S.card().eq(value));
    } break;
  case 2: {
```

```
    // Case != "not equals".
    cardinality.and(S.card().neq(value));
    }
  case 3: {
    // Case < "less".
    cardinality.and(S.card().lt(value));
    }
  case 4: {
    // Case <= "less equals".
    cardinality.and(S.card().le(value));
    }
  case 5: {
    // Case > "greater".
    cardinality.and(S.card().gt(value));
    }
  case 6: {
    // Case >= "greater equals".
    cardinality.and(S.card().ge(value));
    }
  default: throw new UnsupportedOperationException();
}
```

A questo punto per ogni variabile del vettore dato `vars` viene aggiornato il vincolo con la congiunzione delle due implicazioni $v_i = a \Rightarrow i \in I_S$ e $i \in I_S \Rightarrow v_i = a$.

Listato A.3: postCardinality()

```
for (int i = 0; i < vars.length; i++) {
  SetLVar X = new SetLVar("_X"+i, new MultiInterval(i,i));
  IntLVar tmp = (IntLVar) vars[i].getImpl();
  // C <- C e v_i = a --> i in S.
  cardinality.and(tmp.eq(cardValue).impliesTest(X.subset(S))
      );
  // C <- C e i in S --> v_i = a.
  cardinality.and(X.subset(S).impliesTest(tmp.eq(cardValue))
      );
  }
```

Al termine del ciclo `cardinality` rappresenta il vincolo $\mathcal{C}$ definito da (A.2).

Esempio di utilizzo

Vediamo un semplice esempio di come si può utilizzare il vincolo di cardinalità. Costruiamo cinque variabili intere v_0, v_1, v_2, v_3 e v_4 di dominio $[0, 4]$ e costruiamo un vincolo di cardinalità tale che il numero delle variabili v_i che assumono il valore 2 sia esattamente 3.

Ecco la definizione del problema:

Listato A.4: esempio di utilizzo di postCardinality

```
int n = 5;
// Dichiarazione e inizializzazione delle 5 variabili.
Var[] vars = new Var[n];
for (int i = 0; i < vars.length; i++)
  vars[i] = p.variable("v" + i, 0, n - 1);

// Aggiunta nel problema p del vincolo di cardinalita'
// sulle variabili.
p.postCardinality(vars, 2, "=", 3);
```

Chiamando in seguito la funzione `findSolution()` ecco l'output di questo semplice esempio:

```
JSetL - 2.2
Solution #0:
 v0[0] v1[0] v2[2] v3[2] v4[2]
```

A.3 Una soluzione più efficiente

La soluzione sopra descritta, progettata ed applicata in prima istanza durante la fase d'implementazione, sebbene abbastanza elegante e dichiarativa, si è poi rivelata inefficiente nella fase di testing. Probabilmente i problemi di efficienza sono dovuti all'utilizzo eccessivo delle variabili insiemistiche per chiamate multiple ai metodi `postCardinality` o l'istanziazione di più classi `Cardinality` e `GlobalCardinality`. Si è quindi deciso di adottare un sistema alternativo che sfrutta un nuovo vincolo definito da utente con il solver JSetL: `Occurrence`.

A.3.1 Il vincolo `Occurrence`

Il vincolo `Occurrence` è stato creato come vincolo JSetL definito da utente, aggiunto quindi ad un package della libreria per supportare lo standard JSR-331. Come ogni vincolo utente di JSetL viene definito nel seguente modo:

```
public class Occurence extends NewConstraintsClass {
```

La funzione fornita dal vincolo, chiamata `occurrence`, definisce la semantica dello stesso: data una lista di variabili logiche intere `l`, due variabili `v` e `n`, vincola la lista `l` ad avere esattamente `n` elementi che assumono il valore `v`.

All'interno dell'implementazione della classe `Cardinality` il questo metodo viene utilizzato con l'ausilio di una nuova variabile intera (libera) nel parametro `n`. Di fatto questa variabile rappresenta la cardinalità dell'insieme I_S definito in precedenza. Questa variabile viene poi vincolata al valore richiesto mediante l'operatore definito.

Listato A.5: `Occurrence` usato in `cardinality`

```
SolverClass solver = ((Solver) p.getSolver()).
    getSolverClass();
Occurrence listOps = new Occurrence(solver);
Vector<IntLVar> varsList = new Vector<IntLVar>();
for (int i = 0; i < vars.length; i++)
        varsList.add((IntLVar) vars[i].getImpl());
cardinality.and(listOps.occurrence(varsList, v, k));
switch(p.getOperator(oper)) {
case 1: {
        // Case = "equals".
        cardinality.and(k.eq(value));
.
.
.
```

Inizialmente viene caricato il solver per poter inizializzare il vincolo utente JSetL. Quindi gli oggetti implementativi (`IntLVar`) degli elementi dell'array `vars` vengono inseriti in un `Vector`. A questo punto si utilizza `occurrence` per vincolare `k` elementi di `vars` ad essere uguali a `v`.

Infine, mediante il costrutto `switch`, a seconda della relazione richiesta si vincola `k` per ottenere il risultato voluto:

$$|I_S| \odot b.$$

Appendice B

Test e Valutazioni

È noto che la parte di testing e validazione del codice sia di fondamentale importanza per quanto riguarda lo sviluppo del software. La specifica JSR-331, come descritto nel capitolo 2, definisce un pacchetto di test per la validazione, ovvero per verificare la conformità con le specifiche. Sebbene i contenuti di questo pacchetto rappresentano ciò che è normativo, non sono un valido sistema per la verifica del codice prodotto nell'implementazione sottostante. A dimostrazione di questo si evidenzia il fatto che, mediante il suddetto, il *coverage* dell'implementazione basata su JSetL prodotta risulta del 17% circa.

B.1 Validazione e coverage

Per la validazione del codice sono stati quindi scritti alcuni test ad *hoc* (circa sessanta) mediante un approccio *white box* per cercare di coprire tutti i casi individuati. Per quanto concerne le variabili logiche intere si è riusciti a coprire praticamente il 100% del codice, grazie anche al fatto che la specifica JSR-331 risulta effettivamente matura. Purtroppo non si può dire lo stesso per la trattazione delle variabili booleane, insiemistiche e reali. Pertanto i test prodotti coprono circa l'87% del codice totale, comunque un risultato molto migliore rispetto ai test normativi. Occorre comunque sottolineare che i test del TCK (`org.jcp.jsr331.tests`) non sono pensati per la totale verifica dell'implementazione o per il coverage.

B.2 Valutazioni

All'interno del TCK è presente anche un altro pacchetto denominato *sample* (`org.jcp.jsr331.samples`) che fornisce alcuni test che consentono di visualizzare il tempo e la memoria impiegati per la risoluzione dei problemi dati. Questo è stato quindi utilizzato per una serie di confronti

tra i solver che attualmente supportano la specifica JSR-331: JaCoP, Constrainer, Choco e ovviamente JSetL.

Si riportano quindi i risultati ottenuti, la macchina sulla quale questi test sono stati eseguiti utilizza il sistema operativo Fedora 15 a 64-bit, processore Intel® Core™2 Duo CPU P7450 @ 2.13GHz × 2 e 3,8 GB di ram.

I *sample* utili per la comparazione sono in totale dieci, per semplificare la valutazione e per chiarezza sono stati suddivisi in due tipologie: semplici e complessi.

B.2.1 Test semplici

I test che si possono inserire in questa categoria presentano tutti un numero limitato di variabili (5–30) e di vincoli, anche se a volte utilizzano vincoli globali, notoriamente più complessi.

Sono stati catalogati semplici poiché terminano con tutti i solver provati in tempi rapidi e con un basso o moderato utilizzo di memoria.

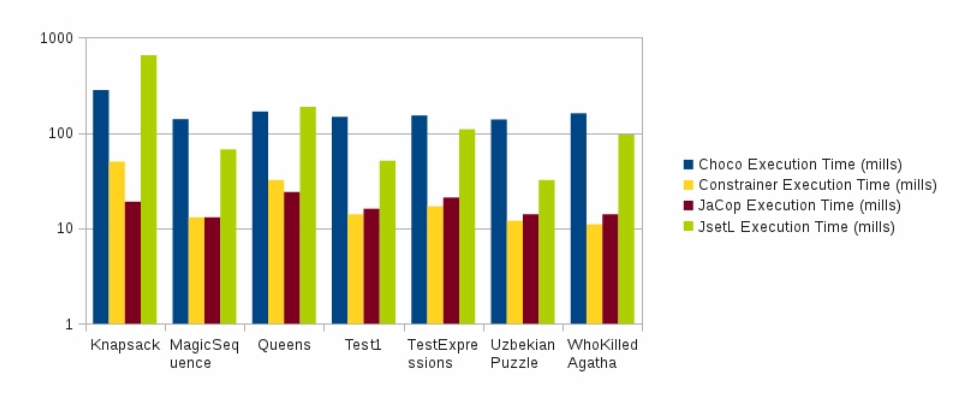

Figura B.1: test semplici, tempo impiegato.

Come si evince in figura, in questi semplici test JSetL si comporta abbastanza bene, meglio di Choco in ben cinque casi su sette e praticamente un pareggio sul problema delle otto regine. Si potrebbe dire che con poche variabili l'ordine di grandezza del tempo impiegato coincide con Choco. Il discorso è differente se si confronta con Constrainer o JaCoP, decisamente più performanti.

Per quanto riguarda la memoria utilizzata invece, JSetL riesce ad essere sempre sotto ai livelli di Choco e ben tre volte meglio di tutti gli altri. I casi in cui viene utilizzata tanta memoria sono *Knapsack* e *Queen*. Il primo probabilmente è causato dalla ricerca ottima (`findOptimalSolution`), che non è direttamente supportata dal solver JSetL, ma viene implementata mediante un algoritmo fatto ad *hoc* che espande e propaga un vincolo all'interno del solver. Mentre la pesantezza del problema delle regine, in

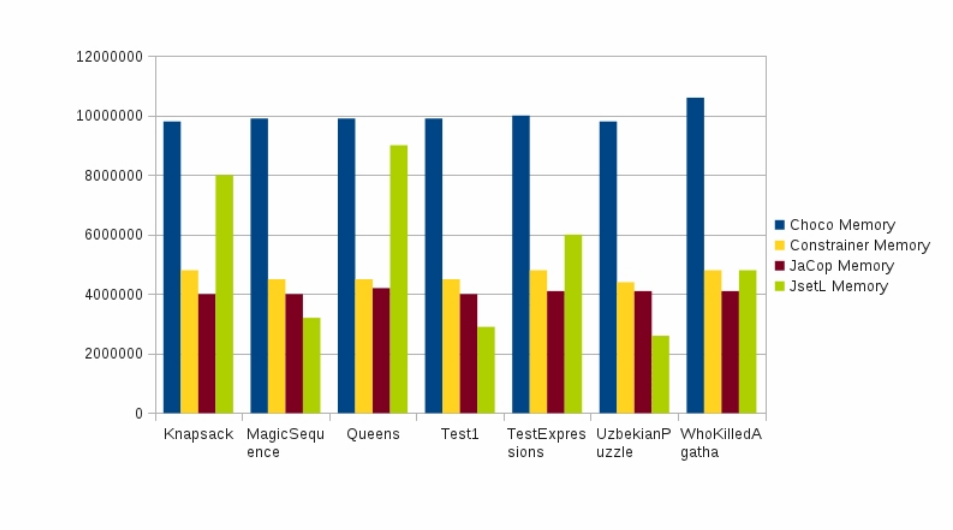

Figura B.2: test semplici, memoria utilizzata.

termini di memoria, è indubbiamente dovuto alla presenza del vincolo AllDifferent su tante variabili, e come questo è implementato in JSetL.

B.2.2 Test complessi

I test più complessi sono caratterizzati da un elevato numero di variabili legate al problema o di supporto e da un elevato numero di vincoli, su tutti quelli globali: Cardinality, AllDifferent, etc.

Sono stati catalogati come complessi poiché anche i solver che hanno dato prova di essere i più performanti (JaCoP e Constrainer) risolvono i problemi dati con tempi ben al di sopra dei precedenti.

Dal grafico si può notare che JSetL in questi casi è ben al di sopra degli altri solver come tempo d'esecuzione. Nel test *Golomb Rulers* il tempo impiegato è circa il doppio rispetto a Choco e il triplo rispetto JaCoP e Constrainer. In *Magic Square* è ben dieci volte più lento degli altri, mentre in *Graph Coloring* circa sessanta. In questo caso però occorre dire che JaCoP sembra non terminare, infatti la sua barra non è presente nel grafico.

Per quanto riguarda la memoria utilizzata nei test complessi è invece evidente che JSetL ne utilizzi veramente troppa rispetto agli altri solver e molto probabilmente anche il deficit di prestazioni è dovuto a questo.

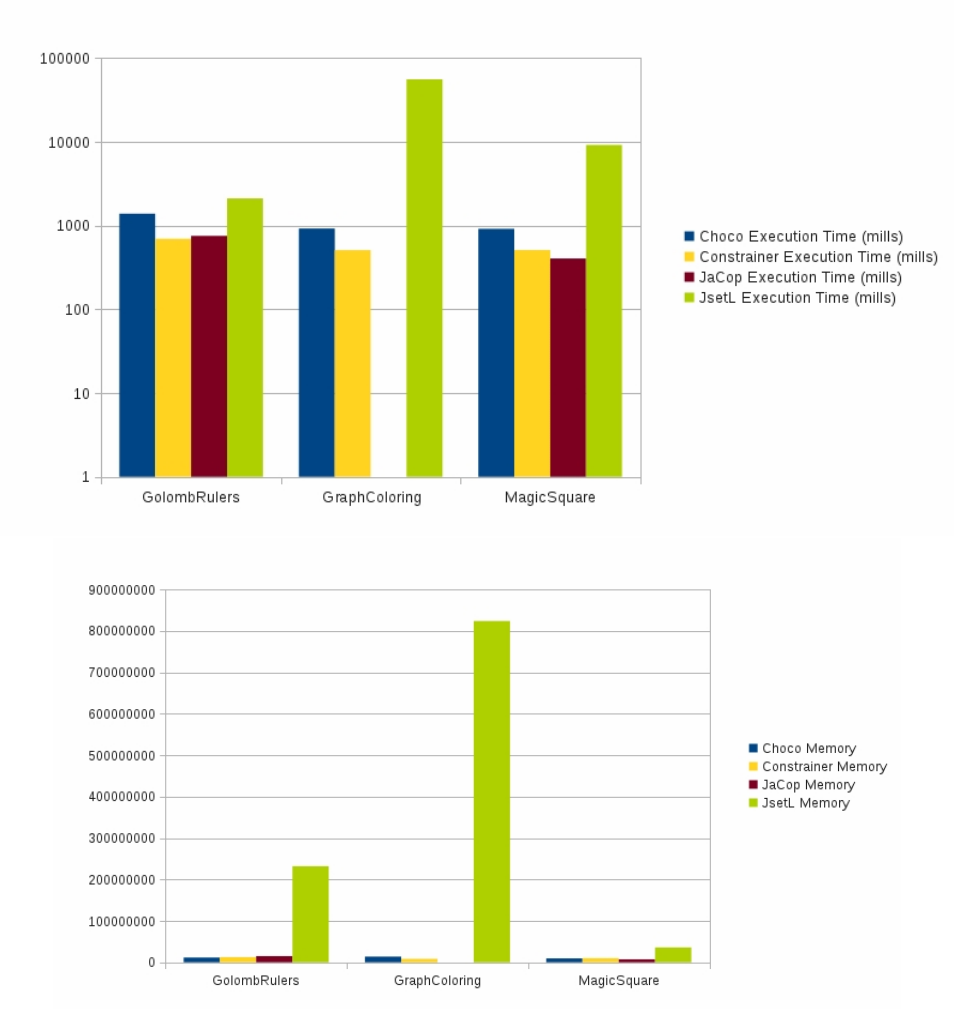

Figura B.3: test complessi.

Bibliografia

[1] Jacob Feldman
JSR-331 Java Constraint Programming API SPECIFICATION
`http://openrules.com/downloads/jsr331/JSR331.Specification.v081.pdf`

[2] Roberto Amadini
Studio e realizzazione in Java di domini e regole per la risoluzione di vincoli su interi e insiemi di interi
`http://www.cs.unipr.it/Informatica/Tesi/Roberto_Amadini_20111109.pdf`

[3] Luca Console, Evelina Lamma, Paola Mello, Michela Milano
Programmazione Logica e Prolog
UTET Libreria srl, 1997.

[4] Stuart Russell, Peter Norvig
Intelligenza Artificiale
Un approccio moderno - Volume 1
Pearson Education Italia S.r.l., 2005.

[5] A. Dovier, C. Piazza, E. Pontelli, G. Rossi
Sets and constraint logic programming
ACM TOPLAS 2000; 22(5):861-931.

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

[7] Gianfranco Rossi, Roberto Amadini
JSetL User's Manual Version 2.3
Quaderni del Dipartimento di Matematica, n. 507, Università di Parma, 24 Gennaio 2012.

[8] JaCoP - Java Constraint Programming solver
`http://jacop.osolpro.com/`

[9] Java Platform, Standard Edition 6: API Specification
`http://java.sun.com/javase/6/docs/api/`

[10] JSetL Home Page
`http://cmt.math.unipr.it/jsetl.html`

[11] Constraint Programming Standardization Blog
`http://cpstandard.wordpress.com/`