Departament de Llenguatges i Sistemes Informatics
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Master in Computing

Master of Science Thesis

EINES BASADES EN LA LÒGICA PER A MODELATGE I RESOLUCIÓ DE PROBLEMES COMBINATORISMASTER'S THESIS TITLE

Miquel Palahí Sitges

Advisor/s: Dr. Miquel Bofill Arasa
Dr. Mateu Villaret Auselle

07 September 2010

Resum

Durant els últims anys dins la informàtica hi ha hagut importants avenços en les tècniques i eines basades en la lògica. Especialment en el camp de la satisfactibilitat de la lògica proposicional (SAT), fins al punt que els resoledors SAT actuals poden arribar a tractar instàncies de problemes amb milions de clàusules i amb centenars de milers de variables. Les tècniques de SAT també han evolucionat adaptant-se a lògiques més expressives. Per exemple, en la Satisfactibilitat Mòdul Teories (SMT), s'intenta decidir la satisfactibilitat d'una fórmula respecte una teoria de fons (per exemple, aritmètica lineal d'enters o reals) en lògica de primer ordre amb igualtat (normalment sense quantificadors).

A més del guany expressiu que això suposa, les fórmules generades seran molt menors. Però el més important és que aquest guany expressiu no s'obté en detriment de l'eficiència, atès que en SMT s'usen les mateixes tècniques que a SAT (aprenentatge, backtracking no cronològic, etc) sobre fragments decidibles de la lògica de primer ordre. Per tant, SMT ofereix un bon compromís entre eficiència i expressivitat.

Independentment del fet que SMT hagi anat lligat des dels seus orígens a l'àrea de la verificació de hardware i software, aquesta tecnologia s'ha començat a aplicar satisfactòriament a altres tipus de problemes, com per exemple problemes de planificació i scheduling.

Amb aquest esperit, en aquesta tesi de màster presentem el sistema `Simply`: que resol els problemes de satisfacció de restriccions (*Constraint Satisfaction Problem*, CSP) primer traduint les instàncies dels problemes a fórmules SMT, per a després buscar si té solució amb un resoledor SMT. Si es troba una solució a la fórmula, aquesta es tradueix a la corresponent solució de la instància original. Les instàncies dels CSP estan expressades mitjançant el llenguatge declaratiu també anomenat `Simply`, que també definim en aquest projecte. El sistema també permet resoldre problemes d'optimització.

En aquesta memòria començarem explicant el *background* necessari (problemes combinatoris, programació amb restriccions, SMT...), després presentarem el sistema `Simply` (llenguatge, generació de codi, optimització,...), i finalment mostrarem i comentarem els resultats experimentals obtinguts.

Agraïments

Primer de tot m'agradaria agrair als meus tutors Mateu Villaret i Miquel Bofill la gran oportunitat que m'han donat, i sobretot la paciència que han tingut. A en Josep Suy per l'ajuda i consells que m'ha donat. A la Núria i a la meva familia per el suport i ànims que m'han donat durant aquest temps. Als companys del Màster que tantes hores hem passat junts a la universitat. I a tota la gent que durant aquest temps s'ha interessat pel projecte. Moltes gràcies a tots!

Índex

Capítol 1

Introducció

En el dia a dia ens trobem amb problemes que, tot i poder-se plantejar fàcilment, el nombre de combinacions de valors de les variables que poden donar lloc a solucions creix exponencialment amb la mida del problema, com per exemple la confecció d'horaris escolars, el fixar un calendari de partits, la planificació de torns de treballadors, la planificació d'una seqüència de tasques a realitzar, etc. De vegades les solucions d'aquests problemes tenen associades una funció de cost que volem maximitzar o minimitzar, per exemple els temps de producció. Per aquest motiu des de l'àmbit informàtic s'ha anat estudiant com resoldre de manera eficient aquest tipus de problemes: fent programes ad hoc, utilitzant tècniques de programació entera, utilitzant tècniques de programació amb restriccions, mitjançant eines lògiques, etc. Nosaltres ens centrarem en la programació amb restriccions mitjançant eines basades en la lògica.

A l'última dècada hi ha hagut avenços importants en les tècniques i eines basades en la lògica. Aquests avenços han estat especialment importants al camp de la satisfactibilitat proposicional (SAT), fins a tal punt que els resoledors SAT moderns poden arribar a tractar problemes reals amb instàncies de centenars de milers de variables i milions de clàusules. Per tant, els resoledors SAT han esdevingut una bona eina per a resoldre problemes combinatoris discrets; la idea és codificar els problemes combinatoris com a fórmules booleanes, el model de les quals (si existeix) ens permeti donar una solució al problema combinatori original. A [CMP06] per exemple s'hi descriu una aplicació que tradueix especificacions escrites amb un llenguatge de modelatge declaratiu a instàncies de SAT. A [Kau06, ZLS04] podem veure diverses aplicacions que empren la tecnologia SAT per a resoldre problemes de planificació i per a la programació de tornejos respectivament. També s'han fet comparatives interessants entre SAT i problemes de satisfacció de restriccions (*Constraint Satisfaction Problem* (CSP)) com ara [Hua08]. Diverses codificacions i tècniques es poden trobar a [Wal00, GJM06].

Les exitoses tècniques per a SAT han estat adaptades darrerament a lògiques més expressives. Per exemple, el problema de Satisfactibilitat Mòdul Teories (*Satisfiability Modulo Theories (SMT)*) consisteix en determinar la satisfactibilitat de fórmules booleanes en lògica de primer ordre amb igualtat respecte una teoria de fons, com podria ser l'aritmètica lineal (ja sigui d'enters o reals), la teoria dels arrays, etc. o combinacions d'aquestes teories [Seb07, NOT06, GHN$^+$04, BDS02, ACG00]. Les fórmules són sovint sintàcticament restringides (com per exemple ser lliures de quantificadors) de manera que el problema de la satisfactibilitat continuï essent decidible. Per tant, una fórmula SMT és una generalització d'una fórmula booleana SAT on algunes de les variables proposicionals s'han substituït per àtoms construïts amb els predicats de les teories. Així els problemes poden contenir fórmules de l'estil:

$$f(f(x) - f(y)) \neq f(z) \wedge x + z \leq y \wedge y \leq x \Rightarrow z < 0$$

Passem doncs a disposar d'un llenguatge de modelatge més ric que en les fórmules proposicionals. Fixem-nos que a la fórmula de l'exemple, apareixen funcions (f, $+$ i $-$) i predicats ($\leq$, $<$ i $\neq$) sobre enters, i que les variables x, y i z són de tipus enter, mentre que en proposicional les variables només poden ser booleanes.

Adaptacions recents de tècniques SAT a l'entorn de SMT es descriuen a [SS06]. La principal àrea d'aplicació de SMT és la verificació de software i hardware. Tanmateix, les teories disponibles no restringeixen l'ús de SMT per a problemes de verificació. De fet, aquestes teories permeten codificar d'una manera natural problemes molt diversos fora de l'àrea de verificació (veure [NO06] per a una aplicació on s'empra un resoledor SMT sobre el problema d'optimització d'assignació de freqüències d'enllaços radiofònics plantejat al 1993 pel CELAR (Centre d'ARmes ELectròniques) Francès, essent competitiu amb els actuals resoledors de CSP amb les seves millors heurístiques sobre el problema). Els principals desafiaments de SMT per a programació amb restriccions (*Constraint Programming* (CP)) i optimització han estat plantejats a [NORCR07]. Típicament la resolució amb CP consisteix en determinar si existeix una assignació de valors per a les diferents variables d'acord al seus dominis (finits), que compleixin certes restriccions.

D'altra banda, des dels seus inicis el "sant graal" de CP ha estat obtenir un llenguatge declaratiu que permeti als usuaris especificar el problema i oblidar-se de les tècniques a emprar per a resoldre'l. Hi ha diversos sistemes que han obtingut bons resultats en aquesta direcció, dels quals només en comentarem dos dels més rellevants: MiniZinc [NSB$^+$07][1] va sorgir com una proposta de "llenguatge estàndard per al modelatge de CSP" que pot traduir-se a un codi intermig anomenat FlatZinc, el qual pot ésser resolt per diversos resoledors especialitzats (veure Subsecció 2.2.1); ESSENCE [FHJ$^+$08] que permet a l'usuari especificar problemes combinatoris amb una barreja de llenguatge natural i de matemàtica discreta.

Donades les possibilitats aparents de SMT per a resoldre CSPs, i com que les instàncies en SMT de problemes reals (tot i que més curtes que en SAT) poden arribar a ésser molt extenses, apareix la necessitat d'automatitzar d'alguna manera aquest procés de traducció de CSP a SMT. Així doncs, es va plantejar crear un nou llenguatge de CP per a modelar problemes d'una manera senzilla per al programador, que hem anomenat `Simply`[2]. De fet, `Simply` és també el nom del sistema que compilarà instàncies de CSPs a fórmules en format SMT-LIB per a ser resoltes amb un resoledor SMT. El sistema per tant, ens permetrà comparar la resolució d'aquests problemes mitjançant SMT respecte els resoledors de CSP actuals. En particular, serà interessant comprovar la robustesa de la política de "caixa negra" dels resoledors SMT, és a dir, el fet que la gestió de l'estratègia de cerca de solucions sigui totalment transparent per a l'usuari.

Encara que actualment la riquesa del llenguatge no és equiparable a la del MiniZinc o l'ESSENCE, la seva simplicitat el fa realment pràctic. El llenguatge `Simply` és similar al de l'EaCL [MTW$^+$99] i al de MiniZinc. Les característiques més destacades del llenguatge són: els arrays, les sentències `Forall`, llistes per comprensió, alguns constraints globals i la definició de funcions i restriccions de l'usuari. A la figura 1.1 es mostra una modelització naïf amb `Simply` per resoldre el problema de les n-*reines*: posar n reines al tauler d'escacs de manera que no es matin entre elles. A la figura 1.2 podem veure una possible solució per al problema quan $n = 8$.

`Simply` treballa amb l'esperit de SPEC2SAT [CS05], que transforma especificacions escrites en NP-SPEC [CIP$^+$00] a instàncies SAT, així com de *Universal Booleanization of Constraint Models* [Hua08], on es tradueixen instàncies FlatZinc (codi intermig de MiniZinc) a SAT. Tanmateix,

[1] MiniZinc és un subconjunt de Zinc dissenyat per ser més pròxim als resoledors de CSPs.

[2] El nom ve de que intenta ser un sistema simple amb un llenguatge fàcil i que estalviï feina al programador ja que no s'ha de preocupar de gestionar la cerca, etc.

```
Problem:queens.sly
    Data
        int n=8;
    Domains
        Dom rows=[1..n];
    Variables
        IntVar q[n]::rows;
    Constraints
        AllDifferent([q[i]|i in [1..n]]);
        Forall(i in [1..n-1]) {
            Forall(j in [i+1..n]) {
                q[i]-q[j]<>j-i;
                q[j]-q[i]<>j-i;
            }
        }
```

Figura 1.1: Modelització naive per resoldre el problema de les n-*reines*

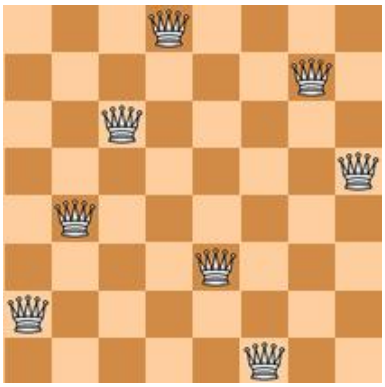


Figura 1.2: Exemple de solució del problema de les 8 reines.

tal com s'ha dit, el llenguatge d'entrada del `Simply` és similar al de l'EaCL i, el més rellevant, genera instàncies SMT d'acord amb l'estàndard del llenguatge SMT-LIB [RT06], en lloc d'instàncies SAT. Actualment, `Simply` genera fórmules lliures de quantificadors sobre la teoria d'aritmètica lineal entera, tot i que està previst d'estendre el compilador per poder treballar amb altres teories interessants com *difference logic* o *bit-vectors*. Un cop traduït el problema ja pot ésser solucionat per qualsevol dels resoledors SMT que suportin les teories necessàries. Gràcies a la riquesa expressiva de SMT, les instàncies SMT són més petites que les equivalents aplanades a SAT i, com es pot veure a la secció 4.1 i a la secció 4.3, molts problemes estàndards de restriccions poden ésser resolts (en general) amb un temps raonable pels resoledors SMT actuals. El sistema `Simply` també permet resoldre la versió d'optimització del CSP, amb uns temps també competitius com es pot veure a la secció 4.2.

Finalment, `Simply` pot servir com a generador de *benchmarks* per a comparar els diferents resoledors SMT.

Capítol 2

Estat de l'art

Abans de descriure `Simply` primer introduirem diferents aspectes importants per a situar el projecte: començarem parlant sobre el tipus de problemes que es volen resoldre i les tècniques clàssiques per a resoldre'ls com a CSPs, per tot seguit centrar-nos amb el problema SAT i la seva resolució i finalment presentar SMT.

2.1 Problemes Combinatoris

Com ja hem dit, en el tipus de problemes que considerarem, el nombre de combinacions de valors de les variables que poden donar lloc a solucions creix exponencialment amb la mida del problema. Típicament la resolució d'aquests problemes consisteix en saber si existeix una assignació de valors per a les diferents variables d'acord al seus dominis (finits), que compleixi certes restriccions.

La manera més fàcil de resoldre un problema d'aquest tipus és l'algorisme de *Generate and Test*. Podem veure'n l'esquema bàsic a l'Algorisme 1, on la crida inicial és `generate_and_test` $(X, \{\}, C)$.

Algorithm 1 Generate And Test

Require: X : Variables, A : Assignació, C : Restriccions

1: **if** $X = \{\}$ **then**
2: **return** `satisfa`(A, C);
3: **end if**
4: x := `tria_una_variable`(X);
5: **for all** v in domini(x) **do**
6: R := `generate_and_test`$(X - \{x\}, A + +(x, v), C)$;
7: **if** R != **false then**
8: **return** R;
9: **end if**
10: **end for**
11: **return false**;

Com a paràmetres de la funció `generate_and_test` tenim un conjunt de variables pendents d'assignar X, un conjunt d'assignacions de valors a variables A (presentades com a parells variable-valor: (x, v)) i un conjunt de restriccions C. Si no tenim variables pendents d'assignar retornem si es satisfà el conjunt de restriccions segons el conjunt d'assignacions. Altrament vol dir que hi ha variables pendents d'assignar un valor, n'agafem una (x := `tria_una_variable`(X)), i per tot

Mida de n						
Complexitat	10	20	30	40	50	60
n	0.00001 segons	0.00002 segons	0.00003 segons	0.00004 segons	0.00005 segons	0.00006 segons
n^2	0.0001 segons	0.0004 segons	0.0009 segons	0.0016 segons	0.0025 segons	0.0036 segons
n^3	0.001 segons	0.008 segons	0.027 segons	0.064 segons	0.125 segons	0.216 segons
n^5	0.1 segons	3.2 segons	24.3 segons	1.7 minuts	5.2 minuts	13.0 minuts
2^n	0.001 segons	1.0 segons	17.9 minuts	12.7 dies	35.7 anys	366 segles
3^n	0.059 segons	58 minuts	6.5 anys	3855 segles	2×10^8 segles	1.3×10^{13} segles

Figura 2.1: Temps necessari per a resoldre una instància segons la complexitat del problema en funció de la mida de l'entrada n

valor dins del domini de x cridem recursivament `generate_and_test` amb el conjunt de variables pendents d'assignar sense la variable actual $X - \{x\}$, el conjunt d'assignacions de valors a variables més el nou valor donat a $A + +(x, v)$ i el mateix conjunt de restriccions. Fins que per algun valor tenim èxit.

D'aquesta manera en el pitjor dels casos si tinguéssim n variables i m valors a prendre per variable, tindríem m^n combinacions a mirar.

2.1.1 Complexitat

Tot i que existeixen tècniques que en determinats casos ens permeten acotar enormement l'espai de cerca (veure Subsecció 2.2.2), la complexitat intrínseca del problema segueix essent la mateixa i en la majoria de problemes combinatoris el cas pitjor l'espai de cerca segueix essent exponencial. Per veure el que això implica podem veure la taula de la Figura 2.1 (extreta de [GJ79]) on veurem el temps necessari per a resoldre una instància segons la complexitat del problema en funció de la mida de l'entrada, n. A mesura que anem augmentant la mida el temps augmenta lleugerament passant de tardar centèsimes de mil·lèssima de segon a tardar minuts en els casos de n, n^2, n^3 i n^5. On es veu un augment important en el temps és en els casos exponencials 2^n i 3^n, on de seguida necessitem anys o segles per a resoldre una instància del problema. Si tenim en compte que en l'exemple estàvem parlant que teníem m^n podem veure que el temps necessari creix moltíssim. Un cas particular seria SAT, quan $m = 2$ (cert/fals), on el temps augmenta exponencialment.

En la taula de la Figura 2.2 (extreta de [GJ79]) podem veure el nombre N_i d'instàncies que es poden resoldre segons la seva complexitat en funció de la velocitat de l'ordinador: un d'actual, un de 100 cops més ràpid que un d'actual i un de 1000 cops més ràpid. En els primers nivells de complexitat ($n \ldots n^5$), considerant que amb un ordinador actual podríem resoldre N_i instàncies, veiem que a l'augmentar la velocitat de l'ordinador podem arribar a resoldre $v \times N_i$ instàncies, que és un augment acceptable respecte l'increment de velocitat de l'ordinador. Mentre que en el cas dels exponencials (2^n i 3^n) podríem resoldre $v + N_i$ instàncies, que és un augment insignificant comparat amb l'increment

Complexitat	Pc actual	PC 100 cops més ràpid	PC 1000 cops més ràpid
n	N_1	100 N_1	1000 N_1
n^2	N_2	10 N_2	31.6 N_2
n^3	N_3	4.64 N_3	10 N_3
n^5	N_4	2.5 N_4	3.98 N_4
2^n	N_5	N_5+6.64	N_5+9.97
3^n	N_6	N_6+4.19	N_6+6.29

Figura 2.2: Nombre N_i d'instàncies que es poden resoldre segons la complexitat del problema en funció de la velocitat de l'ordinador

de velocitat de l'ordinador. Per exemple, resolent problemes de complexitat 2^n, amb un ordinador 1000 cops més ràpid només podríem resoldre unes 10 instàncies més que amb un ordinador actual.

Així doncs mirarem de traçar una línia separadora entre els problemes que direm que són "tractables" (fàcils de resoldre) i els que direm, informalment, que són "intractables" (degut a que es tracta de problemes molt durs de resoldre), en funció de la seva complexitat computacional. Considerem com a tractables aquells problemes pels quals existeixi un algorisme determinista que resol el problema amb un temps polinòmic respecte la mida de l'entrada, n. El conjunt de problemes que es poden classificar dins d'aquest grup són els anomenats problemes polinòmics (*Polynomial* **P**). Dins aquest tipus de problemes hi trobem els logarítmics ($log(n)$), lineals (n), quadràtics (n^2), cúbics (n^3)... En canvi, considerem com a intractables aquells problemes pels que no s'ha trobat un algorisme determinista que resol el problema en temps polinòmic.

El conjunt de problemes polinòmics no deterministes els anomenem **NP** (de *Non-deterministic Polynomial*). És fàcil veure doncs, que tot problema de P és trivialment un problema de la classe NP, ara bé, no se sap si tot problema de NP també és de P, és a dir si P=NP. Majoritàriament la comunitat assumeix que P≠NP.

Per estudiar la complexitat computacional dels problemes una eina fonamental és la *reducció*. Per poder reduir un problema a un altre ha d'existir una funció de transformació f que transformi les instàncies del problema Π a instàncies de Π', de manera que tota instància I de Π tindrà solució si i només si la instància $f(I)$ de Π' té solució. Típicament, i pel que fa al que tot seguit exposem, es requereix que f sigui polinòmica, és a dir, que existeixi un algorisme determinista de temps polinòmic que la calculi.

Els problemes que pertanyen a NP, que no se sap si pertanyen a P i als quals tot problema NP s'hi pot reduir, es coneixen com *NP-Complets*, i pertanyen a la categoria que n'hem dit intractables. En particular, quan a un problema NP n'hi reduïm un d'NP-Complet, el primer passa a ser també NP-Complet. Per tant només falta tenir un problema NP-Complet per a poder classificar els problemes. Això ho va fer Cook [Coo71] demostrant que el problema de SAT és NP-Complet.

2.2 Programació amb Restriccions

Un cop introduït de manera genèrica el tipus de problemes que considerarem, passem a veure com es poden atacar mitjançant la resolució de restriccions.

Per atacar aquests problemes ho podem fer de dues maneres: desenvolupar aplicacions específiques per a cada problema donat, o bé triar una eina genèrica de resolució de restriccions i modelar el nostre problema per a poder-lo resoldre mitjançant aquesta eina. La primera manera pot tenir l'avantatge que

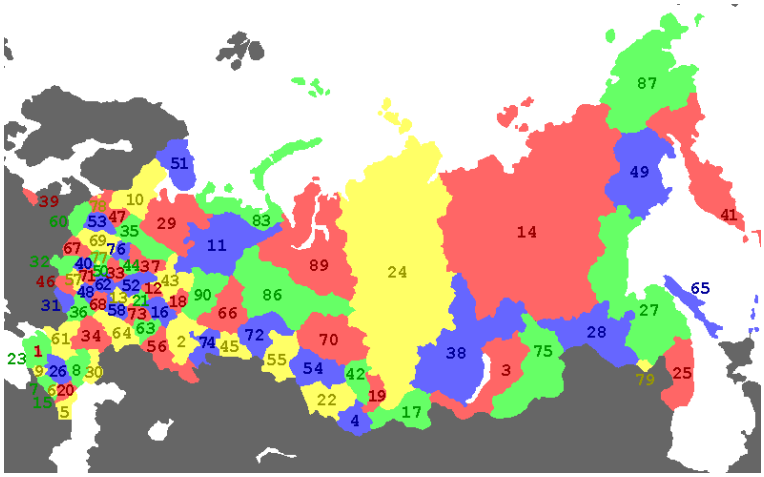


Figura 2.3: Exemple de mapa pintat amb quatre colors.

suposa conèixer el problema que s'està tractant a l'hora de desenvolupar els algorismes de resolució. No obstant, qualsevol canvi en l'especificació del problema pot implicar canvis importants en aquests algorismes. Cada cop més, doncs, s'imposa la segona manera. Es pot optar per utilitzar un llenguatge de programació específic de restriccions (MiniZinc, Essence, OPL), o bé per utilitzar llibreries sobre un altre tipus de llenguatge, com per exemple C++ amb la llibreria ILOG CP o llenguatges de programació lògica com SICSTUS, o Gprolog.

En qualsevol cas, el primer pas consistirà en modelar el problema per tal de poder-lo resoldre amb l'eina (o eines) que haguem triat.

2.2.1 Modelatge

Independentment de com es resolgui, un problema de satisfacció de restriccions es pot modelar mitjançant els següents tres elements:

1. *Variables*, que agafaran un valor d'un determinat domini (normalment finit).
2. *Dominis*, que definiran els valors que poden prendre cadascuna de les variables.
3. *Restriccions*, que estabiran les relacions que han de satisfer els valors que prenguin les variables del problema.

Per tant, modelar el problema consistirà en definir els tres elements: variables, dominis i restriccions, determinant **què** representarà cada variable, **quin** domini tindrà cada variable, i **com** hem de restringir les variables per tal que els valors que prenguin siguin solucions, és a dir, quines restriccions definiran el problema.

Exemple 1 *Considerem el problema de pintar un mapa amb n colors (podem veure'n un exemple a la figura 2.3) de manera que els països contigus, és a dir, que tinguin frontera entre ells, se'ls pinti un color diferent. Les variables serien els diferents països del mapa: Espanya, Andorra, etc. Els dominis, serien els diferents colors permesos per pintar el mapa:*

$$Espanya \in \{Vermell, Blau, Groc, \dots\}$$

$$Andorra \in \{Vermell, Blau, Groc, \dots\}$$

...

I les restriccions indicarien que el valor de les variables (color que pintem cada país) de països que són contigus ha de ser diferent:

$$Espanya \neq Andorra$$

...

Les restriccions entre variables es representaran d'una manera o d'una altra, segons el llenguatge/llibreria que es vulgui usar per resoldre el problema. Però en general, la programació amb restriccions proporciona moltes funcionalitats per expressar els problemes complexos d'una manera directa i declarativa proporcionant, per exemple:

- *Restriccions numèriques*. Solen ser restriccions que treballen generalment amb les variables i operacions aritmètiques, per exemple $2x + y = z$.

- *Restriccions simbòliques*. Solen ser restriccions que permeten treballar sobre funcions o estructures de dades, com per exemple $t[x] = y$ on les variables t, x i y representen respectivament una taula, un índex i un valor.

- *Meta-restriccions*. Solen ser restriccions booleanes que permeten a l'usuari representar problemes mitjançant disjuncions, implicacions, o fins i tot reificacions, en lloc de només tenir una conjunció de restriccions. Un exemple seria la reificació, que consistiria en afegir una variable booleana que agafarà el valor cert si la restricció ($x - y = z$) es compleix o fals en cas contrari: $(x - y = z) \leftrightarrow b$. Per exemple, la restricció que la distància entre dues variables ha de ser 5 ($x - y = 5 \vee y - x = 5$) es podria expressar com la conjunció de $\{(x - y = 5) \leftrightarrow b_1, (y - x = 5) \leftrightarrow b_2, b_1 \vee b_2.\}$

- *Restriccions globals*. Són restriccions que treballen sobre conjunts de variables. Un exemple típic és l'*AllDifferent*, que donada una llista de variables, $[x_1, \dots, x_n]$, restringeix tots els seus elements a ser diferents entre si, és a dir $\forall i \neq j \in \{1..n\}$. $x_i \neq x_j$. El principal avantatge d'aquest tipus de restriccions és que molts cops permeten aplicar algorismes dedicats per restringir i propagar valors dels dominis, com per exemple [Rég94].

Com ja hem comentat, és interessant disposar d'un llenguatge que proporcioni una sintaxi d'alt nivell per expressar restriccions. A continuació parlarem d'alguns dels llenguatges més emprats i sistemes semblants al `Simply` que hem desenvolupat.

IBM ILOG CP Optimizer

Segons la pàgina [ILO10], IBM ILOG CP Optimizer és la segona generació del motor de CP que ofereix IBM, que pot emprar-se o bé mitjançant C++, Java, C# o VB.Net, o bé dins l'eina de modelatge *IBM ILOG OPL-CPLEX Development Bundles*. Consta d'un optimitzador per resoldre problemes de *scheduling* i problemes combinatoris discrets. Les principals característiques són les següents:

- **Scheduling:**
  - Definició de tasques opcionals.

  - Definició de restriccions de precedència entre tasques, que poden incloure retards i aplicar-se a grups d'intervals.
  - Definició de propietats en els intervals: costos, preferències, ...
  - Definició de límits del nombre de tasques en paral·lel per un recurs.
  - Definició de restriccions en l'inici i fi de les tasques i solapaments.

- **Optimització de problemes combinatoris discrets:**
  - Definició de restriccions aritmètiques lineals i no lineals.
  - Definició de restriccions lògiques.
  - Definició de restriccions i expressions especialitzades: restricció *all-diferent*, restricció *pack*, restricció *lexicogràfica*, expressió *count*, expressió *desviació estàndard*, etc.
  - Definició de restriccions de compatibilitat/incompatibilitat: definició de les possibles assignacions per taules de variables de decisió. Per exemple, per modelar les transicions permeses en un problema de seqüenciació.

Prolog

El Prolog és un llenguatge de programació desenvolupat inicialment a la Universitat de Marsella [Rou75], com un eina pràctica per a programació lògica [Kow74]. Hi ha diferents entorns per a programar amb Prolog, i sobre alguns d'aquests s'han desenvolupat llibreries per programar amb restriccions, donant lloc al que és coneix com *Constraint Logic Programming* (CLP). En particular CLP és pràctic per a la resolució de problemes d'optimització discreta i verificació, per exemple scheduling, planificació, packing, etc. Alguns dels prologs amb restriccions més usats són: **Sicstus** [Cea10], **ECL^{i}PS^e** [Ecl10], **Gprolog** [DC01] o **SWI-prolog** [Wie03].

La resolució de restriccions es realitza típicament mitjançant llibreries dedicades (en Sicstus *clpfd* (CLP Finite Domains), en ECL^{i}PS^e*fd*). Els resoledors estan inspirats en l'esquema de CLP introduït per [JM87]. Sicstus treballa internament amb dues classes de restriccions: primitives i globals. De fet, les restriccions es tradueixen automàticament a una conjunció de restriccions primitives i globals de la llibreria *clpfd*. El valor d'una restricció primitiva es por reflectir a una variable de valor 0/1 (cert/fals, reificació). Típicament es poden afegir noves restriccions primitives definides pel programador (en Sicstus anomenades indexicals). Semblantment es poden afegir noves restriccions globals, escrites en Prolog per mitjà d'una interfície. A mesura que el programa prolog es va executant i les restriccions es van imposant, es va garantint la possible consistència[1] de les variables fins a arribar al procés de cerca de solució. Aquest procés de cerca de solució `labeling`, és bàsicament un *backtracking* o un *branch and bound* en el cas d'optimització. Normalment aquest procés de cerca és configurable amb diferents estratègies en la tria de variables i/o de valors.

Comet

Comet [MH03], és un llenguatge de programació orientat a objectes que pot treballar amb restriccions per a la cerca de veïnatge[2] i amb declarativitat i control d'abstraccions.

Des del punt de vista del llenguatge, està inspirat en la programació amb restriccions. Té una arquitectura per capes amb components de modelatge i de cerca, i un ric vocabulari, que inclou restriccions numèriques i estructurals. Les abstraccions de control s'allunyen de les típiques dels llenguatges

[1] Veure secció 2.2.2.
[2] Neighborhood search.

de programació amb restriccions, però és normal ja que la naturalesa de les cerques per veïnatge és diferent a les de la cerques sistemàtiques.

Des del punt de vista computacional, el Comet és diferent als altres sistemes de programació amb restriccions existents. Al Comet les restriccions no hi són per a podar l'espai de cerca. Més aviat mantenen unes propietats que serviran per dirigir l'exploració eficientment. Per exemple, una restricció típicament manté la informació de quant contribueixen les seves variables a la seva violació; aquesta informació sovint és útil per escollir el següent veí. A més a més, les restriccions són *objectes diferenciables*, que vol dir que poden ser consultats per determinar l'impacte de moviments locals segons les seves propietats. A nivell d'implementació, les restriccions, i els objectes diferenciables en general, encapsulen algorismes incrementals que sorgeixen en diverses aplicacions. Però des d'un punt de vista conceptual, hi ha diverses semblances entre Comet i els sistemes tradicionals de programació amb restriccions. En particular, tots dos poden separar clarament la modelització del problema de la cerca de la solució, proporcionant modularitat, composició i possibilitats d'extensió.

NP-SPEC

Segons [CS05], NP-SPEC és un llenguatge declaratiu d'alt nivell per a resoldre problemes de cerca. Existeixen dues versions sobre les quals pot treballar el llenguatge, la primera treballa amb Prolog i la segona treballa amb lògica proposicional. L'última dóna més bons resultats i per tant només comentarem aquesta. La semàntica està basada en la noció de *model minimality*, una extensió de la semàntica coneguda de Horn amb el mínim punt fix, fragment de la lògica de primer ordre. NP-SPEC permet a l'usuari expressar tots els problemes que formen part de la classe de complexitat NP, on hi ha inclosos reconeguts problemes de la vida real. Determinades limitacions en l'expressivitat de NP-SPEC permet garantir l'acabament i ajuda a obtenir execucions eficients.

El nucli del sistema de NP-SPEC és SPEC2SAT, que tradueix les especificacions de problemes escrites en NP-SPEC a instàncies SAT. Una instància Π del problema original es tradueix a una fórmula T de la lògica proposicional en forma normal conjuntiva (CNF), tal que T és satisfactible si i només si Π té solució. A més a més, construeix la solució de Π a partir de les assignacions de variables de T.

MiniZinc

El MiniZinc [NORCR07] és una simplificació del llenguatge Zinc, i intenta ser un llenguatge de modelatge de nivell mitjà. Permet expressar la majoria de problemes de restriccions de manera fàcil i independent del resoledor. Suporta conjunts, arrays, i predicats definits per l'usuari. És de primer ordre i suporta variables de decisió dels tipus que suporten la majoria dels resoledors de CP: enters, reals, booleans i conjunts d'enters. Entre d'altres coses, també permet separar la modelització del problema (model) de les dades (instància). Proporciona una llibreria on hi ha definicions d'algunes restriccions globals i també té un sistema d'anotacions que permet posar informació no declarativa (com estratègies de cerca) i informació específica dels resoledors (com representació de variables), per sobre del model declaratiu. Una de les característiques més interessants de MiniZinc és que permet la compilació d'instàncies de problemes a un "codi intermig" anomenat FlatZinc pel que existeixen diversos resoledors (Sicstus, G12, ECL^{i}PS^e, Gecode o FZNTINI)

2.2.2 Resolució de CSPs

Aquesta secció segueix bàsicament la descripció presentada a [BHZ06].

Al llarg dels anys s'han proposat un bon nombre d'algorismes i tècniques per resoldre problemes de satisfacció de restriccions i optimització. Però els resoledors més actuals només utilitzen un nombre

reduït d'aquests algorismes i tècniques, alguns dels quals daten dels anys 60. En general es poden diferenciar en dues categories de mètodes de resolució de restriccions: mètodes incomplets i mètodes complets.

- **Incomplets**. Els mètodes incomplets intenten trobar solucions per mitjans heurístics sense explorar exhaustivament l'espai de cerca. Aquests mètodes tenen l'inconvenient que generalment no poden detectar quan el problema no té solució. Quan no s'ha trobat solució al cap d'un límit de temps no es pot dir si no existeix cap solució pel problema o si s'ha "saltat" la solució. En general aquests mètodes solen ser estocàstics, que fan moviments aleatoris.
- **Complets**. Els mètodes complets intenten explorar tot l'espai de solucions, generalment utilitzant cerca per *backtracking*. Donat que la cerca exhaustiva en l'espai de cerca és molt costosa és convenient acotar-la, ja sigui utilitzant tècniques de poda determinant quina és la millor variable a la que se li ha d'assignar valor primer, etc.

A part d'aquestes dues categories també s'han inventat altres maneres de resoldre aquest tipus de problemes, les quals només anomenarem i no entrarem en detalls ja que no és rellevant per aquest treball: combinacions entre mètodes complets i incomplets, arquitectures alternatives per mirar d'atacar els problemes com la CP en paral·lel (veure [Ham03] per a veure l'efecte del multithreading en CP), CSP distribuïts (veure DiscCSP [Yok90]) o inclús mitjançant *hardware* específic.

Els tipus de mètodes més emprats solen ser els mètodes complets, que són en els que ens centrarem nosaltres (ens interessa trobar, sempre que existeixi, una resposta). Aquests mètodes, com ja hem dit, estan basats típicament en l'algorisme de cerca de *backtracking* (o variants). Podem veure'n l'esquema bàsic a l'Algorisme 2, on la crida inicial és `Backtracking(`$X, \{\}, C$`)`.

Algorithm 2 Backtracking

Require: X : Variables, A : Assignació, C : Restriccions

```
1:  if X = {} then
2:     return A;
3:  end if
4:  x := tria_una_variable(X);
5:  for all v in domini(x) do
6:     if consistent(A++(x, v), C) then
7:        R := backtracking(X − {x}, A++(x, v), C);
8:        if R != fail then
9:           return R;
10:       end if
11:    end if
12: end for
13: return fail;
```

L'algorisme de backtracking és molt semblant al *generate and test* (Algorisme 1), però té una diferència substancial que és que quan assignem un valor a una variable mirem si el valor assignat és consistent amb els valors ja assignats segons el conjunt de restriccions (és a dir, que els valors assignats fins al moment amb el que estem assignant ara, no violin cap restricció). Només fent això ja es pot reduir molt la cerca. En canvi en l'algorisme de *G*enerate and Test si el valor assignat a la variable actual no era consistent, aniríem provant tots els valors per a la resta de variables abans de canviar el de l'actual.

En els següents apartats veurem refinaments i millores d'aquest algorisme. Primer veurem els criteris de *branching*, és a dir, com determinar quina variable se li assigna valor primer i quin valor se li dóna. Tot seguit veurem com es pot fer *prunning* a l'arbre de cerca, detectant si l'assignació de valors a les variables ens duu a branques inconsistents. Llavors veurem algunes de les tècniques per millorar el *backtracking*. Finalment parlarem una mica de com es realitzen les cerques d'optimització.

Branching

En general a l'hora de seleccionar la variable a donar valor, i el valor a assignar, es segueix el principi de *first fail*, que bàsicament diu que primer es provi per on hi ha més possibilitats de fallar. Tot i això, existeixen diverses heurístiques que tot seguit comentem.

- Escollir variable:
  - MINDOM: es selecciona la variable que té el domini més petit.
  - MAXDEG: es selecciona la variable que apareix a més restriccions.
  - DOMDEG: es selecciona la variable que té el domini més petit i apareix a més restriccions.
  - LEX: considerar les variables d'acord a un ordre predefinit per l'usuari.
- Escollir valor per a una variable:
  - MIDDLE: primer el valor de la mediana del domini.
  - RANDOM: aleatòriament.
  - LEX: en ordre lexicogràfic.
  - INVLEX: en ordre lexicogràfic invers.

S'ha de tenir en compte que a part de les heurístiques comentades l'usuari pot definir les seves pròpies. I tot i així ens podríem trobar que l'heurística escollida no sigui la més adient i quedar-nos estancats en l'arbre de cerca. Per aquest motiu, han sorgit estratègies de cerca més intel·ligents:

La *Limited Discrepancy Search* (LDS [HG95]) es basa en què una heurística ben escollida s'equivoca pocs cops al llarg de la seqüència d'eleccions. La cerca comença aplicant l'heurística, si no es troba solució després s'exploren les altres seqüències d'eleccions. La manera d'explorar-les és la següent, primer s'explora fins que arribem a un punt de decisió, no és té en compte l'heurística (això és coneix com *discrepància*, i a partir d'aquí tornem a tenir en compte l'heurística per a la resta de punts de decisió. En cas que no es torni a trobar solució, es va incrementant el nombre de discrepàncies fins arribar al màxim de permeses.

En la *Interleaved Depth-First Search* (IDFS [Mes97]) en lloc de fer la cerca *Depth-First Search* (DFS) normal es manté un subconjunt d'arbres actius on fer la cerca. Quan s'arriba a una fulla, si no és la solució, es guarda l'estat del subarbre actual i es passa a explorar un altre subarbre actiu. La idea és que interessa evitar fer eleccions dolentes en les etapes inicials de ramificació perquè poden dur a explorar subarbres molt grans que no tenen solució. La mateixa idea porta a variants de la LDS, com pot ser el *Depth-Bounded Discrepancy Search* (DBDS [Wal97]), on s'aplica la idea del LDS només als nodes de les etapes inicials de l'arbre. A [MW98] hi ha una explicació de les diferències entre DBDS i IDFS.

Prunning

Els resoledors, a més a més, necessiten mecanismes de deducció per detectar les conseqüències de les assignacions imposades. Aquestes conseqüències li permetran identificar les branques inconsistents ràpidament i així poder-les podar. En el cas de CP es tracta de la *propagació de restriccions* i consisteix bàsicament en eliminar els valors dels dominis de les variables que no siguin vàlids segons la tria que haguem fet prèviament.

Els mètodes de propagació van aparèixer a principis dels anys 70 en el context de CSP, concretament en el reconeixement d'escenes a [Wal75]. A [Mon74], treball també inspirat per aplicacions de processament d'imatges, s'introdueix la noció de *path-consistency*, i finalment a [Mac77], apareix la noció d'*arc-consistency* (AC).

La majoria de motors de propagació moderns estan basats en l'algorisme de propagació anomenat AC-3 [Mac77] que, tot i no ser òptim per segons quin tipus de restriccions, és general i flexible. La idea és raonar localment considerant cada restricció per torns. Cada restricció reacciona a les modificacions del domini d'alguna de les seves variables, reduint els dominis de les altres variables del seu àmbit si és necessari. Per exemple, la restricció $x \neq y$ farà que si x pren per valor a llavors es treurà aquest valor del domini de y. Típicament es manté una cua que conté totes les variables que s'han modificat recentment o les restriccions que depenen d'aquestes variables (es pot implementar dependent de les variables o de les restriccions).

La majoria de resoledors tant poden representar els rangs de valors permesos a les variables per dominis explícits com per límits. Per exemple, a GNU-Prolog [CD96] la representació canvia automàticament d'un domini explícit emmagatzemat en un vector de bits cap a un interval depenent de la mida del domini. Com a exemple de *propagador* associat a una restricció aquí tenim les regles de propagació de l'interval per a la restricció $x + y = z$ (on $lb(x)$ i $ub(x)$ representen el límit inferior i superior de la variable x, respectivament):

```
when lb(x) modified do:  lb(z) := lb(x) + lb(y)   ub(y) := ub(z) - lb(x)
when lb(y) modified do:  lb(z) := lb(x) + lb(y)   ub(x) := ub(z) - lb(y)
when lb(z) modified do:  lb(x) := lb(z) - ub(y)   lb(y) := lb(z) - ub(x)
when ub(x) modified do:  ub(z) := ub(x) + ub(y)   lb(y) := lb(z) - ub(x)
when ub(y) modified do:  ub(z) := ub(x) + ub(y)   lb(x) := lb(z) - ub(y)
when ub(z) modified do:  ub(x) := ub(z) - lb(y)   ub(y) := ub(z) - lb(x)
```

Backtracking no cronològic

Per més bé que s'escullin les heurístiques sempre hi ha conflictes durant la cerca que provoquen que el resoledor hagi de tornar a un punt anterior (faci *backtracking*). La implementació més naïf en algorismes de *backtracking* torna un nivell per sobre en l'arbre de cerca i escull, si és possible, un valor diferent per a la variable de ramificació. Algorismes més intel·ligents de *backtracking* analitzen el conflicte i reculen fins al nivell de decisió que solventa el conflicte. Això es coneix com *backtracking* no cronològic (de vegades també dit *backjumping*).

Analitzant els conflictes els resoledors poden adquirir coneixements i emmagatzemar-los per prevenir que conflictes del mateix estil tornin a succeir. Aquest procés és conegut com *no-good learning*.

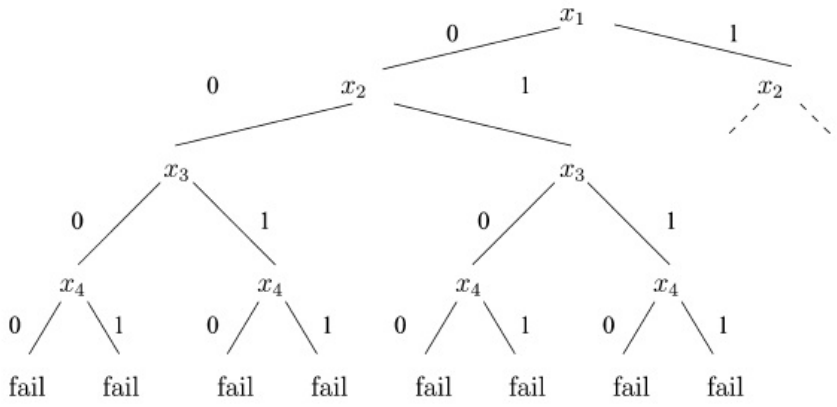
S'han proposat diferents tècniques per analitzar conflictes. Quan es detecta la inconsistència d'una branca aquests mètodes intenten analitzar les raons de la inconsistència i utilitzar-ho per tornar enrere d'una manera més intel·ligent. A vegades el conflicte és degut a decisions preses a un nivell inicial de l'arbre de cerca, i per més que canviem valors d'altres variables intermitges i no el d'aquella elecció, continuarem trobant la mateixa inconsistència. L'anàlisi de conflictes ens permet retrocedir al nivell

més profund, de l'arbre de decisió, que intervé realment a la causa del conflicte. Això evita que el resoledor faci feina redundant i pot evitar detectar les mateixes inconsistències diverses vegades.

Una de les tècniques més rellevants és el *conflict-directed backjumping* [Pro93] (CBJ). Per veure'n les idees principals considerarem un CSP sobre les variables $x_1, \ldots, x_5 \in \{0, 1\}$ i les següents restriccions:

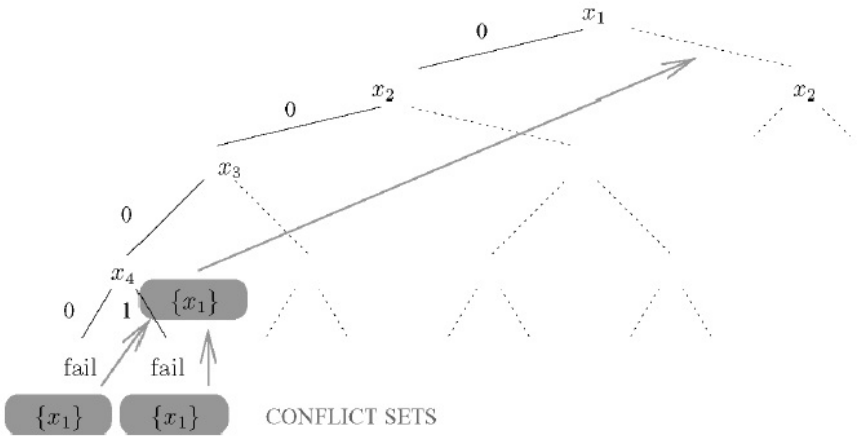
$$(c_1)\ x_4 \neq x_5 \qquad (c_2)\ x_2 + x_3 + x_5 \geq 2x_1 \qquad (c_3)\ x_1 + x_4 = x_5$$

Si assignem el valor 0 a x_1 tindrem com a conseqüència una contradicció per c_1 i c_3, $(x_4 \neq x_5)$ i $(0 + x_4 = x_5)$ respectivament. Aquesta inconsistència no és òbvia per a resoledors amb propagació perquè no apareix cap contradicció tenint en compte les restriccions separadament. Típicament el resoledor detecta la inconsistència quan x_4 o x_5 s'instancien, per exemple donant el valor 0 a x_4, llavors el resoledor deduirà que x_5 també s'haurà d'instanciar amb el valor 0 (propagació mitjançant c_3) i que portarà a una contradicció (per la restricció c_1).



La imatge anterior ajuda a veure perquè el resoledor pot fer la mateixa deducció diverses vegades, fent repeticions innecessàries per les branques de l'arbre. En el nostre cas, mentre s'inspecciona la branca $x_1 = 0$ el resoledor detecta inconsistències degut a assignar successivament valors de 0 i 1 a la variable x_4, acabant cada cop amb conflicte.

El que passa en l'exemple anterior és que el conflicte és independent dels valors que s'assignin a x_2 i x_3. Les restriccions que es violen a les fulles són sempre c_1 i c_3, que no involucren cap de les dues variables x_2 ni x_3. Analitzant el conflicte, s'observa que l'elecció de $x_1 = 0$ s'ha de reconsiderar, ja que qualsevol branca que tingui aquesta elecció serà inconsistent. La idea del CBJ és mantenir un seguiment de les causes dels conflictes. Concretament, el que fa és mantenir un conjunt de conflictes, a cada node de l'arbre de cerca, que contingui les variables involucrades en les deduccions fetes pel motor de propagació en aquell node.



Un exemple de com es mantenen els conjunts de conflictes el podem veure en la imatge anterior. Les úniques propagacions en aquest exemple són les que causen la fallada a cada fulla. En cada cas, les instanciacions de les variables x_1 i x_4 són la raó de la inconsistència, i per tant mantenim un seguiment del fet que x_1 és la única variable que està relacionada amb el conflicte i que s'ha instanciat en un nivell més alt. Ara quan es detecta la fallada per tots els possibles valors de x_4, es calcula la unió del conjunt de conflictes obtingut per cada ramificació, obtenint $\{x_1\}$. Llavors es fa *backjump* al nivell més profund de les variables que apareixen al conjunt de conflictes, i anem directament a escollir un nou valor per a x_1.

Optimització

En la majoria d'aplicacions de CP el problema no consisteix només en trobar una solució, sinó que es vol trobar una solució òptima segons un criteri, bàsicament mitjançant una funció objectiu.

La manera més general d'expressar aquest criteri és permetre a l'usuari especificar una funció objectiu. El resoledor haurà de trobar una de les solucions tal que llur avaluació per aquesta funció sigui òptima. En el cas que hi hagi més d'un criteri (optimització multicriteri o multiobjectiu), és més difícil decidir quina solució és més òptima.

Una de les tècniques principals en optimització és el *branch and bound*. Consisteix en anar creant conjunts de candidats a ser solució, estimant un límit superior i límit inferior a cada conjunt de candidats. Després es descartaran conjunts de candidats comparant els límits inferiors amb els límits superiors, dependrà de si es vol minimitzar o maximitzar. Per exemple, en el cas de voler minimitzar es descartaran tots els conjunts de candidats que tinguin un límit inferior més gran que el límit superior del conjunt de candidats que s'està mirant actualment.

Algunes variants utilitzen una tècnica que es coneix com *optimistic partitioning*, que consisteix en fer una cercà dicotòmica entre el valor mínim i el valor màxim que pot donar la funció objectiu. La hipòtesi és que aquestes divisions optimistes permetran convergir més ràpidament cap a bones solucions que amb l'aproximació clàssica.

Una altra variant és la Russian Doll Search(RDS) que ve de l'àrea de problemes de *scheduling*. La idea és solucionar successivament subproblemes creixents aniuats. Per fer-ho es defineix un ordre per a les variables. A partir d'aquest ordre es busca solució per la última variable, i després es busca solució iterativament incrementant el nombre de variables. Aprofitant la solució de cada subproblema s'impulsa la resolució dels següents. Aquest aprofitament de coneixement fa més ràpid tot el conjunt de resolucions que no pas atacar directament el problema original.

Mentre les funcions objectiu permeten a l'usuari expressar les seves preferències sobre les possibles solucions del problema, d'una manera arbitràriament complexa, hi ha casos en què les preferències es poden expressar utilitzant marcs conceptuals especialitzats. Per exemple algú podria voler:

- maximitzar la qualitat de les solucions segons la seva robustesa: es prefereixen solucions que continuïn essent vàlides encara qui hi hagi petits canvis en el problema.
- resoldre problemes sobre-restringits: quan no es poden satisfer totes les restriccions d'un problema, a vegades és preferible proposar una solució parcial. S'han proposat diverses aproximacions per a satisfaccions parcials que maximitzen el nombre de restriccions satisfetes (MAX-CSP). Donat que no sempre totes les restriccions tindran la mateixa importància, una alternativa és permetre a l'usuari expressar les seves preferències donant pesos a les restriccions o especificar un ordre per dir quines s'han de satisfer primer. Diverses aproximacions s'han proposat per solucionar problemes de *soft constraints* (per exemple *valued CSP*, CSP basats en semianells, etc.).

Un cop vistes les idees principals de com resoldre CSPs des de la perspectiva de CP, en els següents apartats explicarem què és SAT i què és SAT Mòdul Teories (SMT). A simple vista podrem veure que hi ha certs paral·lelismes amb el que hem vist fins ara. Per exemple, en els CSPs intentem satisfer un conjunt de restriccions i a SAT intentem satisfer un conjunt de clàusules. Inclús a nivell expressiu, com podrem veure, en SMT (segons la teoria de fons que vulguem emprar) podrem utilitzar expressions aritmètiques tal com podem fer en els CSPs.

Les següents seccions segueixen bàsicament la descripció presentada a [Seb07].

2.3 Satisfactibilitat Booleana (SAT)

SAT és el problema de decidir si una fórmula booleana en lògica proposicional és satisfactible o no. Un resoledor de SAT és un procediment que decideix si una fórmula booleana és satisfactible, i retorna l'assignació que la fa satisfactible en el cas que en sigui.

Recordem ara breument els principals conceptes de la lògica booleana que farem servir. Una fórmula en *forma normal conjuntiva* (CNF), està formada per una conjunció de clàusules ($c_1 \wedge c_2 \wedge \cdots \wedge c_n$), on cada clàusula estarà formada per una disjunció de literals ($c_i = l_1 \vee l_2 \vee \cdots \vee l_m$), i un literal és una variable booleana (típicament $a, b, ...$) o una negació de variable booleana (e.g. $\neg a$). Una fórmula és satisfactible si i només si existeix una assignació de valors de veritat a cada variable que ens permeti avaluar a cert la fórmula (de vegades generalitzem aquesta noció d'interpretació i ens referirem a literals enlloc de a variables). de les interpretacions que satisfan les fórmules en direm models.

La majoria de procediments actuals per a SAT són evolucions del procediment *Davis-Putman-Logemann-Loveland* [DP60, DLL62] (DPLL). A diferència de la representació clàssica del DPLL les implementacions modernes del DPLL són dirigides per conflicte[3]. Es beneficien de tècniques de cerca sofisticades, decisions heurístiques, estructures de dades molt elaborades i de filigranes en la implementació, així com de tècniques de preprocessament.

En la Figura 2.4 podem veure una formulació abstracta basada en regles per al DPLL (Abstract-DPLL logical framework) corresponent a l'algorisme 3 que descriurem tot seguit a la propera subsecció. D'acord amb aquesta descripció, podem dir que les cinc primeres regles d'aquesta formulació

[3]Els motors DPLL moderns es poden classificar bàsicament en dues families: DPLL dirigit per conflicte i DPLL amb look-ahead.

$$
\begin{array}{lllll}
\text{Unit-Propagate:} & \langle \mu \mid \varphi, C \vee l \rangle & \Rightarrow \langle \mu, l \mid \varphi, C \vee l \rangle & if & \left\{ \begin{array}{l} \mu \models \neg C \\ l \; is \; undefined \; in \; \mu \end{array} \right. \\
\text{Decide:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu, l \mid \varphi \rangle & if & \left\{ \begin{array}{l} l \; or \; \neg l \; occurs \; in \; \varphi \\ l \; is \; undefined \; in \; \mu \end{array} \right. \\
\text{Fail:} & \langle \mu \mid \varphi, C \rangle & \Rightarrow fail & if & \left\{ \begin{array}{l} \mu \models \neg C \\ \mu \; contains \; no \; decision \; literals \end{array} \right. \\
\text{Backjump:} & \langle \mu, l, \mu' \mid \varphi, C \rangle & \Rightarrow \langle \mu, l' \mid \varphi, C \rangle & if & \left\{ \begin{array}{l} \mu, l, \mu' \models \neg C \\ there \; is \; some \; clause \; C' \vee l' \; s.t. : \\ \quad \varphi, C \models C' \vee l' \; and \; \mu \models \neg C' \\ \quad l' \; is \; undefined \; in \; \mu \\ \quad l' \; or \; \neg l' \; occurs \; in \; \varphi \; or \\ \quad\quad in \; \mu \cup \{l\} \cup \mu' \end{array} \right. \\
\text{Learn:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu \mid \varphi, C \rangle & if & \left\{ \begin{array}{l} all \; atoms \; in \; C \; occur \; in \; \varphi or \; in \; \mu \\ \varphi \models C \end{array} \right. \\
\text{Discharge:} & \langle \mu \mid \varphi, C \rangle & \Rightarrow \langle \mu \mid \varphi \rangle & if & \{ \; \varphi \models C \\
\text{Restart} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \emptyset \mid \varphi \rangle &
\end{array}
$$

Figura 2.4: Abstract-DPLL logical framework. A la regla de backjump, C i $C' \vee l'$ representen la clàusula del conflicte i la clàusula conflictiva respectivament.

abstracta representen respectivament la unit-propagation del pas `deduce`, la selecció de literal del `decide_next_branch`, el pas de fallada de les files 12-13 de l'algorisme 3, el backjumping i el mecanisme d'aprenentatge del `backtrack` i `analyze_conflict` respectivament. Les dues últimes regles representen els mecanismes de descàrrega i restart.

2.3.1 DPLL dirigit per conflicte

Un esquema a alt nivell d'un motor DPLL dirigit per conflicte el podem veure a l'Algorisme 3. La fórmula booleana ϕ està en CNF; l'assignació μ inicialment està buida, i es va actualitzant com una pila.

Primer de tot mirarem com treballa l'algorisme a grans trets i després ja parlarem de què fan internament les diferents funcions que el componen.

La funció `preprocess`(ϕ, μ) simplifica ϕ a una fórmula més simple i *equisatisfactible*, i si s'escau actualitza el paràmetre d'entrada i sortida μ . Si la fórmula resultant és insatisfactible llavors el DPLL retorna `Unsat`.

En el bucle principal, `decide_next_branch`(ϕ, μ) escull un literal l no assignat de ϕ i el seu valor segons el criteri de l'heurística, i l'afegeix a μ. (Aquesta operació es coneix com *decisió*, l es coneix com el *literal de decisió* i el nombre de literals de decisió dins de μ després d'aquesta operació es coneix com el *nivell de decisió* de l).

En el bucle interior, `deduce`(ϕ, μ) dedueix iterativament el valor de literals derivant de l'assignació actual i d'acord amb això n'actualitza els seus valors a ϕ i μ (l'aplicació iterativa de passos de deducció booleana dins `deduce` també es coneix com *Boolean Constraint Propagation, BCP*). Aquest pas es repeteix fins que:

- μ satisfà ϕ retornant `Sat`, i per tant el DPLL retorna `Sat`
- μ falsifica ϕ retornant `Conflict`. En aquest cas, si `analize_conflict`(ϕ, μ) pot detectar el subconjunt η de μ que causa el conflicte (*conflict set*) i el nivell de decisió `blevel` on farem

Algorithm 3 Esquema Basic d'un DPLL modern basat en conflicte

Require: ϕ : Formula_Booleana, $\&\mu$: Assignació

```
1: if preprocess(φ, μ) = Conflict then
2:    return Unsat
3: end if
4: while true do
5:    decide_next_branch(φ, μ);
6:    while true do
7:       status := deduce(φ, μ);
8:       if status = Sat then
9:          return Sat
10:      else if status = Conflict then
11:         blevel := analyze_conflict(φ, μ);
12:         if blevel = 0 then
13:            return Unsat;
14:         else
15:            backtrack(blevel,φ, μ);
16:         end if
17:      else
18:         break;
19:      end if
20:   end while
21: end while
```

backtracking, mirarem: si `blevel=0`, llavors el conflicte existeix sense haver-se ramificat, i per tant el DPLL retorna `Unsat`; altrament, `backtrack(blevel,`ϕ, μ`)` afegeix $\neg\eta$ dins ϕ (*learning*) i fa backtracking a `blevel` (*backjumping*), actualitzant ϕ i μ.

- No es pot deduir el valor de cap més literal i l'assignació no satisfà ni falsifica la fórmula retornant `Unknown`, cas en que el DPLL surt del bucle interior, i passa a mirar la següent decisió.

Un cop vist per sobre com treballa l'algorisme ara veurem en més detall les diferents funcions que hem comentat.

- La funció `preprocess` implementa tècniques de simplificació com per exemple:
  - Detectar equivalències Booleanes entre els literals.
  - Aplicar passos de resolució a les parelles de clàusules seleccionades.
  - Detectar i eliminar clàusules *subsumides*.
  - Aplicar BCP si és dóna el cas.
- La funció `decide_next_branch` implementa el pas clau no determinista del DPLL, pel qual han aparegut diferents criteris heurístics.

  Les heurístiques clàssiques com MOMS i *JeroslowWang* seleccionen un literal nou a cada punt de ramificació, agafant el literal que apareix més sovint en les clàusules de mida mínima.

L'heurística implementada al SATZ selecciona un conjunt de candidats, fa BCP i escull el candidat que portarà al conjunt de clàusules més petit. Això maximitza la propagació booleana, però introdueix un gran overhead.

Quan les fórmules deriven de la codificació d'un problema específic, molts cops és útil permetre al codificador subministrar al DPLL una llista de variables prioritàries per a ramificar primer. Els resoledors DPLL moderns guiats per conflicte adopten evolucions de l'heurística VSIDS, on els literals de decisió es seleccionen d'acord a una puntuació. Aquesta puntuació només s'actualitza al final de la branca i quan apareixen variables prioritàries en clàusules recentment apreses. Això fa que `decide_next_branch` sigui independent de l'estat (i per tant més ràpid ja que no hi ha necessitat de recalcular les puntuacions a cada decisió) i permet tenir en compte l'històric de cerca, que fa la cerca més robusta i eficient.

- La funció `deduce`, en general, està basada en l'aplicació iterativa de la *unit-propagation*, que consisteix en que si trobem un literal l com a clàusula unitària, podem eliminar la resta de clàusules que continguin l i també eliminar totes les aparicions de $\neg l$ de la resta de clàusules (per exemple, $\{a \vee b, \neg a \vee c, \neg c \vee d, a\}$ quedaria $\{c, \neg c \vee d, a\}$, on podríem tornar a aplicar unit-propagation ja que ens apareix c i tenim una clàusula on apareix $\neg c$). Estructures de dades complexes i filigranes en la implementació (com per exemple el *two-watched-literal scheme*) permeten implementacions extremadament eficients de la programació unitària.

  És important veure que la majoria de resoledors DPLL dirigits per conflicte no retornen `Sat` quan les clàusules es satisfan, sinó quan totes les variables tenen assignat un valor. Com a conseqüència els resoledors SAT dirigits per conflicte moderns típicament retornen assignacions totals de veritat, encara que les fórmules es satisfacin amb assignacions parcials.

- Les funcions `analyze_conflict` i `backtrack` treballen de la següent manera. Cada literal es marca amb el seu nivell de decisió, això és: el literal corresponent a la n-èsima decisió i tots els literals derivats per la *unit-propagation* després d'aquesta decisió s'etiquetaran amb n; cada literal derivat l es marcarà amb un enllaç a la clàusula C_l que ha causat la *unit-propagation* (anomenada com *clàusula antecedent* de l). Quan una clàusula C es falsifica per l'assignació actual – direm que hi ha hagut un *conflicte* i C és la *clàusula del conflicte* – típicament es calcula una clàusula conflictiva C' a partir de C, tal que C' només conté un literal l_u que se li ha assignat valor a l'últim nivell de decisió. C' es calcula començant per $C' = C$ i fent resolució iterativament de C' amb la clàusula antecedent C_l d'algun literal l de C' (típicament l'últim literal assignat a C'), fins que ens trobem amb el criteri de parada (típicament quan C' només conté un literal amb valor assignat).

  Un cop calculada C' a `analize_conflict` s'afegeix a la fórmula ϕ, i llavors `backtrack` treu tots els literals assignats de μ fins a un nivell de decisió `blevel` derivat de C', que es calcula a `analyze_conflict` segons l'estratègia. En les implementacions modernes del DPLL dirigit per conflicte, recula fins al punt més alt de la pila on el literal l_u après de C' no està assignat, i es fa *unit-propagation* de l_u.

  L'aprenentatge s'ha d'emprar amb molt de compte, ja que pot provocar una explosió en la mida de ϕ. Per evitar aquest problema les eines modernes de DPLL dirigit per conflicte implementen tècniques per descartar clàusules apreses quan és necessari (descarregar). A més, per tal d'evitar quedar encallat a parts dures de l'espai de cerca, la majoria d'eines DPLL *tornen a començar* (restart) la cerca des de zero d'una manera controlada; les clàusules apreses eviten que s'explori

el mateix espai de cerca altre cop. La descàrrega de clàusules i el restart són substancialment ortogonals per a la discussió de SMT i s'utilitzen típicament en els resoledors DPLL actuals.

2.4 Satisfactibilitat Mòdul Teories (SMT)

SMT és el problema de decidir si una fórmula proposicional on es permeten literals formats per predicats amb interpretacions predefinides segons teories de fons, és satisfactible o no. Per exemple, $a \vee b \vee (x + 2 \leq y) \vee (x > y + z)$, on a i b són variables booleanes i x, y i z són variables enteres. Els predicats sobre les variables no booleanes, com les desigualtats lineals enteres, s'avaluen d'acord a les regles de la teoria de fons.

Formalment una teoria seria un conjunt de fórmules de primer ordre tancada per implicació lògica. Així doncs, una manera equivalent i alternativa de dir què és el problema d'SMT per una teoria donada seria: donada una fórmula de primer ordre F, determina si $T \cup F$ és satisfactible. Normalment es consideraran teories decidibles i F estaria lliure de quantificadors.

A grans trets, la implementació més comú de resoledor SMT (*SMT-solver* a partir d'ara) consta d'un resoledor SAT (*SAT-solver* a partir d'ara) més un (o més) resoledor de teoria (*T-solver* a partir d'ara, que serà l'encarregat de comprovar la factibilitat de conjuncions de predicats de la teoria que li va enviant el SAT-solver a mesura que explora l'espai de cerca Booleà de la fórmula). Per tant, el T-solver serà el que ens determinarà quin tipus d'àtoms poden aparèixer en les fórmules booleanes (per exemple, en aritmètica lineal entera un àtom d'una clàusula podria ser $x + 3 > y$). A l'hora de mirar si un problema té solució, el SMT-solver ja s'encarregarà de repartir la feina entre els dos components. Aquesta manera de resoldre el problema es coneix com a *lazy SMT*.

També es va proposar una formulació abstracta basada en regles per al DPLL basat en sistemes Lazy SMT (Abstract-DPLL Modulo Theories logical framework). Aquesta formulació estén el marc conceptual abstracte DPLL afegint el conjunt de regles de la Figura 2.5 a les de la Figura 2.4 (Cal observar que tots els literals, assignacions i fórmules estan en el llenguatge de T, i que aquí el símbol "$\models$" de la Figura 2.4 s'han d'interpretar com un "$\models_p$"[4].). Les primeres tres regles representen respectivament la noció de T-propagation, T-backjumping i T-learning que veurem a la següent subsecció. La quarta regla guarda relació amb que les clàusules T-apreses que s'han de "descarregar" quan sigui necessari, per tal d'evitar l'explosió en la mida del de la fórmula d'entrada.

2.4.1 Resoledor per a una teoria (T-solver)

Per definir de manera senzilla un T-solver podríem dir que es tracta d'un procés que ens diu quan una assignació (un conjunt finit o conjunció) de literals (d'una teoria determinada) és T-satisfactible (satisfactible segons la teoria) o no. On un literal serà un àtom de la teoria de fons (per exemple, en aritmètica lineal $(3v_1 - 2v_2 \leq 3)$) o una negació d'un àtom (també en aritmètica lineal $(\neg(2v_2 - v_3 > 2))$. En cas de ser satisfactible, el model de l'assignació ens donaria el valor de cada una de les variables (de la teoria) que apareixen en els literals.

Per aconseguir una màxima eficiència dependrà de dos aspectes: l'efectivitat de la interacció amb el DPLL i l'eficiència en temps i memòria. El primer dependrà de la capacitat del T-solver de produir, intercanviar i explotar la informació útil amb el DPLL. I el segon dependrà bàsicament de la teoria T i de l'eficiència dels algorismes dedicats per a aquesta.

[4] A partir d'ara emprarem F^p quan fem referència a una abstracció booleana d'una fórmula F. Per exemple, l'abstracció booleana de la fórmula $x + 3 > y \vee y + 3 > x$ seria $a \vee b$.

$$
\begin{array}{lllll}
\text{T-Propagate:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu, l \mid \varphi \rangle & if & \left\{ \begin{array}{l} \mu \models_T l \\ l \; is \; undefined \; in \; \mu \\ l \; or \; \neg l \; occurs \; in \; \varphi \end{array} \right. \\
\text{T-Backjump:} & \langle \mu, l, \mu' \mid \varphi, C \rangle & \Rightarrow \langle \mu, l' \mid \varphi, C \rangle & if & \left\{ \begin{array}{l} \mu, l, \mu' \models_p \neg C \\ there \; is \; some \; clause \; C' \vee l' \; s.t. : \\ \quad \varphi, C \models_T C' \vee l' \; and \; \mu \models_p \neg C' \\ l' \; is \; undefined \; in \; \mu \\ l' \; or \; \neg l' \; occurs \; in \; \varphi \; or \\ \quad in \; \mu \cup \{l\} \cup \mu' \end{array} \right. \\
\text{T-Learn:} & \langle \mu \mid \varphi \rangle & \Rightarrow \langle \mu \mid \varphi, C \rangle & if & \left\{ \begin{array}{l} all \; atoms \; in \; C \; occur \; in \; \varphi \\ \varphi \models_T C \end{array} \right. \\
\text{T-Discharge:} & \langle \mu \mid \varphi, C \rangle & \Rightarrow \langle \mu \mid \varphi \rangle & if & \{ \; \varphi \models_T C
\end{array}
$$

Figura 2.5: Abstract-DPLL logical framework Modulo Theories. A la regla de T-Backjump, C i $C' \vee l'$ representen la clàusula del conflicte i la clàusula conflictiva respectivament.

Teories més habituals

Les següents teories són les que utilitzarem en aquest projecte. En comentarem les característiques més importants i no entrarem en detalls de combinacions de teories ja que només es vol donar la idea general d'aquestes.

- *Linear Arithmetic (LA)*: la teoria d'aritmètica lineal (ja sigui sobre racionals o enters) és la teoria de primer ordre lliure de quantificadors amb igualtat sobre àtoms de la forma $(a_1 \cdot x_1 + \cdots + a_n \cdot x_n \bowtie a_0)$, on $\bowtie \in \leq, <, \neq, =, \geq, >$ i les a_i representen constants. La LA-satisfactibilitat sobre els enters i lliure de quantificadors és decidible i NP-completa.

- *Difference Logic (DL)*: és una subteoria de LA sobre àtoms de la forma $(x_1 - x_2 \bowtie a)$, on $\bowtie \in \leq, <, \neq, =, \geq, >$ i a representa una constant. La DL-satisfactibilitat sobre els enters i lliure de quantificadors és decidible i de temps polinòmic.

- Altres teories interessants que podrien ser útils a l'hora de resoldre algun tipus concret de problemes i que es podrien fer proves en treball futur són: la teoria *Equality and Uninterpreted Functions (EUF)* i la teoria dels *fixed-width Bit Vectors (BV)*.

Generació del model

Un aspecte important per al T-solver és l'habilitat de donar un model consistent amb la teoria quan li donen una assignació μ a satisfer. En general tots els T-solvers són capaços de produir un model sota demanda de la majoria de teories d'interès.

Per exemple, tenim $\mu = \{\neg(2v_2 - v_3 > 2), (3v_1 - 2v_2 \leq 3), (v_3 = 3v_5 + 4)\}$. Un resoledor $LA(Q)$ dirà que μ és $LA(Q)$-satisfactible i donarà el model: $\{v_1 = v_2 = v_3 = 0, v_5 = -4/3\}$

Cal observar que a vegades produir un model demana un esforç extra de còmput. Això és degut a que el T-solver ha de realitzar transformacions preservant la satisfactibilitat dels literals d'entrada, i òbviament per construir el model s'ha d'invertir la transformació altre cop.

Generació del conjunt conflictiu

Un aspecte important per a l'eficiència del T-solver, sempre i quan es faci una petició sobre una assignació μ T-inconsistent, és l'habilitat de produir el conjunt conflictiu (millor si és mínim) de μ que ha causat la inconsistència. Donada una assignació μ T-insatisfactible, podem anomenar *conjunt conflictiu de la teoria* (només "conjunt conflictiu" quan no es donin ambigüitats) a una sub-assignació $\mu' \subseteq \mu$ que és T-insatisfactible; direm que μ' és el *conjunt conflictiu mínim de la teoria* si tots els subconjunts estrictes de μ' són T-consistents.

Per exemple, tenim $\mu = \{\neg(2x_2 - x_3 > 2), (x_1 - x_5 \leq 1), (3x_1 - 2x_2 \leq 3), \neg(2x_3 + x_4 \geq 5), \neg(3x_1 - 2x_3 \leq 6), (x_2 - x_4 \leq 6), (x_5 = 5 - 3x_4), (x_3 = 3x_5 + 4)\}$. Un resoledor $LA(Q)$ dirà que μ és $LA(Q)$-insatisfactible, i retornarà el conjunt conflictiu: $\{(3x_1 - 2x_2 \leq 3), \neg(2x_2 - x_3 > 2), \neg(3x_1 - x_3 \leq 6)\}$

Incrementalitat i recuperabilitat (backtrackejable)

Normalment les crides al T-solver són seqüencials i per a assignacions incrementals, de manera que primer se li demana per una assignació μ_1, tot seguit per una assignació $\mu_1 \cup \mu_2$, ... Per això un altre aspecte important per a l'eficiència del T-solver és que sigui *incremental* i capaç de fer *backtrack*. Quan diem *incremental* ens referim que si prèviament ens han fet una consulta per a l'assignació μ_1 i tot seguit ens fan la consulta per $\mu_1 \cup \mu_2$, haurem de recordar-nos de l'assignació prèvia μ_1, suposant que era T-satisfactible, per no tenir de començar de zero a mirar si $\mu_1 \cup \mu_2$ és T-satisfactible. I quan diem backtrackejable volem dir que sigui possible tornar a estats anteriors d'una manera eficient.

Deducció de literals no assignats (propagació)

Per algunes teories és possible implementar el T-solver de manera que quan retorna `Sat`, pot realitzar un conjunt de deduccions de la forma $\eta \models_T l$, on $\eta \subseteq \mu$ i l és un literal d'un àtom encara no assignat de φ. Direm que el T-solver és *complet sota deducció* si pot fer totes les deduccions d'aquest tipus, o direm que no ho és en cas contrari.

Per exemple, tenim $\mu = \{\neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_3 \leq 6), (x_3 = 3x_5 + 4)\}$. Un resoledor $LA(Q)$ dirà que μ és $LA(Q)$-satisfactible, i serà capaç de deduir: $\{\neg(2x_2 - x_3 > 2), \neg(3x_1 - 2x_3 \leq 6)\} \models_{LA(Q)} \neg(3x_1 - 2x_2 \leq 3)$.

En principi tot T-solver té capacitats de deducció ja que sempre es pot cridar el T-solver($\mu \cup \{\neg l\}$) per cada literal no assignat l. Aquesta tècnica es coneix com *plunging*, però a la pràctica, excepte per algunes aplicacions, és molt ineficient.

2.4.2 Lazy SMT: Integració SAT-solver + T-solver

Existeixen dues aproximacions de SMT: *eager SMT* i *lazy SMT*. La diferència principal és com s'utilitza el SAT-solver. Nosaltres ens centrarem en la versió més estesa, la lazy, però abans explicarem breument les característiques generals de les dues aproximacions.

En el *eager SMT* es redueix la T-satisfactibilitat a SAT: la T-fórmula d'entrada es tradueix a una fórmula booleana equi-satisfactible, i el SAT-solver s'utilitza per a validar la satisfactibilitat d'aquesta fórmula. Tot i que s'han aconseguit bons resultats en alguns camps (per exemple, la verificació formal del pipeline de microprocessadors), el problema rau en l'explosió combinatòria en generar la fórmula proposicional quan es treballa amb teories com DL o LA.

En el cas de lazy SMT la versió més senzilla consistiria en: agafar la T-fórmula d'entrada φ i crear la seva abstracció booleana $\varphi^p =_{def} T2B(\varphi)$[5]. Passar φ^p com a paràmetre d'entrada al SAT-solver, que retornarà una llista d'assignacions μ^p de valors booleans als literals de φ^p. Tot seguit es passarà μ com paràmetre d'entrada al T-solver, on $\mu =_{def} B2T(\mu^p)$. Si μ és T-satisfactible el T-solver retornarà `Sat` i el SMT(T)-solver retornarà també `Sat`. En cas que μ no sigui T-satisfactible el T-solver retornarà `Unsat` i el SMT-solver tornarà a demanar una altra assignació de valors μ' al SAT-solver, fins que el T-solver retorni `Sat` o bé no quedin més assignacions possibles, i per tant retornar `Unsat`.

Tot seguit mirarem amb més detall el lazy SMT. Començarem veient de forma general les dues aproximacions de lazy SMT (offline i online) i una comparació de la versió online i la offline.

Integració Offline

Aquest esquema d'integració del DPLL amb el T-solver s'ha proposat independentment per [BDS02] i [MR02]. La seva forma més simple seria la que es veu a l'algorisme 4.

Algorithm 4 Esquema simplificat de la integració offline pel lazy SMT(T)

Require: φ : T-formula
1: φ^p := T2B(φ);
2: **while** DPLL(φ^p,μ^p) = Sat **do**
3: **if** T-solver(B2T(μ^p)) = Sat **then**
4: **return** `Sat`;
5: **end if**
6: $\varphi^p := \varphi^p \wedge \neg\mu^p$;
7: **end while**
8: **return** `Unsat`;

L'abstracció proposicional φ^p de la fórmula d'entrada φ es dóna com a entrada al SAT-solver, que o bé decideix que φ^p és insatisfactible, i per tant φ és T-insatisfactible, o bé retorna una assignació μ^p satisfactible; en aquest últim cas $B2T(\mu^p)$ es dóna com entrada al T-solver. Si es troba que $B2T(\mu^p)$ és consistent, llavors φ és T-consistent. Si no, $\neg\mu^p$ s'afegeix com a clàusula a φ^p, i el SAT-solver es reinicia des de zero amb la fórmula resultant. Cal veure que el DPLL s'empra com un SAT-solver de caixa negra, i el bucle 2-7 treballa com un enumerador, ja que el pas 6 evita al DPLL de trobar la mateixa assignació més d'un cop.

Una manera més eficient és quan el T-solver pot tornar el conjunt conflictiu η que ha causat la T-inconsistència de $B2T(\mu^p)$. Si es dóna el cas, llavors s'afegeix $T2B(\neg\eta)$ a la clàusula φ enlloc de $\neg\mu^p$. Típicament aquest últim és molt més petit que l'anterior, i redueix dràsticament l'espai de cerca.

Integració Online

Aquest esquema d'integració, el DPLL(T), és una variant del DPLL, modificat per a treballar com un enumerador d'assignacions de veritat, on la seva T-satisfactibilitat la comprova el T-solver.

A l'algorisme 5 podem veure l'esquema online DPLL(T) basat en un motor DPLL modern (algorisme 3). Les entrades φ i μ són una T-fórmula i una referència a un conjunt de T-literals (inicialment buit) respectivament. El resoledor DPLL que està integrat al DPLL(T), raona sobre φ^p i μ^p, i els

[5]Considerem $T2B(F)$ i $B2T(F^p)$ com a funcions que generen a partir d'una fórmula F la seva abstracció booleana F^p i viceversa, respectivament.

Algorithm 5 Esquema online de DPLL(T) basat en el DPLL modern

Require: φ : T-formula, $\&\mu$: T-assigment

```
1: if T-Process(φ,μ) = Conflict then
2:    return Unsat;
3: end if
4: φ^p := T2B(φ); μ^p := T2B(μ);
5: while true do
6:    T-decide_next_branch(φ^p,μ^p);
7:    while ture do
8:       status := T-deduce(φ^p,μ^p);
9:       if status = Sat then
10:         μ := B2T(μ^p)
11:         return Sat;
12:      else if status = Conflict then
13:         blevel :=T-analyze_conflict(φ^p,μ^p);
14:         if blevel = 0 then
15:            return Unsat;
16:         else
17:            T-backtrack(blevel,φ^p,μ^p);
18:         end if
19:      else
20:         break;
21:      end if
22:   end while
23: end while
```

actualitza, i el DPLL(T) manté diverses estructures de dades codificant el conjunt $Lits(\varphi)$ i el mapeig bijectiu $T2B/B2T$ dels literals.

La funció `T-preprocess` simplifica φ i actualitza μ si es dóna el cas, de manera que es preservi la T-satisfactibilitat de $\varphi \wedge \mu$. Si hi ha algun conflicte en aquest procés llavors el DPLL(T) retorna `Unsat`. La funció `T-preprocess` combina les tècniques de preprocessament booleanes mencionades a 2.3.1, amb tècniques dependents de la teoria T, on l'efectivitat i l'aplicabilitat d'aquestes tècniques dependrà de T:

- *Normalització dels T-àtoms*: la idea és detectar les parelles d'àtoms sintàcticament diferents però que són T-equivalents (per exemple: $(x_1 + (x_3 + x_2) = 1)$ i $((x_1 + x_3) - 1 = x_2)$), o bé T-equivalents amb la negació d'un dels dos. Mitjançant les regles de reescriptura següents podem transformar àtoms possiblement T-equivalents en idèntics sintàcticament:
  - *Eliminar els operadors duals*: $(x_1 < x_2), (x_1 \geq x_2) \Rightarrow \neg(x_1 \geq x_2), (x_1 \geq x_2)$.
  - *Aplicar l'associativitat*: $(x_1 + (x_2 + x_3) = 1), ((x_1 + x_2) + x_3) = 1 \Rightarrow (x_1 + x_2 + x_3 = 1)$.
  - *Ordenar les variables*: $(x_1 + x_2 + x_3 \leq 1), (x_1 + x_2 - 1 = x_3) \Rightarrow (x_1 + x_2 + x_3 = 1)$.
  - *Aplicar propietats especials de la teoria T*: si T és LA($\mathbb{Z}$) $(x_1 \leq 3), (x_1 < 4) \Rightarrow (x_1 \leq 3)$.

  Cal observar que la normalització no és per fer la representació d'àtoms més simple de treballar, sinó que l'objectiu es reconèixer el màxim possible de literals equivalents, de manera que l'abstracció booleana pugui mapejar-los al mateix literal Booleà. Per a més detalls consultar [GS96].

- *Static learning*: en algun tipus de problemes és possible detectar a priori T-inconsistències obvies i curtes per a assignacions de T-àtoms de *Atoms(φ)*. Per exemple:
  - *assignació de valors incompatibles*: $\{x = 0, x = 1\}$
  - *restriccions de congruència*: $\{(x_1 = y_1), (x_2 = y_2), \neg(f(x_1, x_2) = f(y_1, y_2))\}$
  - *restriccions transitives*: $\{(x - y \leq 2), (y - z \leq 4), \neg(x - z \leq 7)\}$
  - *restriccions d'equivalència*: $\{(x = y), (2x - 3z \leq 3), \neg(2y - 3z \leq 3)\}$

  Si això passa es poden afegir a priori la negació d'assignacions abans que comenci la cerca (per exemple: $\neg(x = 0) \vee \neg(x = 1)$. Així per unit-propagation si s'assigna una de les dues inconsistències es previndrà que s'esdevingui l'altra.

La funció `T-decide_next_branch` té el mateix rol que el `decide_next_branch` de DPLL, però ha de tenir en compte la semàntica de T a l'hora de seleccionar els literals. Tot i que en general no han sortit propostes d'heurístiques realment satisfactòries, i normalment s'empren les heurístiques DPLL estàndard.

La versió més simple de la funció `T-deduce` es comporta gairebé igual que el `deduce` de DPLL: iterativament dedueix literals booleans l^p que es deriven proposicionalment de l'assignació actual ($\varphi^p \wedge \mu^p \models_p l^p$) i actualitza φ^p i μ^p, fins que ens trobem en una de les situacions següents:

1. μ^p viola proposicionalment φ^p ($\mu^p \wedge \varphi^p \models_p \bot$). En aquest cas `T-deduce` es comporta igual que el `deduce` de DPLL retornant `Conflict`.

2. μ^p satisfà proposicionalment φ^p ($\mu^p \models_p \varphi^p$). En aquest cas `T-deduce` invoca el T-solver amb $B2T(\mu^p)$: si aquest retorna `Sat`, llavors el `T-deduce` retorna `Sat`; altrament el T-deduce retorna `Conflict`.

3. No es pot deduir cap més literal. En aquest cas `T-deduce` retorna `Unknown`. Tot i que una versió més elaborada del `T-deduce` pot invocar al T-solver amb $B2T(\mu^p)$ en aquest estat intermig: si el T-solver retorna `Unsat`, llavors el `T-deduce` retorna Conflict. Aquest millora, s'anomena *early prunning* i pot suposar una sobrecàrrega realitzant crides innecessàries al T-solver. Per evitar-ho podem modificar-ho de les següents maneres: (i) emprar una heurística per determinar quan es crida (per exemple, determinant si el nou literal pot causar inconsistència a μ, o sinó cridar el T-solver cada k ramificacions); (ii) només quan haguem de validar subconjunts de μ; (iii) cridar el T-solver cada nou T-àtom afegit a l'assignació.

Es podria implementar una versió més elaborada del `T-deduce` si el T-solver fos capaç de realitzar deduccions sobre literals no assignats $\eta \models_T l$, on $\eta \subset \mu$ i l és un literal no assignat (comentat a 2.4.1). Aquest procés s'anomena *T-propagation*. El T-solver retorna l, llavors el DPLL(T) farà unit-propagation de $T2B(l)$, això farà que nous literals siguin assignats, que provocarà noves crides al T-solver, es deduiran noves assignacions, repetidament causant un espiral positiu entre la T-propagació i la unit-propagació. Es pot aplicar amb més o menys intensitat, tenint en compte la sobrecàrrega que això suposa.

La funció `T-analyze_conflict` és una extensió del `analyze_conflict` del DPLL: si el conflicte produït per `T-deduce` és causat per una fallada booleana (cas 1), `T-analyze_conflict` produirà un conjunt conflictiu booleà η^p i el corresponent nivell `blevel`, tal com està descrit a 2.3.1; si el conflicte és causat per una T-inconsistència (casos 2 i 3) llavors el `T-analyze_conflict` produirà com a conjunt conflictiu l'abstracció booleana de η^p a partir del conjunt conflictiu η produït pel T-solver ($\eta^p := T2B(\eta)$), o computarà el conjunt conflictiu barreja de booleà + teoria a partir de la clàusula conflictiva $\neg T2B(\eta)$ com s'ha comentat a `analyze_conflict`. En el cas que el T-solver no pugui retornar el conjunt conflictiu de la teoria, tota l'assignació μ es farà servir, després de treure tots els literals booleans de μ. Un cop s'han computat el conjunt conflictiu η^p i el `blevel`, la funció `T-backtrack` es comportarà com la seva anàloga `backtrack` de DPLL: afegirà la clàusula $\neg\eta^p$ a φ^p i farà backtrack fins a `blevel`. Aquestes característiques, s'anomenen *T-learning* i *T-backjumping*.

En resum, el DPLL(T) es diferencia respecte el DPLL perquè aprofita:

- una notació estesa de la deducció de literals: no només deducció booleana ($\mu^p \wedge \varphi^p \models_p l^p$), sinó que també deducció de la teoria ($B2T(\mu^p) \models_T B2T(l^p)$);
- una notació estesa de conflicte: no només conflicte booleà ($\mu^p \wedge \varphi^p \models_p \bot$), sinó que també conflicte de la teoria ($B2T(\mu) \models_T \bot$), o inclús conflictes barrejats de teoria i booleans ($B2T(\mu^p \wedge \varphi^p) \models_T \bot$).

Online vs Offline

Tal com s'ha descrit, a l'aproximació offline el SAT-solver es consulta cada cop des de zero però amb la fórmula booleana augmentada. Per una banda, això permet utilitzar un SAT-solver directament del calaix o amb el mínim de modificacions al seu codi font, mentre que a l'aproximació online es necessita una integració més estricta entre el codi font del SAT-solver i el T-solver. Per altra banda, a l'aproximació offline cada crida al SAT-solver ha de repetir part de la cerca que ja s'ha fet en crides prèvies. A l'aproximació online, en canvi, després de cada crida al T-solver la cerca booleana continua des del punt on s'ha interromput, sense refer cap feina. A més, cal observar que en l'aproximació offline és estrictament necessari mantenir les clàusules conflictives amb la teoria apreses per poder garantir la completesa de l'aproximació, mentre que en l'aproximació online el T-learning només és

una tècnica per millorar l'eficiència, de manera que les clàusules conflictives amb la teoria apreses es poden descarregar quan es vulgui. A [FJOS03] es pot trobar un test empíric que mostra la superioritat important de l'aproximació online sobre la offline.

Encara que es facin les millores existents en ambdues aproximacions, es manté la diferència, ja que l'efecte de passar de l'aproximació offline més ingènua a la més efectiva passant a explotar conjunts conflictius és anàloga a l'ús de T-backjumping i T-learning amb l'aproximació online, i l'efecte de la versió "eager notification" de l'aproximació offline és la de (eager) early prunning. Tot i que l'última millora, potser, requereix una integració més estricta entre el codi font del SAT-solver i el T-solver.

En general, l'elecció entre l'esquema offline i online rau en l'equilibri entre eficiència i esforç en la implementació, en particular per a modificar i integrar amb el codi font del SAT-solver. La integració offline és aconsellable per prototipatge, mentre que la integració online és recomanable per construir eines més eficients i estables.

Capítol 3

Resolució del problema

Un cop introduïts a les àrees on s'emmarca el projecte procedirem a explicar el procés que es va dur a terme per a resoldre el problema. El primer pas va consistir en fer un seguit de proves per a veure com es comportaven els SMT-solvers a l'hora de resoldre CSPs. Es van escollir dos problemes dels quals es va fer una modelització directament amb llenguatge SMT. Un cop vist el que el rendiment era raonablement satisfactori, es va procedir a dissenyar el sistema, el llenguatge `Simply` i el compilador a SMT. Una part del treball presentat en aquesta memòria ha estat publicat a ModRef09 [BPSV09] i al Prole09 [BPV09].

3.1 Problemes de prova

Per fer les proves per mirar el rendiment dels SMT-solvers es van generar instàncies dels següents problemes típics de CSP. Es van generar amb programes Haskell ad hoc que generaven el codi SMT-LIB i dels quals en podem trobar les codificacions en Haskell a l'Annex II (Capítol 7). A continuació podem veure una petita explicació de cada problema i una taula amb els temps de resolució de les instàncies en segons executades sobre un Intel Core 2 Duo a 3.00 GHz amb 2Gb de RAM.

- *Golomb's ruler* (*Problema 006 CSP-Lib*): Un Golomb Ruler de llargada l amb n marques es defineix com un conjunt de n enters $0 \leq a_1 < a_2 < ... < a_n \leq l$ tals que les $m(m-1)/2$ diferències $a_j - a_i, 1 \leq i < j \leq n$ són totes diferents. En la nostra formulació, donades una l i una n, es vol trobar un Golomb Ruler de qualsevol llargada no més gran que l.

  A la taula següent podem veure el temps de generació del fitxer SMT i de resolució mitjançant el SMT-Solver Yices, per a diverses instàncies del *Golomb ruler*.

N Marques	Llargada	Temps generació SMT	Temps resolució (Yices)
3	3	0.01	0.01
4	6	0.01	0.5
7	30	0.1	1
8	34	1	9
9	44	2	300

- *Social Golfer* (*Problema 010 CSP-Lib*): Donat un conjunt de p jugadors que volen jugar al golf un cop per setmana, es vol trobar una distribució dels jugadors en g grups de mida s durant w setmanes, de manera que dos jugadors no juguin més d'un cop en el mateix grup. La següent relació s'ha de mantenir a partir de les quantitats que acabem de mencionar: $p = g * s$.

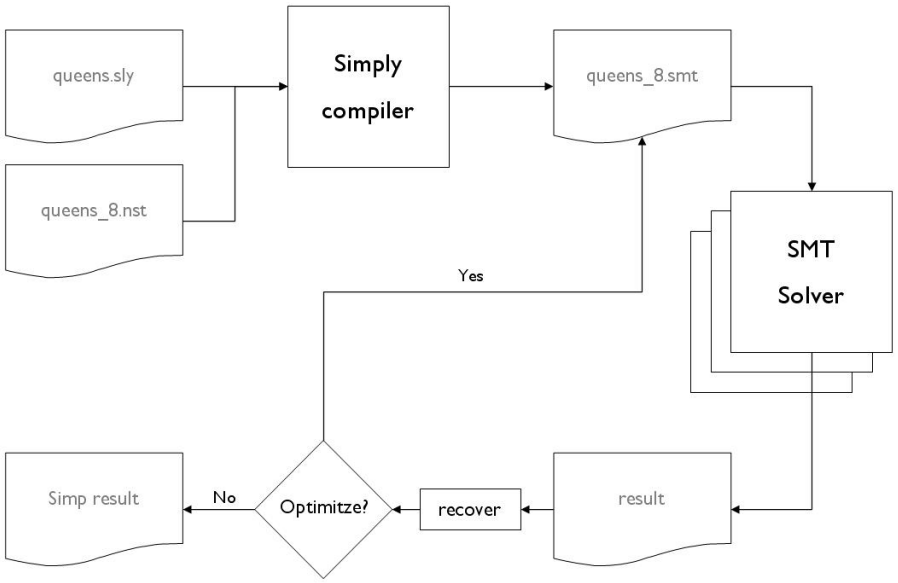
A la taula següent podem veure el temps de generació del fitxer SMT i de resolució mitjançant el SMT-Solver Yices, per a diverses instàncies del *Social Golfer*.

N Setmanes	N Grups	Mida Grup	Temps generació SMT	Temps resolució (Yices)
3	2	6	0.01	0.01
3	3	3	0.01	0.1
3	6	6	2.5	4
6	4	4	2.3	time out[1]
6	5	5	16.1	5
6	6	4	14.5	6

Tenint en compte que les modelitzacions d'aquests problemes no van ser gens sofisticades, fent trencament de simetries com a molt, els resultats obtinguts van ser força esperançadors.

3.2 `Simply`: el sistema en general

Un cop comprovat que SMT pot ser útil per resoldre aquest tipus de problemes, el següent pas era dissenyar l'arquitectura del sistema. Aquesta es pot veure a la imatge següent, on es veu el procés que hauria de seguir la resolució d'un problema: des de l'especificació en llenguatge `Simply` fins a recuperar la resposta del resoledor.



Per explicar l'esquema anterior i així veure tot el procés que hauria de seguir el sistema utilitzarem el problema de les `n` reines. Primer de tot haurem d'escollir el problema a resoldre, en el nostre cas el de les reines, on la modelització emprada és molt senzilla (sense trencament de simetries, etc). En podem veure el codi a continuació:

```
Problem:queens.sly
    Data
```

```
    int n=8;
Domains
    Dom rows=[1..n];
Variables
    IntVar q[n]::rows;
Constraints
    AllDifferent([q[i]|i in [1..n]]);
    Forall(i in [1..n-1]) {
        Forall(j in [i+1..n]) {
            q[i]-q[j]<>j-i;
            q[j]-q[i]<>j-i;
        }
    }
```

En aquest cas estem treballant amb el problema per a $n = 8$ reines. Tenim una taula `q` de `n` variables enteres on `q[i]`, per `i` de `1` a `n`, representa la fila on està situada la reina de la columna `i`. El problema es pot solucionar si existeix una assignació per `q`, d'acord amb el seu domini, tal que es satisfacin totes les restriccions, concretament, no poder haver-hi dues reines a la mateixa fila (tots els valors de `q` han de ser diferents) i no poder haver-hi dues reines a la mateixa diagonal (les distàncies entre els índexs i els valors, que representen respectivament la columna i la fila on hi ha cadascuna de les reines, de qualsevol parella de variables de la taula `q` han de ser diferents).

Aquesta modelització la guardarem a un fitxer, per exemple *queens.sly*. Per tal de separar la modelització dels problemes de les instàncies concretes, a part del fitxer de modelització es pot tenir un fitxer d'instància. En l'exemple podríem tenir un fitxer *queens_4.nst* on s'imposés que `n=4;` i en compilar es tindria en consideració els valors del fitxer d'instància en lloc dels de modelització. Tot seguit s'haurà de compilar per obtenir una instància del problema però traduït a codi SMT. El fitxer generat quedaria per exemple de la següent manera:

```
(benchmark queens_8.smt
  :source {Generated by Simpl.y, ima.udg.edu (ESLiP)}
  :category { testing }
  :logic QF_LIA
  :extrafuns ((q_1 Int) ..... (q_8 Int))
  :formula
 (and
  (and
   (and (>= q_1 1) (<= q_1 8))
    .....
   (and (>= q_8 1) (<= q_8 8))
  )
  (and
   (distinct  q_1 ..... q_8)
   (and
     (and (distinct (- q_1 q_2) (- 2 1)) (distinct (- q_2 q_1) (- 2 1))
          .....
          (distinct (- q_1 q_8) (- 8 1)) (distinct (- q_8 q_1) (- 8 1))
     )
     (and (distinct (- q_2 q_3) (- 3 2)) (distinct (- q_3 q_2) (- 3 2))
          .....
          (distinct (- q_2 q_8) (- 8 2)) (distinct (- q_8 q_2) (- 8 2))
     )
```

```
      .....
     (and (distinct (- q_7 q_8) (- 8 7)) (distinct (- q_8 q_7) (- 8 7))
     )
   )
  )
 )
)
```

Un cop tenim la instància del problema en SMT ja es pot mirar si té solució amb un resoledor que suporti la lògica `QF_LIA`. Cada resoledor dóna la resposta en un format diferent, però com a mínim tots i cada un dels resoledors ens diran si el problema és satisfactible (*sat*) o insatisfactible (*unsat*), ja que els SMT-solvers per QF_LIA són complets. Per exemple, el resoledor Yices retorna el resultat següent quan li introduïm la instància anterior:

```
sat
(= q_1 5)(= q_2 2)(= q_3 8)(= q_4 1)(= q_5 4)(= q_6 7)(= q_7 3)(= q_8 6)
```

En aquest exemple es pot interpretar fàcilment el resultat que retorna directament el resoledor. Això és degut a que la codificació de les variables és molt intuïtiva: `q_1` és el primer element de la taula `q`, `q_2` és el segon element de la taula `q` ... De totes maneres s'ha desenvolupat un mòdul que recupera el resultat del resoledor emprat per mostrar-lo i per quan es fa optimització. Aquesta tasca podria simplificar-se si s'estandarditzés com donen el resultat els SMT-solvers. Actualment el mòdul de recuperació només suporta el resoledor Yices.

En el cas de ser un problema d'optimització, primer es mirarà si l'instància del problema té solució. Si en té, un cop trobada la primera solució, s'utilitzarà el valor trobat de la variable a optimitzar i el seu domini per anar restringint, amb una variació de la cerca dicotòmica, el rang dels valors que pot prendre la variable en les següents execucions.

3.3 El llenguatge

Un cop vist el funcionament global del sistema podem començar a definir el llenguatge d'entrada. Primer analitzarem per sobre les característiques importants que ens interessa que tingui, per després definir-ne la gramàtica i la semàntica.

3.3.1 Característiques bàsiques del llenguatge

Tenint en compte que volem que el llenguatge tingui facilitats declaratives, els requisits que ha de complir són els següents:

- Per simplificar l'especificació del problema es vol diferenciar les diferents parts que formen un CSP: **definició de dominis, definició de variables i definició de restriccions**.
- S'han de poder definir **constants**, que seran utilitzades al llarg del programa, i poder-se definir en un **fitxer d'instància**, per separar la modelització del problema de les instàncies.
- Les variables que considerarem seran de tipus **Booleà** o **Enter**, o bé taules multidimensionals d'aquests dos tipus. En el fitxer d'instància també s'haurien de poder incloure inicialitzacions de variables.

- Donat que el llenguatge ha de resoldre problemes de satisfacció de restriccions, cal poder definir **restriccions bàsiques**: =, $\neq$, $\leq$, $\geq$, *and*, *or*, *xor*, *implies* i *iif*.
- A part de les restriccions bàsiques ha de tenir **restriccions globals**.

  Per exemple, l'*AllDifferent(Llista)* que obliga que tots els elements d'una llista han de ser diferents.
- També és necessari tenir **estructures de control** que permetin per exemple definir fàcilment restriccions similars. Les dues imprescindibles són les següents:
  - `Forall`: una sentència repetitiva per poder tractar les taules de variables i per definir restriccions similars.
  - *If-Then-Else*: una sentència per poder filtrar quan s'ha de generar un conjunt de restriccions.
- Una eina molt útil i que enriquiria molt el llenguatge són les **llistes per comprensió**.
- Permetre a l'usuari definir les seves pròpies restriccions (retornen una fórmula) i funcions (retornen un booleà o un enter) per simplificar el modelat i la reutilització de codi.

  Per exemple, en el problema anterior de les reines, enlloc de posar: `q[i]-q[j]<>j-i; q[j]-q[i]<>j-i;`, posar `no_atac_diagonal(q,i,j);`, què prèviament l'usuari ja hauria definit.
- També ha de suportar **optimització**, permetent indicar que es vol minimitzar o maximitzar el valor d'una variable.

3.3.2 Descripció del llenguatge

Un cop vistes les característiques del llenguatge, ara veurem la descripció de la seva gramàtica (podem trobar-la tota a l'Annex I, Capítol 6) i la semàntica del llenguatge. Cal dir que per facilitar l'explicació de la memòria s'han fet petites variacions de la gramàtica respecte el parser implementat. Per exemple, el parser no permet definir regles de la gramàtica amb * o $^+$.

El llenguatge constarà de cinc parts diferenciades:

- Definició de constants: `Data`.
- Definició de dominis: `Domains`.
- Definició de variables: `Variables`.
- Definició de nous predicats i funcions: `Functions`.
- Definició de restriccions: `Constraints`.
- Definició opcionalment d'optimització: `Minimize/Maximize`.

$$\langle simply\_program \rangle \quad ::= \quad \texttt{Program:} \langle id \rangle \ \langle data \rangle \ \langle domains \rangle \ \langle variables \rangle \ \langle user\_defined\_functions \rangle \ \langle constraints \rangle \ \langle optimitzation \rangle ?$$

Anem a analitzar cadascuna de les parts del llenguatge. La primera part és la definició de constants que s'empraran al llarg del programa. La definició de constant comença amb el token `Data` i tot seguit

les definicions de constants. Una definició de constant comença amb un identificador per a la constant (el nom de la constant que estem definint), seguit de =, el valor que es vulgui donar a la constant i com a final d'expressió un punt i coma `;`. Un identificador és una seqüència no buida de lletres/digits que no comença per digit ni _.

El valor de les constants pot ser de tipus aritmètic (`int`), booleà (`bool`), llista d'enters o llista de booleans. Els definim com $\langle arithm\_exp \rangle$, $\langle formula \rangle$ i $\langle list \rangle$, ja que poden ser expressions aritmètiques (ex. `3+4, 3+a, etc`), expressions booleanes (ex. `b implies c, b or c, etc`) o llistes. S'ha de tenir en compte que aquestes expressions han de poder ser avaluades en temps de compilació, ja que una constant només guarda un valor no l'expressió en si, i per tant si hi intervenen altres identificadors, aquests han de ser constants ja definides prèviament al seu ús. En temps de compilació totes les aparicions d'identificadors de constants seran substituïdes pel seu valor.

$\langle data \rangle$ ::= `Data` $\langle data\_exp \rangle^*$
$\langle data\_exp \rangle$ ::= `int` $\langle id \rangle$ = $\langle arithm\_exp \rangle$ `;`
| `bool` $\langle id \rangle$ = $\langle formula \rangle$ `;`
| (`bool | int`) $\langle id \rangle$ = $\langle list \rangle$ `;`

Tot seguit vindrà la definició de dominis. La definició de dominis comença amb el token `Domains` i tot seguit les definicions de dominis. Una definició de domini comença amb el token `Dom`, seguit d'un identificador per al domini (el nom del domini que estem definint), tot seguit el token =, una definició de llista amb els valors que formaran el domini i com a final d'expressió un `;`.

La definició de les llistes s'explicarà més endavant, però és important observar que els valors d'un domini han de ser coneguts, per tant la llista que defineixi el domini haurà de ser avaluable en temps de compilació.

$\langle domains \rangle$ ::= `Domains` $\langle domain\_exp \rangle^*$
$\langle domain\_exp \rangle$ ::= `Dom` $\langle id \rangle$ = $\langle list \rangle$ `;`

Un cop definits els dominis ja podem definir les variables que s'empraran per definir restriccions i que contindran el model del problema en cas que tingui solució. La definició de variables comença amb el token `Variables` seguit de les definicions de variables. Les variables poden ser booleanes, enteres o taules d'aquests dos tipus. Les definicions de variables es diferencien a nivell sintàctic entre booleanes i enteres per dues raons. La primera és per simplificar la tasca semàntica, i la segona raó, és perquè les variables enteres se'ls ha d'assignar un domini, mentre que les variables booleanes no els cal.

Per una banda, una definició de variable entera comença amb el token `IntVar`, seguit d'un o més identificadors de variable $\langle var\_id \rangle$ (separats per `,` en cas que n'hi hagi més d'un), seguit de `::`, seguit d'un identificador de domini o bé el token `int` (per variables que no tinguin un domini restringit) i amb un `;` com a final d'expressió.

Per altre banda, una definició de variable booleana comença amb el token `BoolVar`, seguit d'un o més identificadors de variable $\langle var\_id \rangle$, separats per `,` en cas que n'hi hagi més d'un, i amb un `;` com a final d'expressió.

Per a definicions de taules, l'identificador va seguit de `[`, la mesura de les dimensions de la taula i finalment `]`. La definició de les dimensions de la taula l'aconseguim amb una o més expressions aritmètiques separades per `,` entre elles tenint en compte que comencen a partir de 1. Exemple de definició de variable multidimensional de 9×9: `IntVar s[9,9]::rows;`.

Cal recordar que durant tot el procés de compilació les variables tenen valor indeterminat. Només agafaran valors un cop s'estigui buscant la solució per al problema, que això serà un cop traduït el problema a SMT i el SMT-solver doni un valor a la seva variable.

$\langle variables \rangle$ ::= `Variables` $\langle variable\_exp \rangle^*$
$\langle variable\_exp \rangle$ ::= `IntVar` $\langle var\_id \rangle$ `(` `,` $\langle var\_id \rangle)^*$ `::` (($\langle id \rangle$ | `int`) `;`
| `BoolVar` $\langle var\_id \rangle$ `(` `,` $\langle var\_id \rangle)^*$ `;`
$\langle var\_id \rangle$ ::= $\langle id \rangle$
| $\langle id \rangle$ `[` $\langle arithm\_exp \rangle$ `(` `,` $\langle arithm\_exp \rangle)^*$ `]`

Un cop definides les variables, ja es poden fer les definicions de funcions i predicats d'usuari que utilitzarem durant modelat del problema, i que en simplifiquen la lectura. Per començar a definir-les posarem el token `Functions` seguit de les diferents funcions.

Les definicions de funcions d'usuari poden ésser de dos tipus: *definicions de restriccions* o *definicions de funcions*. La principal diferència és que les primeres serveixen per generar una fórmula en la generació de codi, mentre que les segones serveixen per generar expressions aritmètiques/booleanes que s'avaluaran en temps de compilació. Aquestes definicions simplifiquen la lectura del codi, ja que restriccions o avaluacions complexes queden encapsulades dins la crida.

Les definicions de restriccions comencen per la paraula clau `DefConstraint`, seguit d'un identificador (el nom de la restricció d'usuari), una llista de paràmetres i finalment una o més sentències entre claus ({ $\langle sentence \rangle^+$ }).

Les definicions de funcions comencen per la paraula clau `DefFunction`, seguit del tipus de la funció `(` `int` | `bool`)`, un identificador (el nom de la funció), una llista de paràmetres i finalment una sentència, que serà la que s'avaluarà en temps de compilació.

Les llistes de paràmetres formals permesos actualment són identificadors amb un tipus bàsic associat.

Tot seguit podem veure un exemple de definició de restricció per a determinar que dues reines no s'ataquin en la diagonal.

```
DefConstraint no_atac_diagonal(int t,int i,int j)
{
    t[i]-t[j]<>j-i;
    t[j]-t[i]<>j-i;
}
```

$\langle user\_defined\_functions \rangle$ ::= `Functions` $\langle user\_defined\_functions\_exp \rangle^*$
$\langle user\_defined\_functions\_exp \rangle$ ::= $\langle def\_constraint \rangle$
| $\langle def\_function \rangle$ `;`
$\langle def\_constraint \rangle$ ::= `DefConstraint` $\langle id \rangle$ $\langle dfc\_params \rangle$ { $\langle sentence \rangle^+$ }
$\langle def\_function \rangle$ ::= `DefFunction` (`int` | `bool`) $\langle id \rangle$ $\langle dfc\_params \rangle$ $\langle sentence \rangle$
$\langle dfc\_params \rangle$ ::= `(` (`int` | `bool`) $\langle id \rangle$ (`,` (`int` | `bool`) $\langle id \rangle^*$) `)`

Un cop acabades les definicions de funcions d'usuari, ja podem passar a definir les restriccions que formaran l'especificació del problema. La definició de restriccions comença amb el token `Constraints` seguit per les sentències que generaran les restriccions. Que podran ser de dos tipus: estructures de control (`statement`) o restriccions directament.

$\langle constraints \rangle$ ::= `Constraints` $\langle sentence \rangle^+$
$\langle sentence \rangle$ ::= $\langle statement \rangle$
| $\langle constraint \rangle$ `;`

S'han definit dues estructures de control:

- *If-Then-Else*: que s'encarrega de filtrar restriccions.
- `Forall`: que s'encarrega de replicar restriccions.

La sentència *If-Then-Else* comença amb el token `If` seguit d'una fórmula entre parèntesis `( )`, el token `Then` i finalment un conjunt de sentències entre claus { }. Opcionalment pot estar precedit pel token `Else` i un altre conjunt de sentències entre claus { }. S'ha de tenir en compte que la fórmula entre parèntesis ha de poder ser avaluada en temps de compilació, perquè segons si es compleix la condició o no, es generaran les restriccions de dins del `Then`, o es generaran les restriccions de dins del `Else`, sempre i quan aquest estigui definit.

Per exemple:

```
If(n>3)Then
{
    a[1,n]=0;
}
Else
{
    b[1,n]=0;
}
```

La sentència `Forall` comença amb el token `Forall` seguit d'obrir parèntesis `(`, d'un identificador, del token `in`, d'una definició de llista, d'un tancar parèntesis `)` i finalment d'un conjunt de sentències entre claus { }. També es pot fer més d'un `Forall` aniuat repetint la definició de l'identificador ($\langle id \rangle$ `in` $\langle list \rangle$) tants cops com es vulgui separat per coma `,`. L'identificador serà com una variable local de compilació, que només tindrà validesa dins les sentències de dins les claus del `Forall`, i en generació de codi es replicarà tantes vegades com valors defineixi la llista associada a la variable. Per tant, aquesta llista també haurà de poder ser avaluada en temps de compilació. Per exemple:

```
Forall(i in [1,2,3])
{
    a[i]=b[i];
}
```

En aquest exemple la variable `i` agafaria els valors de la llista `[1,2,3]` i per tant la sentència `a[i]=b[i]` es repetiria tres cops, quedant les restriccions següents: `a[1]=b[1]; a[2]=b[2]; a[3]=b[3];`

$\langle statement \rangle$ ::= $\langle if\_then\_else \rangle$
| $\langle forall \rangle$

$\langle if\_then\_else \rangle$ ::= `If (` $\langle formula \rangle$ `) Then {` $\langle sentence \rangle^+$ `}`
| `If (` $\langle formula \rangle$ `) Then {` $\langle sentence \rangle^+$ `} Else {` $\langle sentence \rangle^+$ `}`

$\langle forall \rangle$ ::= `Forall (` ($\langle id \rangle$ `in` $\langle list \rangle$)$^+$ `) {` $\langle sentence \rangle^+$ `}`

Les restriccions es poden generar de tres maneres diferents:

- Mitjançant *fórmules*: expressions de caràcter lògic.

- Mitjançant *restriccions globals*: restriccions amb una finalitat concreta sobre una llista.
- Mitjançant una *restricció especial*: `If_Then_Else`.

$$\begin{array}{lcl} \langle constraint \rangle & ::= & \langle formula \rangle \\ & | & \langle global\_constraint \rangle \\ & | & \texttt{If\_Then\_Else (}\langle formula \rangle\texttt{) \{} \langle sentence \rangle^{+}\texttt{\} \{} \langle sentence \rangle^{+}\texttt{\}} \end{array}$$

Una fórmula és una expressió de caràcter lògic i pot estar definida de diverses formes: la fórmula més bàsica són els tokens `True` i `False` que fan referència al valor booleà cert i fals. També pot ser un identificador, que farà referència a una variable prèviament definida a la secció de variables amb tipus booleà. Una fórmula també pot ser una altra fórmula entre parèntesis: `( )` o bé la negació d'una altra fórmula si davant hi apareix el token `Not`. Una fórmula també es pot generar relacionant dues expressions aritmètiques mitjançant un operador relacional de forma infix. Finalment una fórmula pot estar composada per a la unió de dues fórmules amb un operador booleà també infix.

També ens podem trobar fórmules tant com a condicions dins una sentència de control *If-Then-Else*, com a l'hora de generar una restricció. Ambdós casos hauran de ser tractats diferents. En el primer la fórmula s'avaluarà en temps de compilació, mentre que en el segon cas no es calcularà el resultat de la fórmula.

Un cas especial de fórmules són les referències a restriccions definides per l'usuari. Aquestes referències consten d'un identificador i una llista de paràmetres entre parèntesis, el tipus dels quals haurà de coincidir amb els tipus dels paràmetres de dins la funció. En la generació de codi es substituiran els identificadors passats per paràmetre pels noms dels paràmetres de dins el codi de la definició. I es generarà una fórmula a partir de les sentències de la definició.

$$\begin{array}{lcl} \langle formula \rangle & ::= & \texttt{Not} \ \langle formula \rangle \\ & | & \langle formula \rangle \ \langle bool\_operator \rangle \ \langle formula \rangle \\ & | & \langle arithm\_exp \rangle \ \langle relational\_operator \rangle \ \langle arithm\_exp \rangle \\ & | & \texttt{(} \ \langle formula \rangle \ \texttt{)} \\ & | & \langle user\_defined\_references \rangle \\ & | & \langle var\_id \rangle \\ & | & \texttt{True} \\ & | & \texttt{False} \\ \langle user\_defined\_references \rangle & ::= & \langle id \rangle \ \texttt{(} \ \langle list \rangle \ \texttt{)} \end{array}$$

Els operadors relacionals entre expressions aritmètiques poden ser:

- *Igualtat*: el representarem amb un =, i significarà que el valor de l'expressió aritmètica de la part esquerra de l'operador haurà de ser el mateix que el valor de la part dreta de l'operador, en cas d'estar definint una restricció. Així per exemple podem trobar: `a+3 = 4+b;`. Cal observar que l'ús d'aquest operador és completament diferent al dels llenguatges de programació convencionals, on a la part esquerra només hi solem tenir variables o referències a objectes.

  En cas de no estar definint una restricció, com per exemple condició d'un *If-Then-Else*, s'avaluarà a veure si la part esquerra és igual a la part dreta de l'operador i es retornarà un cert o un fals.

- *Desigualtat*: el representarem amb un <>, i significarà el contrari de l'operador anterior.

- *Menor* i *major*: els representarem amb un < i > respectivament, i significarà que la part esquerre de l'operador ha de ser menor (o major) que la part dreta de l'operador. Altre cop hem de tenir en compte que si no és una fórmula que defineixi una restricció s'avaluarà l'expressió.

- *Menor o igual* i *major o igual*: els representarem amb un =< i >= respectivament. I significaran el mateix que els operadors anteriors, però incloent la igualtat.

⟨*relational_operator*⟩ ::= = | <> | < | > | =< | >=

Els operadors booleans entre fórmules poden ser:

- `And`: en cas de que les fórmules estiguin definint una restricció, restringirà que ambdues siguin certes. En cas d'estar dins una sentència de control *If-Then-Else*, s'avaluarà com a cert si ambdues fórmules són certes.

- `Or`: només restringirà que com a mínim una de les dues haurà de ser certa o s'avaluarà a cert si com a mínim una de les dues és certa.

- `Xor`: restringirà que només pot ser certa una de les dues fórmules, o s'avaluarà a cert quan només una de les dues és certa.

- `Implies`: restringirà que si es compleix la fórmula de l'esquerra s'haurà de complir la fórmula de la dreta, però no a l'inversa. O la respectiva avaluació en temps de compilació si està dins la sentència *If-Then-Else* o dins d'una llista per comprensió.

- `Iff`: restringirà que si es compleix una de les dues fórmules l'altra també s'haurà de complir, i que si una de les dues es falsa l'altra també haurà de ser falsa. O la respectiva avaluació en temps de compilació si es pot i cal.

⟨*bool_operator*⟩ ::= `And` | `Or` | `Xor` | `Iff` | `Implies`

El llenguatge disposarà de tres restriccions globals:

- `AllDifferent`: aquest token anirà seguit d'una llista entre parèntesis `( )`. Obligarà que tots els elements de la llista siguin diferents entre ells. Per exemple: `AllDifferent([a,b,c,d]);` restringirà que els valors assignats a les variables a,b,c i d siguin diferents entre ells.

- `Sum`: aquest token anirà seguit d'una llista i una expressió aritmètica separats per una coma `,` i tot entre parèntesis `( )`. Obligarà que la suma dels elements de la llista sigui igual al valor de l'expressió aritmètica. Per exemple: `Sum([1,2,b,4],a);` restringirà que el valor de la variable *a* sigui igual a la suma de la llista.

- `Count`: aquest token anirà seguit d'una llista i dues expressions aritmètiques separades per coma `,`, i tot entre parèntesis `( )`. Obligarà que la primera expressió aritmètica, que representarà el valor a buscar dins a la llista, aparegui tants cops com determini la segona expressió aritmètica. Exemple: `Count([1,2,3,b],a,c);` restringirà que el valor de la variable *a* aparegui *c* cops dins la llista.

Cal remarcar que els elements que formen les llistes no tenen perquè ser determinats, poden ser expression aritmètiques on hi intervinguin variables. Entrarem en més detall de les definicions de llistes una mica més endavant.

⟨*global_constraint*⟩ ::= `AllDifferent (` ⟨*list*⟩ `)`
| `Sum (` ⟨*list*⟩ `,` ⟨*arithm_exp*⟩ `)`
| `Count (` ⟨*list*⟩ `,` ⟨*arithm_exp*⟩ `,` ⟨*arithm_exp*⟩ `)`

L'expressió aritmètica més bàsica que podem trobar són els numerals: `35`. Tot seguit trobaríem les variables de restriccions definides com enteres: `a`. També podem trobar expressions aritmètiques entre parèntesis: `(a-b)`. O també el valor absolut d'una expressió aritmètica amb el token `Abs` i l'expressió aritmètica entre parèntesis: `Abs(a-b)`. Finalment una expressió aritmètica pot estar formada per dues expressions aritmètiques amb un operador aritmètic infix.

Un cas especial d'expressions aritmètiques són les referències a funcions definides per l'usuari. Aquestes referències consten d'un identificador i una llista de paràmetres entre parèntesis, el tipus dels quals haurà de coincidir amb els tipus dels paràmetres de dins la funció. En temps de compilació es substituiran els identificadors passats per paràmetre pels noms dels paràmetres de dins el codi de la definició. S'avaluarà la sentència de la definició, i aquest serà el valor de l'expressió aritmètica.

⟨*arithm_exp*⟩ ::= ⟨*numeral*⟩
| ⟨*var_id*⟩
| ⟨*arithm_exp*⟩ ⟨*arithm_operator*⟩ ⟨*arithm_exp*⟩
| `(` ⟨*arithm_exp*⟩ `)`
| `Abs (` ⟨*arithm_exp*⟩ `)`
| ⟨*user_defined_references*⟩

Els operadors aritmètics són els següents:

- Suma: a+b.
- Resta: a−b.
- Producte: $a \star b$.
- Divisió entera: a `Div` b.
- Mòdul: a `Mod` b.

⟨*arithm_operator*⟩ ::= `+` | `-` | `*` | `Div` | `Mod`

Una altra part important són les llistes, que sempre estaran definides entre claudàtors `[ ]`. Hi ha dos tipus de llistes:

- Llistes explícites.
- Llistes per comprensió.

Les primeres són llistes formades per elements separats per coma `,`. Un d'aquests elements pot ser:

- Una expressió aritmètica.

- Un rang: que serà dues expressions aritmètiques separades per dos punts. Per exemple: `a..b` o bé `0..4`.

- Un cas especial d'element, que només es permet per a entrades de fitxer d'instància, és el _. S'utilitza per indicar que no es vol donar valor a una variable. Per exemple, si tenim la taula de variables enteres `s[9,9]` podríem voler inicialitzar alguns valors: `s[9,9]=[1,_,_,3,_, ...]`, de manera que només agafarien valor les variables que tinguessin un nombre, i les que tinguessin _ no se'ls hi associaria cap valor.

El segon tipus de llistes són una de les eines més potents del llenguatge. Consten de dues parts separades per una `|`.

La primera part de la llista l'anomenarem patró, i estarà definit per una expressió aritmètica. Aquest patró ens definirà com seran els elements generats per la llista.

La segona part de la llista pot estar composta per dos tipus d'expressions: *generadors* i *filtres*. Els generadors i filtres es separen entre ells mitjançant comes `,`. Un generador consisteix en un identificador seguit del token `in` i una llista. Això el que farà serà replicar el patró tants cops com elements tingui la definició de la llista situada darrere del `in`. En cas que en el patró hi aparegui l'identificador que hi ha abans del `in`, aquest serà substituït per cada valor que hi hagi a la llista. Per exemple, en la llista `[i | i in [1..5]]`, al replicar segons els valors de la llista i substituir les aparicions de `i` al patró, quedaria la llista `[1,2,3,4,5]`. Si el que trobem a la segona part de la llista és un filtre el que passarà és que es descartaran els valors generats pel patró que no compleixin el filtre. Per exemple, en la llista `[i | i in[1..5] , i<4]` després d'aplicar el generador tindríem la llista `[1,2,3,4,5]`, i en l'aplicar el filtre ens quedaria `[1,2,3]`.

Hem de tenir en compte que dins al patró de les llistes per comprensió també hi poden aparèixer identificadors de variables multidimensionals, per exemple: `[m[i,j] | i in [1..3], j in [1..3], i<>j]` donaria `[ m[1,2], m[1,3], m[2,1], m[2,3], m[3,1], m[3,2]]`.

$$
\begin{array}{lll}
\langle list \rangle & ::= & \texttt{[}\ \langle list\_element \rangle\ (\ \texttt{,}\ \langle list\_element \rangle)^*\ \texttt{]} \\
& | & \texttt{[}\ \langle arithm\_exp \rangle\ \texttt{|}\ \langle var\_restrict \rangle (\ \texttt{,}\ \langle var\_restrict \rangle)^* \texttt{]} \\
\langle list\_element \rangle & ::= & \langle arithm\_exp \rangle \\
& | & \langle range \rangle \\
\langle var\_restrict \rangle & ::= & \langle id \rangle\ \texttt{in}\ \langle list \rangle \\
& | & \langle formula \rangle \\
\langle range \rangle & ::= & \langle arithm\_exp \rangle\ \texttt{..}\ \langle arithm\_exp \rangle
\end{array}
$$

La última part del programa és la d'optimització. En aquesta part podrem definir si es vol minimitzar o maximitzar una variable del problema.

$$
\begin{array}{lll}
\langle optimitzation \rangle & ::= & \texttt{Minimize}\ \langle id \rangle\ \texttt{;} \\
& | & \texttt{Maximize}\ \langle id \rangle\ \texttt{;}
\end{array}
$$

3.4 Generació de Codi

Un cop definida la gramàtica ja podem procedir a la implementació del compilador. Es va decidir fer la implementació amb Haskell, ja que dóna molta agilitat i potència a l'hora de programar, disposa d'un parser també fet en Haskell, el Happy, i al treballar amb patrons serà ideal a l'hora de generar codi.

Com tot compilador primer de tot llegim els fitxers en llenguatge `Simply`. Un cop llegit el fitxer, es passen els tokens llegits al Happy, que els parseja i genera el que anomenem `Esquelet` que conté,

de forma estructurada, la informació del problema llegida dels fitxers (modelització i instància). Una de les parts d'aquest `Esquelet` és la part d'especificació del problema que conté la llista dels "registres d'expressió", ja siguin estructures de control, restriccions, etc. Aquests registres especifiquen amb una mena de codi intermig la informació necessaria per a la futura generació de codi: tipus de restricció de l'expressió, paràmetres, estructures de control, etc. Finalment, l'`Esquelet` es passarà a una funció `generarCodiEsquelet` per a generar el codi SMT.

Per explicar la generació de codi primer veurem quines parts composen l'objecte de transferència (`SMT_Transfer`), que es farà servir dins les funcions de generació de codi, i l'objecte resultant (`SMT_Code`), que contindrà el codi generat i la informació necessaria per recuperar el resultat cap a `Simply` o per fer optimització. Després parlarem de com serà l'estructura d'un fitxer SMT-LIB generat per `Simply`. I finalment veurem les traduccions de `Simply` a SMT de les parts més interessants.

3.4.1 SMT_Transfer i SMT_Code

El Haskell és un llenguatge funcional i per tant tot el que es necessiti durant la generació de codi s'haurà de passar com a paràmetre. Per aquest motiu s'ha definit un tipus de dades de transferència, `SMT_Transfer`, que ens permetrà passar la informació necessària durant la generació de codi. D'aquesta manera si es necessita passar un nou paràmetre durant la generació de codi no s'hauran de modificar totes les funcions, quedant així tots els paràmetres encapsulats dins d'aquest objecte.

```
data SMT_Transfer = SMT_T Code[1] Dictionary Esquelet Code[2] Int
```

- `Code`[1]: objecte que contindrà una llista de *Strings*, que correspondran al codi SMT generat finalment.
- `Dictionary`: objecte que contindrà una llista de variables de diccionari. Una variable de diccionari conté la informació de la relació entre les variables `Simply` i les variables SMT:
  - Nom de la variable al `Simply`.
  - Nom del domini de la variable al `Simply`.
  - Dimensions de la variable en `Simply`, en cas de ser una variable multidimensional.
  - Tipus de la variable en SMT: bool o int.
  - Llista de variables de SMT que representen la variable `Simply`. En el cas de ser unidimensional només n'hi haurà un.
- `Esquelet`: objecte que conté la informació del problema llegida dels fitxers (modelització i instància en codi intermig) després del parsing amb el Happy.
- `Code`[2]: objecte que contindrà una llista de *Strings*, que correspondran al codi SMT auxiliar generat.
- `Int`: enter que ens dirà el nombre de tabuladors a posar al nivell actual de generació de codi.

L'objecte resultat de la generació de codi és el `SMT_Code`.

```
data SMT_Code = SMT Code Dictionary Optimitzation
```

```
(benchmark file.smt
  :source {Generated by Simpl.y, ima.udg.edu (ESLiP)}
  :category { testing }
  :logic QF_LIA
  :extrapreds ((p1) (p2) ...)
  :extrafuns ((var_int_1 Int) (var_int_2 Int) ...)
  :formula (and ... )
)
```

Figura 3.1: Estructura fitxer SMT-LIB 1.2

- `Code`: objecte que contindrà una llista de *Strings*, que correspondran al codi SMT generat, tant l'auxiliar com el generat per les restriccions.

- `Dictionary`: objecte que contindrà una llista de variables de diccionari (explicades a l'objecte `SMT_Transfer`).

- `Optimitzation`: objecte que contindrà la informació que vindrà del parsing sobre quina variable s'ha de minimitzar o maximitzar.

3.4.2 Fitxer SMT-LIB

L'estructura dels fitxers que generarem, per a ser resolts mitjançant un SMT-Solver, serà complint el llenguatge de SMT-LIB 1.2[2], i és la que podem veure a l'exemple de la figura 3.1. Anem a veure les parts que el formen:

- El `:source` no influeix en l'execució del SMT-solver, però és per indicar que el benchmark ha estat generat amb el sistema `Simply`.

- El `:category` tampoc influeix en l'execució del SMT-solver, però serveix per diferenciar els diferents tipus de benchmarks.

- El `:logic` si que és un paràmetre obligatori que indica en quina lògica treballarà el T-solver.

- El `:extrapreds` ens serveix per indicar al SMT-solver quines seran les variables booleanes.

- El `:extrafuns` és l'anàleg a l'anterior però per a les variables enteres.

- El `:formula` és la part més important, ja que aquí s'hi posarà el codi de la fórmula booleana amb àtoms de la teoria definida a `:logic`. Per tant, tota la traducció de restriccions es trobarà en aquest apartat del fitxer SMT. S'ha de tenir en compte que a part de les restriccions de l'apartat `Constraints`, també hi apareixeran més restriccions, com per exemple les generades pels dominis de les variables enteres.

[2]Recentment ha sortit el SMT-LIB2

3.4.3 De `Simply` a SMT-LIB

Un cop vista l'estructura d'un fitxer SMT, ja podem veure com es tradueix de `Simply` a SMT. Per explicar la traducció del llenguatge de `Simply` a SMT continuarem amb l'exemple de les reines de la figura 1.1. Per començar comentarem l'estructura d'un programa amb `Simply`, que podem veure'n l'encapçalament tot seguit, per tal de veure les parts que afectaran més a la generació.

```
Problem:queens.sly
    Data
        int n=8;
    Domains
        Dom rows=[1..n];
    Variables
        IntVar q[n]::rows;
    Functions
        DefConstraint no_atac_diagonal(int t, int i,int j)
        {
            t[i]-t[j]<>j-i;
            t[j]-t[i]<>j-i;
        }
    Constraints
        AllDifferent([q[i]|i in [1..n]]);
        Forall(i in [1..n-1]) {
            Forall(j in [i+1..n]) {
                no_atac_diagonal(q,i,j);
            }
        }
```

La primera part d'un programa en `Simply` és la definició de constants (`Data`). Les constants no es tradueixen cap a SMT, quan apareix el nom de la constant durant la compilació es substitueix pel seu valor, per tant no en farem res en la generació de codi.

La segona part és la definició de dominis (`Domains`). Tot i que no es tradueixin directament a SMT veurem que s'empren a l'hora de la generar el codi de les variables.

La tercera part del programa és la definició de variables. Bàsicament haurem de generar les definicions i les restriccions segons el seu domini.

La quarta part del programa és la definició de funcions d'usuari, on definirem les funcions o restriccions per a simplificar la modelització del problema.

La cinquena part del programa són les restriccions (`Constraints`). Aquesta serà la part més extensa de la generació de codi.

L'última part del programa és la part de optimització. El fet que hi hagi una variable a minimitzar (o maximitzar[3]) no afecta a la generació de codi pràcticament. Afectarà després de trobar la primera solució del problema. Hi entrarem en detall a la secció 3.5.

A la figura 3.2 podem veure la funció principal de la generació de codi, on tenim un paràmetre d'entrada de tipus `Esquelet` (generat pel parser), i retornem un objecte `SMT_Code` (tot i que el posem encapsulat en una mònada `IO`.

A alt nivell seguim els següents passos:

1. Crear un objecte de transferència inicial per utilitzar durant la generació de codi: `smtTransf`.

[3]Maximitzar és l'invers de minimitzar la variable canviada de signe.

```
generarCodiEsquelet :: Esquelet -> IO (SMT_Code)
generarCodiEsquelet esquelet
  = return smtCode
    where
      smtTransf =
        createSMT_T nouCode nouDictionary esquelet nouCode 2
      smtTransfVar = generateSMTVars smtTransf
      smtTransfCons = generateConstraints smtTransfVar
      smtCodeFull = generateDomainConstraints smtTransfCons
      constrCode = getCodeArray (getCodeST smtCodeFull)
      auxCodeTemp = getCodeArray (getAuxCodeST smtCodeFull)
      auxCode = map (put1Ident . put1Ident) auxCodeTemp
      varDefCode = generateVarDefs smtCodeFull
      finalCode = setCodeArray nouCode ([capcalera, "\n"]++
        varDefCode++["\n","\t:formula (and\n"]++
        auxCode++constrCode)
      finalDict = getDictionaryST smtCodeFull
      finalOpt = getOpt esquelet
      smtCode = SMT finalCode finalDict finalOpt
```

Figura 3.2: Funció principal de la generació de codi.

2. Cridar la funció `generateSMTVars` que ens generarà el diccionari de variables inicial.

3. Cridar la funció `generateConstraints` que generarà el codi SMT de les restriccions (simples, globals, etc). Si s'escau es crearan variables auxiliars i codi auxiliar.

4. Cridar la funció `generateDomainConstraints` que generarà el codi SMT que restringirà els dominis de les variables.

5. Recuperar el codi SMT generat a `constrCode` per al pas 9 concatenar-lo amb la resta de codi SMT.

6. Recuperar el codi SMT auxiliar generat a `auxCodeTemp`.

7. Afegir un parell de tabuladors davant de cada línia de codi auxiliar, perquè quedi més ben identat al fitxer SMT.

8. Generar les definicions de variables.

9. Finalment concatenem tot el codi dins un nou `Code` on hi haurà tot el codi que formarà el fitxer SMT, excepte els dos últims tancar parèntesis. Això és perquè en cas d'haver-hi optimització haurem d'afegir restriccions a la variable a optimitzar.

10. Creem l'objecte resultat: `SMT_Code`, amb el codi generat a 9, el diccionari i optimització.

Primer veurem la traducció de la definició de variables i les restriccions generades pels seus dominis (en el cas que en tinguin). Tot seguit explicarem la traducció de les restriccions globals i finalment de les llistes per comprensió.

A continuació veurem primer com es tradueixen les variables: la generació de definicions i les restriccions de les variables enteres. Seguit de l'explicació de la traducció de restriccions simples, restriccions globals i finalment les llistes per comprensió.

Traducció de variables

La traducció de les definicions de variables a SMT és molt senzilla degut a que els dos tipus suportats per `Simply` els suporta directament SMT: enters i booleans. Les definicions de variables enteres en SMT es representen com `:extrafuns(...)`, on per cada variable `Simply` crearem una variable SMT. En canvi les variables de taules `Simply` les despleguem i creem una nova variable SMT per cada posició.

Per exemple, seguint amb les n-reines, a partir de `IntVar q[n]::rows;` es traduirà amb n variables: `q_1 ... q_n`; quedant de la següent forma: `:extrafuns ((q_1 Int)...(q_n Int))`.

La traducció de les definicions de variables booleanes és igual que en les variables enteres, però amb la diferència que en SMT les variables booleanes es representen com `:extrapreds(...)`.

Una altra diferència entre les variables enteres i booleanes és que les primeres poden tenir un domini associat. En traduir a SMT, això es veu reflectit generant un codi auxiliar que restringirà els possibles valors de la variable entera.

Per exemple, seguint amb les n-reines, a partir de `Dom rows=[1..n];` i `IntVar q[n]::rows;` també es generarà el següent codi auxiliar:

```
(and
  (>= q_1 1) (<= q_1 n)
  ...
  (>= q_n 1) (<= q_n n)
)
```

Podem observar que s'ha desplegat la taula `q` en n variables, i per cada una s'han acotat els seus possibles valors segons el domini `(>= q_1 1) (<= q_1 n)...`.

Durant la generació de codi quan trobem una referència a una taula de variables dins d'una restricció simple, com per exemple `taula[`*exp*`] > 7`, es traduirà d'una de les següents maneres segons el tipus d'expressió de l'índex:

- Si l'expressió només conté constants i nombres (per exemple, $exp = i + 3$ on $i = 2$), és avaluable en temps de compilació i per tant es genera el codi: `(> taula_5 7)`.
- Si l'expressió conté altres variables de restricció (per exemple, $exp = n + 1$), no és avaluable en temps de compilació i es duen a terme els següents passos:
  - Crear una variable auxiliar: `aux_var`.
  - Crear un codi auxiliar per les m posicions de `taula`:

    ```
    (implies (and (= (+ n 1) 1)) (= aux_var taula_1))
      ...
    (implies (and (= (+ n 1) m)) (= aux_var taula_m))
    ```

  - substituir a la restricció l'expressió per la variable auxiliar: `(> aux_var 7)`.

Aquest últim cas on l'índex de la taula està format per una variable de restricció s'ha provat de codificar de dues maneres més, però al final s'han descartat perquè donaven mal resultats.

- La primera versió que s'ha provat ha estat mitjançant l'operador de SMT `ite`. En aquesta codificació es substitueix l'aparició de `taula[n+1]` per un `ite`, que retornarà la variable `taula_1` si es compleix que `n+1 = 1` o, en cas que no es compleixi la condició, un `ite`

aniuat que acabarà retornant una variable `taula_i` quan es compleixi la condició `n+1 = i`. Podem veure un exemple de codi SMT resultant a continuació.

```
...
(>
  (ite (and (= (+ n 1) 1))
    (taula_1)
    (ite (and (= (+ n 1) 2))
        (taula_2)
        (ite ...
        )
    )
  )
  7
)
...
```

Aquesta codificació té l'avantatge que no hem de definir variables auxiliars, però té el desavantatge que és molt ineficient a mesura que tenim més dimensions o quan l'expressió està formada per una referència a una segona taula que alhora també té variables de restricció, com per exemple `taula1[taula2[variable]] > 7`.

- La segona versió que s'ha provat ha estat mitjançant l'ús de la teoria de les funcions no interpretades amb igualtat `QF_EUF`, que ens garanteix que `a(i) = a(j)` sempre que `i=j`. En aquesta codificació es substitueix l'aparició de `taula[n+1]` per `taula(+ n 1)`, on `taula` s'haurà definit com una funció que rep un enter i retorna un enter. Podem veure un exemple de codi SMT resultant a continuació.

```
:extrafuns( (taula Int Int))
...
(> taula(+ n 1) 7)
...
```

Aquesta codificació té l'avantatge que no cal desplegar la taula creant m variables, però té el desavantatge que va més lent al buscar la solució.

A part d'aquestes tres codificacions provades, un podria pensar en utilitzar la teoria dels arrays per a codificar les taules de variables. Aquesta teoria disposa d'operacions de lectura i escriptura, i en general està pensada per al modelat de canvis d'estat en programes amb arrays. Però, com que en CP no hi ha pròpiament noció d'estat, en faríem prou amb la teoria de les funcions no interpretades. Per aquesta raó no s'ha provat d'implementar amb aquesta teoria.

Restriccions simples

La funció `generate1Constraint` s'encarrega de fer la traducció de restriccions, tant les simples com les globals. La funció rep per paràmetre un objecte de transferència i un registre d'expressió i retorna un objecte de transferència.

```
generate1Constraint :: SMT_Transfer -> ExprReg -> SMT_Transfer
```

La traducció de les restriccions simples ara mateix és directa en la majoria d'operadors, on l'única diferència és que passem de notació infixa a prefixa.

Per exemple, seguint amb les n-reines, a partir de `q[1]-q[2]<>2-1;` la traducció a SMT-LIB seria: `(distinct (- q_1 q_2) (- 2 1))`.

En aquest exemple només hi figura l'operador relacional de desigualtat, però el canvi és aplicable per a tots els operadors relacionals i operadors booleans. Podem observar que els operadors aritmètics també passen a una notació prefixa.

Una traducció més interessant la trobem amb el `Div` i el `Mod`, ja que treballem amb lògiques que no disposen d'aquests operadors, i per tant és necessari generar codi i variables auxiliars. Per aquest motiu quan ens trobem amb la següent operació aritmètica: `MOD exp1 exp2` (on `exp1` i `exp2` són expressions aritmètiques), seguirem les següents passes per a traduir-la a SMT:

- Generar el codi de `exp1` i `exp2`: `cExp1` i `cExp2` respectivament.
- Crear una variable nova `div_n'`.
- Crear una variable `mod_n'`.
- Afegir les dues variables al diccionari.
- Generar el codi auxiliar `cExp1 = (div_n' * cExp2)+ mod_n'` i $0 \leq$ `mod_n'` $\leq$ `cExp2`, en format SMT.
- Retornem com a generació de codi: `mod_n'`.

Restriccions globals

Actualment el `Simply` només suporta tres restriccions globals: *AllDifferent*, *Sum* i *Count*. Les traduccions cap a SMT són senzilles i sense cap filigrana.

- **AllDifferent**:

  En la restricció global `AllDifferent (` $\langle list \rangle$ `)` necessitem una `llista` d'elements. El primer que fem és guardar el codi generat fins al moment a `old_cs`. Per després generar els elements de la llista cridant la funció `exprregList2StringList`, que transforma una llista de registres d'expressió a la seva representació en format *String*, amb un objecte `Code` buit. Aquestes elements poden ser variables (uni/multidimensionals) o valors enters i poden estar generats per rangs, llistes per comprensió o bé perquè s'han escrit directament.

  Un cop tenim la llista d'elements, recuperem la representació dels elements de la llista de dins l'objecte de transferència, i només resta fer un `distinct` (operador de SMT-LIB que fa que els `n` elements siguin diferents) de tots ells concatenats. Això ho podem veure en el codi següent on concatenem tots els elements de la llista amb la funció `concat` més un espai entre mig dels elements mitjançant la funció d'ordre superior i la lambda-expressió, quedant la següent expressió Haskell: `concat (map (\x -> " "++x) ll)`. Finalment només hem de posar el `distinct` a davant, i guardar a l'objecte de transferència de retorn el codi generat `newCode` com a capçalera de la llista de codi vella `old_cs`.

  Si la llista d'elements és buida, o bé només hi ha un element, no cal generar codi.

```
generate1Constraint smtTransf (ALLDIF defLs)
  = smtCode
  where
    old_cs = getCodeArray (getCodeST smtTransf)
    smtCodeTemp =
      exprregList2StringList (setCodeST smtTransf nouCode) defLs
    elements = getCodeArray (getCodeST smtCodeTemp)
    nElements = length elements
    codeTemp = if (nElements > 1) then ("(distinct"++
      (concat (map (\x -> " "++x) elements)) ++ ")") else ""
    newCode = Co (codeTemp:old_cs)
    smtCode = setCodeST smtCodeTemp newCode
```

- **Sum**:

  En la restricció global `Sum(` $\langle list \rangle$ `,` $\langle arithm\_exp \rangle$ `)` necessitem una `llista` d'elements i una expressió aritmètica. Per generar el codi el primer que fem és, igual que en el cas anterior, primer guardar el codi generat fins al moment a `old_cs`. Per tot seguit generar les representacions dels elements de la llista (els elements que haurem de sumar), en format *String*, amb la funció `exprregList2StringList` i de l'expressió aritmètica a la qual igualarem el valor de la suma. També en aquest cas els elements es poden tractar de variables (uni/multidimensionals) o valors enters i poden estar generats per rangs, llistes per comprensió o bé perquè s'han escrit directament.

  En aquest cas, emprant la mateixa estructura que en cas anterior, generarem la suma de tots els elements, per tant en lloc de posar un `distinct` posarem un + (ho podem veure a la part `codeTempSuma` del codi següent). Tot seguit només ens manca restringir que l'expressió aritmètica passada per paràmetre sigui igual a la suma d'elements que acabem de generar. En el cas que la llista d'elements només contingui un element, no cal fer cap suma, i per tant igualem directament l'element amb el valor. Finalment guardem a l'objecte de transferència de retorn el codi generat `newCode` com a capçalera de la llista de codi vella `old_cs`.

```
generate1Constraint smtTransf (SUM defLs valor)
  = smtCode
  where
    old_cs = getCodeArray (getCodeST smtTransf)
    smtCodeTemp =
      exprregList2StringList (setCodeST smtTransf nouCode) defLs
    smtCodeTemp2 = generate1Constraint smtCodeTemp valor
    codeTempValor:elements =
      getCodeArray (getCodeST smtCodeTemp2)
    nElements = length elements
    isEmpty = nElements > 0
    codeTempSuma = if isEmpty then ( "(+ "++
      (concat (map (\x -> " "++x) elements))++")" ) else ""
    codeTempSuma2 = if (nElements == 1) then
        (head elements) else codeTempSuma
    codeTemp = if (nElements > 0) then ( "(= "++
      codeTempValor++" "++codeTempSuma++" )" ) else ""
    newCode = Co (codeTemp:old_cs)
    smtCode = setCodeST smtCodeTemp2 newCode
```

- **Count**:

  En la restricció global `Count(` $\langle list \rangle$ `,` $\langle arithm\_exp \rangle$ `,` $\langle arithm\_exp \rangle$ `)` necessitem una `llista` d'elements i dues expressions aritmètiques. Per generar el codi el primer que fem és, igual que en els casos anteriors, generar la representació dels elements de la llista i del

valor a buscar i del nombre de cops que ha d'aparèixer, en format `String`, amb la funció `exprregList2StringList` que ens retornarà una llista amb els elements els valors dels quals haurem de sumar en format `String`, i `generate1Constraint` per al valor i el nombre. També en aquest cas els elements es poden tractar de variables (uni/multidimensionals) o valors enters i poden estar generats per rangs, llistes per comprensió o bé perquè s'han escrit directament.

Un cop tenim la llista dels elements, el valor i el nombre de cops que volem igual que hagi d'aparèixer el valor a la llista, generarem una nova llista: una comparació per cadascun dels elements de la llista generada respecte l'expressió aritmètica a comptar. Un cop tenim aquesta llista de comparacions podem procedir a generar el codi que comptarà quants cops apareix l'expressió. Això ho fem generant una nova llista afegint l'operador de SMT-LIB `ite` *`condició valor1 valor2`*, que en cas de complir-se la *condició* retorna el *valor1* altrament retorna el *valor2*. Cal recordar que l'avaluació de la condició s'efectuarà en temps de resolució pel SMT-solver, ja que pot contenir, i en la majoria de casos contindrà, variables dins de la condició. Generarem una llista de *ites* que ens retornaran 0 o 1 segons si es compleix la condició. Tot seguit generarem la suma de tots els *ites*, i el resultat el restringirem a ser igual que la segona expressió aritmètica.

Finalment guardem a l'objecte de transferència de retorn el codi generat `newCode` com a capçalera de la llista de codi vella `old_cs`.

```
generate1Constraint smtTransf (COUNT defLs valor nombre)
  = smtCode
  where
    old_cs = getCodeArray (getCodeST smtTransf)
    tabs = getTabST smtTransf
    ident=generateIdentation (tabs+1)
    esquelet = getEsqueletST smtTransf
    smtCodeTemp =
      exprregList2StringList (setCodeST smtTransf nouCode) defLs
    smtCodeTemp2 = generate1Constraint smtCodeTemp valor
    smtCodeTemp3 = generate1Constraint smtCodeTemp2 nombre
    codeTempNombre:codeTempValor:elements
      = getCodeArray (getCodeST smtCodeTemp3)
    nElements = length elements
    compOp = if (isAvalExp valor esquelet) then ("=") else ("iff")
    isEmpty = nElements > 0
    codeTempComparacions = map (\x-> "(ite ("++compOp++" "++
      codeTempValor++" "++x++" ) 1 0)") elements
    codeTempSuma =
     if isEmpty then ( "(+ "++
      (concat (map (\x ->" "++x) codeTempComparacions))++")")
     else ""
    codeTemp = if isEmpty then ( "(= "++ codeTempNombre
      ++" "++ codeTempSuma ++" )" ) else ""
    newCode = Co (codeTemp:old_cs)
    smtCode = setCodeST smtCodeTemp3 newCode
```

Llistes per comprensió

`Simply` permet generar llistes de dues maneres: manualment introduint els nombres/variables o bé mitjançant llistes per comprensió. Ara veurem la generació d'elements de les llistes per comprensió, un altre aspecte interessant de la generació de codi. Recordem que una llista per comprensió està formada per un *patró* i una llista de *generadors* i *filtres*. Informalment podríem dir que un generador equivaldria a un per tot i un filtre a un condicional i en codi intermig quedaria codificat de la manera que explicarem a continuació:

- Un generador es codifica internament com un `IN variable llista`, o sigui, un nom de variable i una llista de valors que pot prendre.

- Un filtre es codifica internament amb un `IF condició then else`, on la condició ha de ser avaluable en temps de compilació. El *then* serà una llista que contindrà la resta de generadors/filtres de la llista per comprensió en cas que n'hi hagin sinó només contindrà el patró, i el *else* serà una llista buida.

  I finalment la llista per comprensió en codi intermig serà una llista d'expressions, els generadors, els filtres i com a últim element de la llista el patró. Per explicar el perquè el patró es posa en últim lloc i quedi més clara la generació de codi emprarem el següent exemple:

```
[x[i] | i in [1..10], i<5]
```

  Que un cop traduït a codi intermig quedaria de la forma següent:

```
[(IN i [Rang 1 10]), (IF (i<5) [ x_i ] []) ]
```

  El tractament de les llistes per comprensió dins la generació de codi el fem mitjançant la funció `exprregComprList2StringList`. Aquesta funció treballa de la següent manera: si trobem un generador el que hem de fer és canviar totes les aparicions de la variable del generador, en la resta d'expressions de llista per comprensió, pels valors de la llista associada. Això ho fem desplegant primer la llista associada a la variable mitjançant la funció `exprregList2IntList`, on els valors generats han de ser avaluables en temps de compilació. I un cop desplegats els valors generem una nova llista d'expressions on ja hi haurem substituït les aparicions de la variable.

  En l'exemple anterior després de desplegar la variable `i` amb els valors `1,2,..,10` quedaria una llista de filtres:

```
[(IF (1<5) [ x_1 ] []),
 (IF (2<5) [ x_2 ] []),
 ...,
 (IF (10<5) [ x_10 ] [])
]
```

- Un cop trobem un filtre hem d'avaluar la condició. Si aquesta es compleix tractarem les expressions del *then* i el resultat el concatenarem amb el resultat de tractar la resta d'elements de la llista. Si no es compleix la condició no farem res i continuarem tractant la resta d'elements de la llista.

  Seguint l'exemple, primer trobarem el filtre amb condició `1<5`, que és certa i per tant procedirem a tractar la part del *then*: `[x_1]` i concatenar-ne el resultat al tractament de la resta de filtres pendents de tractar: `[(IF (2<5) [x_2] []),...,(IF (10<5) x_10 ] [])]`.

- Un cop la funció troba un element que no és ni generador ni filtre, vol dir que hem arribat al patró. Com és el cas de l'exemple: `[x_1]`. En aquest cas només manca representar el valor del tipus base que correspongui a l'expressió del patró en format *string*.

Un cop generada la cadena x_1, es concatenaria al resultat de continuar tractant la resta de filtres. En aquest cas això donaria com a resultat la següent llista: `["x_1","x_2","x_3","x_4"]`.

```
exprregComprList2StringList :: SMT_Transfer ->  [ExprReg] -> SMT_Transfer

exprregComprList2StringList smtTransf [] = smtTransf

exprregComprList2StringList smtTransf ((IN (Tpb (Tps vari)) ll ):xs)
  = foldr (\x y -> exprregComprList2StringList y x) smtTransf llll
    where
      esquelet = getEsqueletST smtTransf
      lll = exprregList2IntList ll esquelet
      llll = (map (\i->(canviarVar vari i xs)) lll)

exprregComprList2StringList smtTransf ((IF cond den els):xs)
  = if res then
      (exprregComprList2StringList
        (exprregComprList2StringList smtTransf den) xs)
    else
      (exprregComprList2StringList smtTransf xs)
    where
      esquelet = getEsqueletST smtTransf
      res = avalBool cond esquelet

exprregComprList2StringList smtTransf ((Tpb t):xs)
  = (exprregComprList2StringList (valorTipusBase smtTransf t) xs)

exprregComprList2StringList smtTransf ((AO op e1 e2):xs)
  = exprregComprList2StringList (add1CodeST smtTransf r) xs
  where
    esquelet = getEsqueletST smtTransf
    x = (AO op e1 e2)
    r = show (avalExp x esquelet)
```

3.5 Optimització

Després de la generació de codi només falta solucionar el problema. Com hem dit els SMT-solvers estan només obligats a dir si el problema té solució o no, tot i que la gran majoria els hi pots demanar el model d'aquesta, que és el que ens interessarà a nosaltres. Per això s'ha fet un petit mòdul de recuperació del model, sempre que el problema en tingui, pel resoledor Yices. Aquest mòdul es pot aprofitar per a realitzar optimització.

El procés d'optimització és molt senzill, primer de tot es mira si la instància del problema té solució. Si en té i el problema està definit d'optimització, un cop trobada la primera solució del problema, s'utilitzarà el valor trobat de la variable a optimitzar i el seu domini per anar restringint, amb una variació de la cerca dicotòmica, el rang dels valors que pot prendre la variable en les següents execucions.

Per exemple, si la variable a minimitzar és `IntVar llarg::domini`, on `domini=[1..20]`, si la primera solució dóna un `15` com a solució per la variable `llarg`, a la següent iteració s'afegiran dues restriccions al fitxer SMT: una restricció de valor màxim que pot prendre la variable calculat a partir del valor obtingut (`15`) dividit per dos; i una amb el valor mínim possible que pot prendre la variable: `(and (< llarg 8) (> llarg 1))` respectivament. Tot seguit es buscarà una nova solució, si té solució (per exemple, `n`) voldrà dir que la variable `llarg` encara es pot fer més petita, i per tant tornarem a afegir una restricció al fitxer SMT, aquest cop: `(and (< llarg n/2) (> llarg 1))`. Si no té solució, voldrà dir que el valor de la variable `llarg` està entre `8` i `15`, i per tant haurem de canviar les restriccions del valor de la variable que s'afegeixen al fitxer SMT, per exemple

canviar les anteriors per `(and (>= llarg 8) (< llarg 15))`. Aplicarem successivament el procés anterior fins que el valor mínim de la variable sigui igual al valor màxim.

3.6 Fitxer d'instància

Un altre aspecte important és la separació de la modelització del problema de les instàncies d'aquest. Per fer-ho, al moment de traduir el problema cap a SMT es pot llegir un segon fitxer, a part del fitxer de la modelització, on hi haurà els paràmetres que definiran una instància concreta del problema. En aquest fitxer bàsicament hi trobarem definicions de constants i inicialitzacions de variables. La gramàtica d'aquest fitxer d'entrada és la que expliquem a continuació.

Primer de tot trobarem un identificador, que tant pot fer referència al nom d'una constant com d'una variable, seguit d'un igual =. Darrera l'igual podem trobar: una expressió aritmètica avaluable; una fórmula booleana, també avaluable; o finalment una llista, els elements de la qual són els esmentats anteriorment a la definició de llistes.

$$
\begin{array}{lcl}
\langle instance \rangle & ::= & \langle instance\_exp \rangle^* \\
\langle instance\_exp \rangle & ::= & \langle id \rangle \texttt{ = } \langle arithm\_exp \rangle \texttt{ ;} \\
 & | & \langle id \rangle \texttt{ = } \langle formula \rangle \texttt{ ;} \\
 & | & \langle id \rangle \texttt{ = } \langle list \rangle \texttt{ ;}
\end{array}
$$

Tot seguit tenim un exemple de fitxer d'instància per al problema del Sudoku.

```
n = 9;
s =[_,_,2,_,_,1,_,6,_,
    _,_,7,_,_,4,_,_,_,
    5,_,_,_,_,_,9,_,_,
    _,1,_,3,_,_,_,_,_,
    8,_,_,_,5,_,_,_,4,
    _,_,_,_,_,6,_,2,_,
    _,_,6,_,_,_,_,_,7,
    _,_,_,8,_,_,3,_,_,
    _,4,_,9,_,_,2,_,_];
```

En aquest cas tenim els dos casos d'identificadors possibles: `n` que fa referència a una constant i `s` que fa referència als valors d'*inicialització* de la variable de restricció. Al primer cas es donarà valor a la constant segons l'especificat al fitxer d'instància. Al segon cas, tot i que `s` es defineix al fitxer de modelització com una variable multidimensional: `IntVar s[n,n]::a;`, al fitxer d'instància no s'especifiquen dimensions, i automàticament s'agafen `n` valors de la matriu segons les dimensions de `s`. L'altra característica important és el caràcter `_` que, com ja hem comentat, indica que la restricció d'assignació, que es generaria sobre la variable de restricció, es descarti. Seguint amb l'exemple del Sudoku (`s =[_,_,2, ... ];`), les dues primeres (`s[1,1]=_;` i `s[1,2]=_;`) no es generaria cap restricció, mentre que a la tercera si que es generaria quedant `s[1,3]=2;`.

Capítol 4

Resultats

La part de resultats està dividida en les següents tres seccions, que venen determinades per l'evolució del sistema. A la primera secció hi podem veure els resultats de comparar la primera versió del sistema emprant diferents SMT-solvers amb dos dels sistemes més semblants: MiniZinc (traduint a FlatZinc i emprant el resoledor G12) i SPEC2SAT, resolent quatre problemes típics de restriccions. A la segona secció hi podem veure els resultats de comparar el sistema, respecte MiniZinc, compilant a diversos resoledors de FlatZinc. I a la tercera secció hi podem veure els resultats de comparar els temps de resolució de diferents SMT-solvers.

4.1 Comparació de `Simply` amb altres eines semblants

El primer pas per provar el sistema va consistir en agafar quatre problemes típics de CSP i modelitzar-los en `Simply` per després resoldre'ls amb diversos SMT-solvers.

4.1.1 Problemes de comparació

Els problemes escollits van ser els següents, les modelitzacions dels quals podem trobar a l'Annex III (Capítol 8). Es van escollir aquests problemes ja que semblava que oferirien un equilibri interessant entre la fórmula booleana i l'aritmètica, i que es podien modelar de forma senzilla.

- **Queens:**

  El problema de les reines es va especificar de diferents maneres. Es va escollir aquesta especificació ja que es pot especificar en QF_IDL, i per tant la seva resolució és més ràpida en principi. Aquest problema s'anava a treure ja que el guany de la teoria de fons no és gaire important, però era interessant ja que es podia modelitzar amb QF_IDL.

- **Balanced Academic Curriculum Problem (BACP):**

  L'objectiu del BACP (*CSPLib problema 030*) és dissenyar un currículum acadèmic assignant períodes a cursos de manera que la càrrega acadèmica a cada període estigui balancejada. El currículum ha de seguir les següents regulacions acadèmiques i administratives:

  1. Els cursos han d'estar assignats amb un nombre màxim de períodes (`n_periods` en la nostra modelització).
  2. Cada curs té un nombre de crèdits o unitats que representen l'esforç necessari per passar-lo (`course_load[n_courses]`).

3. Hi ha cursos que tenen com a prerequisits altres cursos.

4. Són necessaris un mínim de crèdits acadèmics per període per considerar un estudiant a temps complet, i només estan permesos un màxim de crèdits acadèmics per evitar sobrecàrregues (`load_per_period_lb, load_per_period_ub`).

5. Són necessaris un mínim de cursos per període per considerar que un estudiant està a temps complert, i només estan permesos un màxim de cursos per període per evitar sobrecàrregues (`courses_per_period_lb, courses_per_period_ub`).

L'objectiu és assignar un període a cada curs de manera que es satisfacin el mínim i el màxim de càrrega acadèmica a cada període, el mínim i el màxim de cursos per a cada període i els prerequisits entre cursos.

- **Schur's Lemma**:

  El problema Schur's Lemma (*CSPLib problema 015*) consisteix en posar n boles etiquetades de $\{1..n\}$ dins de tres caixes de manera que per qualsevol tripleta (x, y, z) amb $x + y = z$, no estan a la mateixa caixa. Aquest problema només té solució per a $n < 14$. (Observem que a la definició hi ha implícit que una bola x no pot estar a la mateixa caixa que la bola $2x$.)

- **Job-shop scheduling**:

  El problema del job-shop hi ha un conjunt de màquines on cada una realitza una operació diferent. Hi ha diferents feines a fer i una feina està composada per diferents tasques. Cada tasca requereix el processament a una màquina determinada durant un lapse de temps, i una màquina pot operar com a molt amb una tasca a la vegada. Les tasques no es poden interrompre. L'objectiu és, donat un temps límit, programar cada feina de manera que acabi el temps abans del temps límit.

4.1.2 Resultats

Després de traduir a SMT els problemes especificats amb el llenguatge de `Simply`, les instàncies s'han resolt amb els resoledors que van participar en la divisió de QF_LIA de la *Satisfiability Modulo Theories Competition* (SMT-COMP) 2008. La primera versió del sistema es va presentar al ModRef09 [BPSV09] i al Prole09 [BPV09]. Per aquesta raó es van utilitzar aquests resoledors. També es pot veure que la sintaxi dels problemes de l'Annex III (Capítol 8) és una mica diferent que l'actual, perquè el llenguatge ha anat evolucionant (per exemple, ara és més estricte en el tipus de les constants, a les constants es permeten llistes de valors, etc).

Com a excepció, el problema de les reines s'ha codificat en QF_IDL. Per fer-ho s'ha hagut de modificar manualment les instàncies de fitxers SMT, ja que actualment a la capçalera es posa per defecte en QF_LIA.

A part també s'han executat sobre MiniZinc 0.9 i SPEC2SAT 1.1 els mateixos problemes, sempre mirant de mantenir el màxim possible la mateixa modelització que amb els problemes SMT i evitant d'emprar cap estratègia de resolució.

	Simply	Z3.2	Mathsat-4.2	Cvc3-1.5	Bclt 1.3	Yices 1.0.10	Minizinc(G12)	SPEC2SAT
Queens_50	0.22	t.o.	53.22	m.o.	11.94	29.69	**0.22**	248.01
Queens_100	0.72	t.o.	t.o.	m.o.	389.76	19.94	**0.84**	t.o.
Queens_150	1.54	t.o.	t.o.	m.o.	997.48	t.o.	**150.4**	t.o.
Bacp_12_6	0.17	0.72	2.7	t.o.	57.15	**0.36**	0.84	m.o.
Bacp_12_7	0.18	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.
Bacp_12_8	0.22	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.
Bacp_12_9	0.21	**0.48**	11.07	t.o.	315.12	0.85	0.94	m.o.
Bacp_12_10	0.24	**0.48**	15.21	t.o.	190.34	1.03	1.44	m.o.
Bacp_12_11	0.24	**0.51**	13.84	t.o.	237.74	1.48	1.7	m.o.
Bacp_12_12	0.26	**0.74**	13.5	t.o.	338.72	1.58	40.59	m.o.
Schurl_12_3	0.06	**0.07**	0.14	7.97	0.09	0.08	t.o.	0.38
Schurl_13_3	0.08	**0.12**	0.15	14.38	0.13	0.13	t.o.	0.55
Schurl_14_3	0.09	0.32	0.59	18.33	**0.21**	0.25	t.o.	0.5
Schurl_15_3	0.1	0.45	0.89	29.25	**0.25**	0.28	t.o.	0.73
Jobshop_54	0.31	**0.43**	0.58	105.28	8.1	3	34.13	80.41
Jobshop_55	0.32	**0.52**	0.67	211.8	11.89	3.95	122.16	80.03
Jobshop_56	0.3	**0.42**	0.76	358.83	12.38	4.67	396.03	88.11
Jobshop_57	0.3	**0.64**	1.19	475.85	16.35	6.92	1115.09	85.66
Jobshop_58	0.34	**0.44**	0.59	135.05	21.09	11.82	0.09	95.11

La taula anterior mostra el temps en segons gastat per solucionar el problema de cada resoledor, s'ha posat un timeout de 1800 segons. Les execucions s'han fet sobre un Intel Core 2 Duo a 3.00 GHz amb 2Gb de RAM i amb SO GNU/Linux 2.6. En negreta es marca el millor resultat en temps.

La columna `Simply` fa referència al temps de compilació. Les següents cinc columnes conten el temps que han tardat a solucionar els problemes els diferents resoledors SMT (amb el temps de compilació del `Simply` inclòs). I les dues últimes columnes donen el temps que han tardat el MiniZinc i SPEC2SAT. SPEC2SAT transforma especificacions escrites en NP-SPEC a instàncies SAT en format DIMACS i pot treballar amb qualsevol resoledor SAT que suporti aquest format, en aquest sentit és semblant al nostre sistema. Els temps de la taula s'ha utilitzat el SPEC2SAT amb zChaff 2007.3.12 [MFM04], i els temps inclouen compilació i resolució.

Si analitzem la taula podem veure:

- Al problema de les *Reines* els millors resultats han estat pel MiniZinc mentre que SPEC2SAT i els resoledors SMT han fallat a resoldre la majoria de les instàncies. La poca eficiència dels resoledors SMT en aquest problema possiblement és degut al fet que la instància SMT de la traducció de la modelització en `Simply` dóna clàusules unitàries, mentre els resoledors SMT guanyen amb problemes on la satisfactibilitat depèn de la combinació entre l'estructura booleana i la teoria de fons.

- Al *BACP*, s'han obtingut resultats similars pels resoledors SMT i el MiniZinc. Tot i així, per instàncies complicades prop de la fase de transició no l'ha solucionat cap d'ells. Les instàncies SMT generades per `Simply` per aquest problema inclouen una estructura booleana barrejada amb expressions aritmètiques, principalment per les restriccions globals `Count` i el `Sum`. Per això és probable que els resoledors SMT hagin obtingut millors resultats que amb el problema de les reines. Finalment, SPEC2SAT no ha pogut resoldre cap instància, ja que la fórmula generada en SAT és massa gran per al zChaff.

- Al problema *Schur's Lemma*, els millors resultats els han obtingut els resoledors SMT i el SPEC2SAT. Sorprenentment el MiniZinc ha expirat per timeout sense donar resposta. Amb la modelització escollida, el `Simply` genera una instància SMT sense aritmètica, ja que totes les expressions es poden avaluar en temps de compilació.

- Al problema del *Job-shop*, els millors resultats els han obtingut alguns dels resoledors SMT, seguits del SPEC2SAT i finalment el MiniZinc. Altre cop les instàncies generades pel `Simply` tenen una estructura mixta booleana i aritmètica.

En general, es pot dir que els resoledors SMT escalen força bé amb la mida del problema. Gràcies a la gran expressivitat del SMT, les instàncies SMT resultants ocupen molt menys que les instàncies a SAT i, a part de les instàncies més dures, els problemes són resolts en temps raonable. Per tant SMT garanteix un bon equilibri entre expressivitat i eficiència per solucionar CSP.

4.2 Comparació de `Simply` amb MiniZinc

Un dels llenguatges més estesos per CP és el MiniZinc, que es tradueix a FlatZinc per tal de que diversos resoledors busquin sila instància del problema té solució. Per això es va trobar interessant poder comparar el `Simply` amb aquest altre sistema. En aquest apartat comentarem una mica la comparació de resultats.

4.2.1 Problemes de comparació

Per poder comparar, es van agafar modelitzacions de problemes de MiniZinc, es van programar amb `Simply` (intentant que la modelització fos el més fidel possible a l'original) i es van executar les diferents instàncies dels benchmarks de MiniZinc[1] amb `Simply`, dels quals n'hi ha que són problemes de la CSPLib, puzzles o problemes de planificació, etc.

Atès que es volia mantenir una modelització fidel a la del MiniZinc i que el `Simply` encara no té tanta expressivitat com MiniZinc només es van poder modelitzar la meitat dels benchmarks de MiniZinc; més concretament, en aquests benchmarks ens trobem amb restriccions globals com el `cumulative`, o operacions sobre llistes de l'apartat de constants com la suma dels elements, o el màxim element de la llista, o són modelitzacions que treballen amb conjunts, etc. que el `Simply` ara mateix no té implementades. Els problemes que no surten a aquesta taula també es poden modelitzar amb `Simply`, però hauria de ser amb una modelització diferent, i per tant s'han descartat.

4.2.2 Resultats

En aquest cas s'ha comparat el SMT-solver Yices2 amb els resoledors de FlatZinc: G12, ECL^{i}PS^e, Gecode i FZNTini.

A la taula següent podem veure el temps de resolució pels diferents problemes. A la primera columna trobem el nom del problema; a la segona si es tracta d'un problema de satisfactibilitat (s) o d'optimització (o); a la tercera el nombre d'instàncies del problema; de la quarta a la setena tenim els temps de resolució dels diferents resoledors de FlatZinc; finalment a la vuitena columna tenim els temps de resolució del `Simply`. En aquestes últimes cinc columnes trobem el temps de resolució del nombre d'instàncies que s'han pogut resoldre (el número entre parèntesis) sense excedir el timeout de 300 segons. En negreta es remarca el *millor resultat*, donant més prioritat al nombre d'instàncies resoltes i en cas d'empat el temps.

Com podem veure a la taula el `Simply` és el que resol més instàncies amb 110, i el segon a resoldre'n més és el G12 amb 103. Però s'ha de dir que el `Simply` tarda més a resoldre les diferents instàncies, si ens tornem a fixar amb el G12, només tarda 900 segons pels 3500 del `Simply`; el per

[1] Benchmarks que venien amb la distribució 1.0.3 de MiniZinc.

Problem		#	G12	ECL^{i}PS^e	Gecode	FZNTini	Simply
alpha	s	1	**0.01(1)**	0.47(1)	0.07(1)	0.60(1)	0.21(1)
areas	s	4	0.13(4)	2.04(4)	**0.05(4)**	0.38(4)	0.63(4)
bibd	s	14	118.16(12)	4.18(7)	35.58(7)	**353.78(13)**	26.63(11)
cars	s	79	0.02(1)	0.81(1)	295.64(3)	1.21(1)	**2420.89(19)**
curriculum	s	3	0.27(2)	95.09(2)	261.82(1)	8.70(3)	**1.26(3)**
eq	s	1	**0.01(1)**	0.49(1)	**0.01(1)**	20.14(1)	0.24(1)
golomb	o	10	**251.18(9)**	85.84(8)	23.83(8)	23.87(6)	66.05(6)
langford	s	25	52.19(20)	121.85(20)	34.54(20)	310.24(18)	**12.13(20)**
latin-squares	s	7	**5.98(6)**	12.54(6)	15.35(6)	129.40(3)	1.45(4)
magicseq	s	9	25.17(7)	21.56(7)	**7.39(7)**	0.30(3)	106.06(5)
nmseq	s	10	281.73(6)	-(0)	**171.03(7)**	-(0)	213.73(2)
photo	o	2	0.10(2)	1.07(2)	**0.04(2)**	0.08(2)	0.66(2)
quasigroup7	s	10	**1.37(5)**	293.67(3)	3.65(5)	380.28(3)	7.74(5)
queens	s	7	88.88(7)	**36.24(7)**	0.52(3)	94.76(4)	110.17(5)
schur_numbers	s	3	1.30(3)	1.03(2)	**0.30(3)**	0.02(2)	0.53(3)
shortest_path	o	10	4.36(4)	292.15(6)	**141.34(7)**	-(0)	88.65(6)
slow_convergence	s	10	68.74(10)	12.84(7)	**11.03(10)**	36.98(4)	479.71(10)
tents	s	3	0.10(3)	1.52(3)	**0.04(3)**	0.30(3)	0.68(3)
Total		208	899.7(103)	983.39(87)	1002.23(98)	1361.04(71)	**3537.42(110)**

què ho podem veure en el problema dels cars on el `Simply` resol 19 instàncies per la única que resol el G12, i és on creix el temps en 2400 segons. Quant a tipus de problemes, el `Simply` és el millor en 3 problemes de 18, mentre que el G12 ho és en 5 problemes, i el Gecode és el millor en 9 de 18 problemes.

4.2.3 Comparació amb `fzn2smt`

Durant el desenvolupament de la màster tesis, dins del grup ESLiP del qual formo part, uns companys han desenvolupat una eina que tradueix de FlatZinc a SMT, el `fzn2smt` [BSV10]. S'ha trobat interessant comparar les resolucions dels problemes traduïts de FlatZinc a SMT i de `Simply` a SMT.

A la taula següent podem veure el temps de resolució pels diferents problemes. A la primera columna trobem el nom del problema; a la segona si es tracta d'un problema de satisfactibilitat (s) o d'optimització (o); a la tercera el nombre d'instàncies del problema; a la quarta tenim els temps de resolució del `fzn2smt`; finalment a la cinquena columna tenim els temps de resolució del `Simply`. En aquestes últimes cinc columnes trobem el temps de resolució del nombre d'instàncies que s'han pogut resoldre (el número entre parèntesis) sense excedir el timeout de 300 segons. En negreta es remarca el *millor resultat*, donant més prioritat al nombre d'instàncies resoltes i en cas d'empat el temps.

Com es pot veure tant el `Simply` com el `fzn2smt` resolen casi el mateix nombre d'instàncies: 110 el `Simply`, i 111 el `fzn2smt`. On realment es pot veure diferència és en els temps de resolució que hi ha més de 700 segons de diferència a favor de `fzn2smt`; però s'ha de tenir en compte que al passar de MiniZinc a FlatZinc s'apliquen tècniques que milloren el codi FlatZinc, en canvi el `Simply` tradueix a SMT-LIB directament sense fer cap processament. En el que sembla que el `Simply` ha donat més bon resultats és en el nombre de problemes que resol millor (més instàncies o més ràpid en cas d'empat): en 12 problemes de 18 el `Simply` obté més bons resultats.

Problem		#	`fzn2smt`	`Simply`
alpha	s	1	0.66(1)	**0.21(1)**
areas	s	4	4.75(4)	**0.63(4)**
bibd	s	14	**79.48(12)**	26.63(11)
cars	s	79	**2271.17(21)**	2420.89(19)
curriculum	s	3	9.97(3)	**1.26(3)**
eq	s	1	0.47(1)	**0.24(1)**
golomb	o	10	**41.88(6)**	66.05(6)
langford	s	25	50.52(20)	**12.13(20)**
latin-squares	s	7	7.54(4)	**1.45(4)**
magicseq	s	9	11.01(4)	**106.06(5)**
nmseq	s	10	1.42(1)	**213.73(2)**
photo	o	2	0.78(2)	**0.66(2)**
quasigroup7	s	10	31.51(5)	**7.74(5)**
queens	s	7	**54.61(5)**	110.17(5)
schur_numbers	s	3	1.45(3)	**0.53(3)**
shortest_path	o	10	**45.79(6)**	88.65(6)
slow_convergence	s	10	**222.37(10)**	479.71(10)
tents	s	3	2.50(3)	**0.68(3)**
Total		208	**2837.88(111)**	3537.42(110)

4.3 `Simply` per a la comparació de diferents resoledors SMT

Finalment es va trobar interessant comparar el temps de resolució dels SMT-solvers que van obtenir més bons resultats a la SMT-Comp del 2009 en la categoria QF_LIA. Es van escollir el MathSAT, el Sateen (campió de la categoria el 2009), el Yices2proto i el Z3.2 (campió de la categoria el 2008).

4.3.1 Problemes de comparació

Els problemes emprats són els mateixos que en l'apartat anterior però traient els que eren d'optimització, degut a que els altres SMT-solvers no es té mòdul de recuperació, i aprofitant les instàncies que es tenien generades ja en format SMT-LIB.

4.3.2 Resultats

A la taula següent podem veure el temps de resolució pels diferents problemes (sense compilació). A la primera columna trobem el nom del problema; a la segona el nombre d'instàncies del problema; finalment de la tercera columna a la sisena columna tenim els temps de resolució dels diferents SMT-solvers. En aquestes últimes quatre columnes trobem el temps de resolució del nombre d'instàncies que s'han pogut resoldre (el número entre parèntesis) sense excedir el timeout de 300 segons. En negreta es remarca el *millor resultat*, donant més prioritat al nombre d'instàncies resoltes i en cas d'empat el temps.

Com podem veure a la taula el Yices2proto és el que resol més instàncies amb 85 (per aquest motiu és el solver que es utilitzar en les comparacions de l'apartat anterior), i el segon a resoldre'n més és curiosament el Z3.2 (campió de l'any anterior) amb 72. Quant a tipus de problemes, tenint en compte que hi ha 2 empats, el que en resol més torna a ser el Yices2proto amb 10 de 14, seguit del Sateen amb 4 de 14, i els altres dos són el millor en només un problema.

És interessant veure que el Sateen campió de la categoria QF_LIA a la SMT-Comp 2009 és el que resol menys instàncies en els problemes que hem provat. Això pot ser degut a que per la competició

Problem	#	MathSAT	Sateen	Yices2proto	Z3.2 (2008)
alpha	1	**0.04(1)**	-1	0.05(1)	0.05(1)
areas	4	0.14(4)	0.07(4)	**0.004(4)**	0.061(4)
bibd	14	17.31(9)	223.07(11)	**26.274(12)**	88.302(11)
cars	79	0.13(1)	0.07(1)	**1329.847(12)**	1.63(1)
eq	1	0.1(1)	**0.001(1)**	**0.001(1)**	0.02(1)
langford	25	54.37(20)	15.493(20)	**6.524(20)**	342.87(20
latin-squares	7	88.61(2)	0.001(1)	0.401(4)	**1.24(5)**
magicseq	9	2.14(2)	**29.06(4)**	93.751(4)	120.21(3)
nmseq	10	0(0)	**183.31(2)**	2.22(1)	114.1(1)
quasigroup7	10	10.17(5)	333.08(5)	**5.37(5)**	7.97(5)
queens	7	0.46(3)	228.212(3)	**95.132(5)**	120.63(4)
schur_numbers	3	0.08(3)	**0.003(3)**	**0.003(3)**	0.012(3)
slow_convergence	10	268.74(10)	3.22(3)	**14.9(10)**	38.58(10)
tents	3	34.79(3)	0.002(2)	**0.102(3)**	0.32(3)
Total	183	477.99(64)	1015.592(60)	**1574.572(85)**	835.905(72)

s'ajusten paràmetres del resoledor per obtenir bons resultats en els benchmarks dels anys anteriors, cosa que no t'assegura, com es pot veure a la taula, que si es posa un nou tipus de problema et doni bons resultats.

Capítol 5

Conclusions i treball futur

L'objectiu principal d'aquesta tesi de màster ha estat el desenvolupament de `Simply`. Una primera versió del sistema es va presentar a ModRef09 [BPSV09] i al Prole09 [BPV09]. `Simply` un sistema (i un llenguatge declaratiu alhora) per a la resolució de problemes de restriccions mitjançant la satisfactibilitat mòdul teories. Hem codificat una sèrie de problemes que ens han permès comparar el nostre sistema amb altres similars. A més a més, aquests problemes també ens han permès comparar el rendiment de diferents SMT-solvers per a problems CSP.

En la tesi de màster, primer hem explicat els conceptes més importants per entendre i situar el `Simply`: CSP, SAT i SMT. Com hem comentat, les aproximacions típiques a resolució de CSPs han estat basades en la cerca i en el control d'aquesta mitjançant estratègies per podar l'arbre de cerca o per orientar-ne la cerca. Si bé en el CSPs també s'han anat considerant tècniques de propagació i d'aprenentatge, és en el marc de SAT i SMT on aquestes guien totalment el mecanisme de resolució. Una vegada presentats els conceptes fonamentals de la tesi, hem explicat el procés seguit per al desenvolupament de `Simply` i les seves característiques més rellevants en el diagrama 3.2. També n'hem donat la gramàtica i descrit la semàntica a la Subsecció 3.3.2. Finalment hem aportat una sèrie de resultats que ens permeten valorar positivament la nostra aportació; com es pot veure a la taula 4.1.2 i a la taula 4.2.2 el fet de resoldre CSPs traduint-los al llenguatge de SMT-LIB i fer servir un SMT-solver, és una opció competitiva en molts casos. Amb la bateria de benchmarks executats es pot veure en quins problemes el `Simply` obté millors temps de resolució: bàsicament els problemes que estan situats en un equilibri entre una component lògica i restriccions d'aritmètica lineal. Aquesta conclusió empírica s'adiu amb el fet que els SMT-solvers combinen les millors tècniques dels SAT-solvers amb les millors tècniques per a les resolucions de teories (per exemple, en el cas particular de LIA, típicament els T-solvers implementen Simplex). Pel que veiem a les nostres experimentacions, els problemes de planificació, com ara `cars` o `currículum`, són d'aquesta naturalesa. Aquesta observació reforça/confirma el suggerit a [NORCR07]. Els que en canvi es troben principalment a SAT o són bàsicament una simple conjunció de restriccions d'aritmètica lineal, no semblen gaudir dels avantatges d'aquesta mena de resolució. `Simply` també ens permet fer fàcilment comparatives de rendiment entre diferents SMT-solvers per a problemes CSP. Una característica interessant d'aquesta comparativa és que s'ha fet sense que els SMT-solvers s'hagin "preparat" per a donar bons rendiments en aquesta mena de problemes. A la SMT-comp és habitual que els resoledors es configurin de manera que rendeixin especialment bé per als problemes d'anys anteriors. De fet, com s'ha vist, el guanyador de la SMT-comp 2009, no és el guanyador de la comparativa que nosaltres hem fet (vegeu taula 4.3.2).

De fet, el fet que haguem triat una separació de fitxers entre model i instància tal com fa MiniZinc i amb un format molt similar, ens permet fàcilment reutilitzar aquestes instàncies, de manera que la

col·lecció de problemes per a `Simply` podrà anar creixent ràpidament i fàcil.

Finalment, podem dir que `Simply` també és una oportunitat, un marc d'estudi que permet i suggereix força treball futur.

Quant a millores en el sistema:

- Generar codi SMT-LIB2. A SMT-LIB2 es va presentar el nou standard per als SMT-solvers.
- Millorar el codi SMT generat. Tal com es descriu a *Effective Compilation of Constraint Models* [Ren10], o es pot apreciar en el procés de compilació de MiniZinc a FlatZinc, en traduir d'un llenguatge d'alt nivell a un de més baix nivell es poden aplicar tècniques per a millorar la traducció.
- Permetre l'especificació de quin resoledor es vol que s'utilitzi per resoldre els problemes un cop es compila del llenguatge de `Simply` a SMT. Ara mateix s'executa el resoledor Yices. Això és degut a que cada resoledor retorna el model d'una manera diferent i només s'ha implementat el mòdul de recuperació per Yices. El mòdul de recuperació es simplificaria si s'estandarditzés la sortida dels resoledors SMT. En aquest sentit, a SMT-LIB2 ja hi ha elements que permeten recuperar el model de forma estàndard.
- Permetre gestionar l'optimització per part dels SMT-solvers que ho contemplin (com ara Yices 2 i MathSAT). De fet serà interessant comparar els resultats del nostre sistema d'optimització amb els resultats de l'optimització del resoledor, encara que l'optimització actual és molt senzilla i dóna marge de millora, ja que l'únic que aplica és una cerca dicotòmica sobre la variable a minimitzar/maximitzar.
- Estendre el compilador per poder treballar amb altres teories. Concretament, seria interessant poder codificar algunes restriccions en teories com *difference logic* o *bit-vectors* per tal de trobar possibles resolucions més eficients.

Quant a millores del llenguatge:

- Donar suport a restriccions sobre conjunts.
- Permetre restriccions no lineals.
- Afegir més codificacions per a restriccions globals: `cumulative`, `global-cardinality`, `knapsack`, etc.
- Millorar la traducció de les restriccions globals a SMT.
- Aprofitar la potència de les llistes per comprensió per generar restriccions.

  Per exemple, `[x[i]>x[i-1]; | i in [2..n]]`.
- Permetre especificar la lògica sobre la qual es vol treballar: QF_LIA, QF_IDL, etc. I per tant, validar si la modelització està dins la teoria, o inclús mirar de fer la traducció de manera que estigui dins la teoria, en cas que es pugui.
- Permetre que els paràmetres de les definicions d'usuari puguin ésser explícitament llistes o tipus més complexos, ja que ara mateix només es substitueix el nom de dins la definició pel nom que es passa per paràmetre.

Altres oportunitats:

- Implementar resoledors de teories per a constraints globals. A [BM10] es presenta un resoledor per a la teoria del constraint `Alldifferent` amb resultats molt interessants. És raonable pensar que disposar de resoledors de teories per a d'altres constraints globals com ara `cumulative`, `global-cardinality`, `knapsack`, etc. seria realment útil per a la resolució de problemes que els fessin servir. El sistema hauria de permetre especificar constraints "externs"per tal que la compilació no els toqués i els deixés en el format necessari per als resoledors que els poguessin resoldre (a l'estil del que es fa amb MiniZinc).
- Implementar resoledors per a la teoria dels conjunts. A part de codificacions adequades i eficients en SMT, disposar d'un resoledor especialitzat per a les restriccions de conjunt també suggereix un guany quant a eficiència.

Capítol 6

Annex I: Gramàtica completa

⟨*simply_program*⟩ ::= `Program:`⟨*id*⟩ ⟨*data*⟩ ⟨*domains*⟩ ⟨*variables*⟩
⟨*constraints*⟩ ⟨*user_defined_functions*⟩ ⟨*optimitzation*⟩?

⟨*data*⟩ ::= `Data` ⟨*data_exp*⟩*
⟨*data_exp*⟩ ::= `int` ⟨*id*⟩ = ⟨*arithm_exp*⟩ `;`
| `bool` ⟨*id*⟩ = ⟨*formula*⟩ `;`
| (`bool | int`) ⟨*id*⟩ = ⟨*list*⟩ `;`

⟨*domains*⟩ ::= `Domains` ⟨*domain_exp*⟩*
⟨*domain_exp*⟩ ::= `Dom` ⟨*id*⟩ = ⟨*list*⟩ `;`

⟨*variables*⟩ ::= `Variables` ⟨*variable_exp*⟩*
⟨*variable_exp*⟩ ::= `IntVar` ⟨*var_id*⟩ (`,` ⟨*var_id*⟩)* `::` (⟨*id*⟩ | `int`) `;`
| `BoolVar` ⟨*var_id*⟩ (`,` ⟨*var_id*⟩)* `;`
⟨*var_id*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ `[` ⟨*arithm_exp*⟩ (`,` ⟨*arithm_exp*⟩)* `]`

⟨*user_defined_functions*⟩ ::= `Functions` ⟨*user_defined_functions_exp*⟩*
⟨*user_defined_functions_exp*⟩ ::= ⟨*def_constraint*⟩
| ⟨*def_function*⟩ `;`
⟨*def_constraint*⟩ ::= `DefConstraint` ⟨*id*⟩ ⟨*dfc_params*⟩ { ⟨*sentence*⟩+ }
⟨*def_function*⟩ ::= `Function` (`int` | `bool`) ⟨*id*⟩ ⟨*dfc_params*⟩ ⟨*sentence*⟩
⟨*dfc_params*⟩ ::= `(` (`int` | `bool`) ⟨*id*⟩ (`,` (`int` | `bool`) ⟨*id*⟩*) `)`

⟨*constraints*⟩ ::= `Constraints` ⟨*sentence*⟩+
⟨*sentence*⟩ ::= ⟨*statement*⟩
| ⟨*constraint*⟩ `;`

⟨*statement*⟩ ::= ⟨*if_then_else*⟩
| ⟨*forall*⟩
⟨*if_then_else*⟩ ::= `If` `(` ⟨*formula*⟩ `)` `Then` { ⟨*sentence*⟩+ }
| `If` `(` ⟨*formula*⟩ `)` `Then` { ⟨*sentence*⟩+ } `Else` { ⟨*sentence*⟩+ }
⟨*forall*⟩ ::= `Forall` `(` (⟨*id*⟩ `in` ⟨*list*⟩)+ `)` { ⟨*sentence*⟩+ }

⟨*constraint*⟩ ::= ⟨*formula*⟩
| ⟨*global_constraint*⟩
| `If_Then_Else` (⟨*formula*⟩) { ⟨*sentence*⟩+} { ⟨*sentence*⟩+}

⟨*formula*⟩ ::= `Not` ⟨*formula*⟩
| ⟨*formula*⟩ ⟨*bool_operator*⟩ ⟨*formula*⟩
| ⟨*arithm_exp*⟩ ⟨*relational_operator*⟩ ⟨*arithm_exp*⟩
| `(` ⟨*formula*⟩ `)`
| ⟨*user_defined_references*⟩
| ⟨*var_id*⟩
| `True`
| `False`

⟨*user_defined_references*⟩ ::= ⟨*id*⟩ `(` ⟨*list*⟩ `)`

⟨*relational_operator*⟩ ::= `=` | `<>` | `<` | `>` | `=<` | `>=`

⟨*bool_operator*⟩ ::= `And` | `Or` | `Xor` | `Iff` | `Implies`

⟨*global_constraint*⟩ ::= `AllDifferent` `(` ⟨*list*⟩ `)`
| `Sum` `(` ⟨*list*⟩ `,` ⟨*arithm_exp*⟩ `)`
| `Count` `(` ⟨*list*⟩ `,` ⟨*arithm_exp*⟩ `,` ⟨*arithm_exp*⟩ `)`

⟨*arithm_exp*⟩ ::= ⟨*numeral*⟩
| ⟨*var_id*⟩
| ⟨*arithm_exp*⟩ ⟨*arithm_operator*⟩ ⟨*arithm_exp*⟩
| `(` ⟨*arithm_exp*⟩ `)`
| `Abs` `(` ⟨*arithm_exp*⟩ `)`
| ⟨*user_defined_references*⟩

⟨*arithm_operator*⟩ ::= `+` | `-` | `*` | `Div` | `Mod`

⟨*list*⟩ ::= `[` ⟨*list_element*⟩ (`,` ⟨*list_element*⟩)* `]`
| `[` ⟨*arithm_exp*⟩ `|` ⟨*var_restrict*⟩(`,` ⟨*var_restrict*⟩)*`]`

⟨*list_element*⟩ ::= ⟨*arithm_exp*⟩
| ⟨*range*⟩

⟨*var_restrict*⟩ ::= ⟨*id*⟩ `in` ⟨*list*⟩
| ⟨*formula*⟩

⟨*range*⟩ ::= ⟨*arithm_exp*⟩ `..` ⟨*arithm_exp*⟩

⟨*optimitzation*⟩ ::= `Minimize` ⟨*id*⟩ `;`
| `Maximize` ⟨*id*⟩ `;`

⟨*id*⟩ ::= seqüència no buida de lletres/digits que no comença per digit i _

⟨*numeral*⟩ ::= seqüència no buida de digits

Capítol 7

Annex II: Implementació Haskell de la generació SMT

Golomb Ruler

```
import Char
import List
import Time

nl = "\n"

extrafuns x = "("++x++" Int)"

diferents x y = "(not (= "++ x ++" " ++ y ++"))"

mesgran n x = "(>= "++ x ++" "++ n ++")"

mespetit n x = "(<= "++ x ++" "++ n ++")"

restringirPosicions [] = []
restringirPosicions (x:[]) = []
restringirPosicions (x1:x2:l)
  = union [(mespetit x2 x1)] (restringirPosicions (x2:l))

restringirDif (m1,m2,m3,m4)
  = (diferents  ("(- " ++ m2 ++ " " ++ m1 ++ ")")
                ("(- " ++ m4 ++ " " ++ m3 ++ ")")
    )

-- com a minim un element
ajuntar (x:[]) = x
ajuntar (x:xs) = "(and "++ x ++" "++(ajuntar xs)++")"

golomb m l = ""++
  "(benchmark golombruler_"++(show m)++"_"++(show l)++".smt\n"++
  ":source {www.udg.edu ESLIP}\n"++
  ":status sat\n"++
  ":difficulty { 0 }\n"++
```

```
  ":category { testing }\n"++
  ":logic QF_LIA\n"++
  ":extrafuns ("++ (concat (map extrafuns marques)) ++ ")\n" ++
  ":formula ("++ (ajuntar llista) ++")"++
  -- tancar benchmark
  ")"

  where
    marques = ["m_"++(show i)| i<-[1..m]]
    diferencies1
      = [("m_"++(show i), "m_"++(show j), "m_"++(show k), "m_"++(show h))|
        i<-[1..m], j<-[(succ i)..m], k<-[1..m],
        h <- [(succ k)..m], (i/=k || j/=h )]
    domini_inferior_marques = (map (mesgran (show 0)) marques)
    domini_superior_marques = (map (mespetit (show l)) marques)
    fixar_marca_petita = "(= 0 m_1)"
    trencar_simetries
      = "(< (- m_2 m_1) (- m_" ++ (show m) ++ " m_" ++ (show (m-1)) ++ "))"
    restringir_diferencies1 = (map restringirDif diferencies1)

    fusio1 = (union restringir_diferencies1 (restringirPosicions  marques))
    fusio2 = (union [trencar_simetries] fusio1)
    fusio3 = (union [fixar_marca_petita] fusio2)
    fusio4 = (union domini_superior_marques fusio3)
    llista = (union domini_inferior_marques fusio4)

main m l= do
    t <- getClockTime
    print t
    writeFile ("golombruler_"++(show m)++"_"++(show l)++".smt") (golomb m l)
    t <- getClockTime
    print t
```

Social Golfer

```
import Char
import List
import Time

nl="\n"

extrafuns1 x = "("++x++" Int Int Int)"

extrafuns2 x = "("++x++" Int)"

extrafunsR [] = ""
extrafunsR (x:xs) =
  (concat (map extrafuns2 x))++
  extrafunsR xs

diferents x y = "(not (= "++ x ++" " ++ y ++"))"

iguals x y = "(= "++ x ++" " ++ y ++")"

mesgran n x = "(>= "++ x ++" "++ n ++")"

mespetit n x = "(<= "++ x ++" "++ n ++")"

ite n x = " (ite (= "++x++" "++n++") 1 0)"

numjug2 _ 0 _  = ""
numjug2 n_groups s n_players =
  (numjug n_groups n_groups (s-1) n_players)++
  (numjug2 n_groups (s-1) n_players)

numjug 0 _ _ _ = ""
numjug ga n_groups s n_players =
  "(= tjg_"++setmana++"_"++grup++" (+ "++
    (concat (map (ite grup) jugadors_setmana))++ "))\n"++
  (numjug (ga-1) n_groups s n_players)
  where
    setmana = show s
    grup = show (n_groups-ga)
    jugadors_setmana = ["js_"++(show i)++"_"++(show s)++"" |
                            i<-[0..n_players-1]]

cond_repetir (js1,js2,js3,js4)
    = "(not (and (= "++js1++" "++js2++ ") (= "++js3++" "++js4++")))"

-- com a minim un element
ajuntar (x:[]) = x
ajuntar (x:xs) = "(and "++ x ++" "++(ajuntar xs)++")\n"

ajuntarRec [] = ""
ajuntarRec (x:xs) = (ajuntar x)++(ajuntarRec xs)
```

```
golfer n_weeks n_groups group_size = ""++
  "(benchmark social_golfer_"++(show n_weeks)++"_"++(show n_groups)++
    "_"++(show group_size)++".smt\n"++
  ":source {www.udg.edu ESLIP}\n"++
  ":status sat\n"++
  ":difficulty { 0 }\n"++
  ":category { testing }\n"++
  ":logic QF_LIA\n"++
  ":extrafuns ("++ (extrafunsR definicions) ++ ")\n" ++
  ":formula ( and ("++sumar_totals++") (and "++ (ajuntarRec llista) ++"))"++
  -- tancar benchmark
  ")"

  where
    n_players = n_groups * group_size
    jugadors_setmana = ["js_"++(show i)++"_"++(show j)++""|
                        i<-[0..n_players-1], j<-[0..n_weeks-1]]
    total_setmana_group = ["tjg_"++(show i)++"_"++(show j)++""|
                        i<-[0..n_weeks-1], j<-[0..n_groups-1]]
    definicions = [jugadors_setmana, total_setmana_group]
    dif_jugadors_setmana = [("js_"++(show i)++"_"++(show j)++"",
                        "js_"++(show k)++"_"++(show j)++"",
                        "js_"++(show i)++"_"++(show l)++"",
                        "js_"++(show k)++"_"++(show l)++"")|
                        i<-[0..n_players-1], j<-[0..n_weeks-2],
                        k<-[(succ i)..n_players-1],
                        l<-[(succ j)..n_weeks-1]]

    domini_inferior_js = (map (mesgran (show 1)) jugadors_setmana)
    domini_superior_js = (map (mespetit (show n_groups)) jugadors_setmana)
    restingir_totals = (map (iguals (show group_size)) total_setmana_group)
    sumar_totals = numjug2 n_groups n_weeks n_players
    no_repetir = map cond_repetir dif_jugadors_setmana

    llista = [domini_inferior_js , domini_superior_js,
              restingir_totals , no_repetir]

main w g s
  = do
    t1 <- getClockTime
    print t1
    writeFile nomFitxer (golfer w g s)
    t2 <- getClockTime
    print t2
  where
    nomFitxer = "social_golfer_"++(show w)++"_"++(show g)++"_"++(show s)++".smt"
```

Capítol 8

Annex III: codi `Simply` dels problemes de comparació

Queens

```
Problem:queens_200
    Data
        n:=200;
    Domains
        Dom rows=[1..n];
    Variables
        IntVar q[n]::rows;
    Constraints
        Forall(i in [1..n-1]) {
            Forall(j in [i+1..n]) {
                q[i]<>q[j];
                q[i]-q[j]<>j-i;
                q[j]-q[i]<>j-i;
            }
        }
```

BACP

```
Problem:bacp_12_6
Data n_courses := 66; n_periods := 6;
     load_per_period_lb := 10; load_per_period_ub := 24;
     courses_per_period_lb := 2; courses_per_period_ub := 10;
Domains
     Dom periods=[1..n_periods];
     Dom addload=[load_per_period_lb..load_per_period_ub];
     Dom addcourses=[courses_per_period_lb..courses_per_period_ub];
     Dom load=[1..5]; Dom load_ext=[0..5];
Variables
     IntVar course_load[n_courses]::load;
     IntVar acourses[n_courses]::periods;
     IntVar mload[n_courses,n_periods]::load_ext;
     IntVar load_per_period[n_periods]::addload;
     IntVar course_per_period[n_periods]::addcourses;
Constraints
// carrega dels cursos
  course_load[1]=1;
  course_load[2]=3;
     ...
// prerequisits dels cursos
  acourses[7] < acourses[1];
  ...
  Forall(t in [1..n_periods]){
     Count([acourses[j] |j in [1..n_courses]],t,course_per_period[t]);
     Forall(c in [1..n_courses]){
       If_Then_Else( acourses[c] = t )
           { mload[c,t] = course_load[c];}
           { mload[c,t] = 0;} ;
     }
     Sum([ mload[i,t] | i in [1..n_courses] ], load_per_period[t]);
  }
```

Schurs Lemma

```
Problem: SchursLemma_10_3
   Data
      n_balls := 10;
      n_boxes := 3;
   Domains
      Dom d_boxes = [1..n_boxes];
   Variables
      IntVar putIn[n_balls] :: d_boxes;
   Constraints
     Forall(i in [1..n_balls]) {
        Forall(j in [1..n_balls]) {
           Forall(k in [1..n_balls]) {
              If (i+j=k) Then {
                (putIn[i] <> putIn[j]) Or (putIn[i] <> putIn[k]);
              }
           }
        }
     }
```

JobShop

```
Problem:jobshop_58
  Data n_machines := 5; n_jobs := 8;
       n_tasks_per_job := 5; max_duration:= 58;
  Domains Dom machines=[0..n_machines-1];
          Dom duration=[0..max_duration];
          Dom task_duration=[1..9];
  Variables
   IntVar job_task_start[n_jobs,n_tasks_per_job]::duration;
   IntVar job_task_machine[n_jobs,n_tasks_per_job]::machines;
   IntVar job_task_duration[n_jobs,n_tasks_per_job]::task_duration;
  Constraints
  // maquines segons feines i tasques
     job_task_machine[1,1]=1;
     ...
  // duració segons feines i tasques
     job_task_duration[1,1]=5;
     ...
     Forall (j in [1..n_jobs]) {
        Forall (k in [1..n_tasks_per_job-1]) {
            job_task_start[j, k] + job_task_duration[j, k]
            =<  job_task_start[j, k + 1]; }  }
     Forall(j in [1..n_jobs]) {
        job_task_start[j, n_tasks_per_job] +
        job_task_duration[j, n_tasks_per_job]
        =<  max_duration; }
     Forall(ja in [1..n_jobs-1]) {
       Forall(jb in [(ja+1)..n_jobs]){
         Forall(ka in [1..n_tasks_per_job]){
             Forall(kb in [1..n_tasks_per_job]){
               (job_task_machine[ja,ka] =  job_task_machine[jb, kb])
                Implies
               (( job_task_start[ja, ka] + job_task_duration[ja, ka]
                 =<  job_task_start[jb, kb] )
                Or
                ( job_task_start[jb, kb] + job_task_duration[jb, kb]
                =<  job_task_start[ja, ka] ) ); } } } }
```

Bibliografia

[ACG00] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. SAT-based procedures for temporal reasoning. In *5th European Conference on Planning, ECP'99*, volume 1809 of *LNCS*, pages 97–108. Springer, 2000.

[BDS02] Clark W. Barrett, David L. Dill, and Aaron Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *Computer Aided Verification, 14th International Conference, CAV'02*, volume 2404 of *LNCS*, pages 236–249. Springer, 2002.

[BHZ06] Lucas Bordeaux, Youssef Hamadi, and Lintao Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Comput. Surv.*, 38(4), 2006.

[BM10] M. Bankovic and F. Maric. Alldifferent constraint solver in smt. In *8th International Workshop on Satisfiability Modulo Theories*, 2010.

[BPSV09] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *8th International Workshop on Constraint Modelling and Reformulation, ModRef'09*, pages 30–44, Lisbon, Portugal, 2009.

[BPV09] Miquel Bofill, Miquel Palahí, and Mateu Villaret. A system for CSP solving through satisfiability modulo theories. In *IX Jornadas sobre Programación y Lenguajes (PROLE'09)*, pages 303–312, Donostia, Spain, 2009.

[BSV10] Miquel Bofill, Josep Suy, and Mateu Villaret. A system for solving constraint satisfaction problems with smt. In *Theory and Applications of Satisfiability Testing, 13th International Conference, SAT'10*, pages 300–305, 2010.

[CD96] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, June 1996.

[Cea10] Mats Carlsson and et al. *SICStus Prolog 4.1.2 User's Manual*. Swedish Institute for Computer Science, PO Box 1263, S-164 29 Kista, Sweden, 4.1.2 edition, October 2010. Available from `http://www.sics.se/sicstus/`.

[CIP+00] Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: an executable specification language for solving all problems in NP. *Computer Languages*, 26(2–4):165–195, July 2000.

[CMP06] Marco Cadoli, Toni Mancini, and Fabio Patrizi. SAT as an effective solving technology for constraint problems. In *Foundations of Intelligent Systems, 16th Intl. Symposium, ISMIS'06*, volume 4203 of *LNCS*, pages 540–549. Springer, 2006.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference record of third annual ACM symposium on theory of Computing*, pages 151–158. ACM, 1971.

[CS05] Marco Cadoli and Andrea Schaerf. Compiling problem specifications into SAT. *Artificial Intelligence*, 162(1–2):89–120, 2005.

[DC01] D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming*, 2001(6), October 2001.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *JACM*, 7(3):201–215, 1960.

[Ecl10] Eclipse features, 2010. http://www.eclipse-clp.org/eclipse/features.html.

[FHJ+08] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13(3):268–306, 2008.

[FJOS03] Flanagan, Joshi, Ou, and Saxe. Theorem proving using lazy proof explication. In *CAV: International Conference on Computer Aided Verification*, 2003.

[GHN+04] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification, 16th International Conference, CAV'04*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, 1979.

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *17th European Conference on Artificial Intelligence, ECAI'06*, volume 141 of *Frontiers in Artificial Intelligence and Applications*, pages 98–102. IOS Press, 2006.

[GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures — the case study of modal K. *Lecture Notes in Computer Science*, 1104, 1996.

[Ham03] Youssef Hamadi. Disolver: A distributed constraint solver. Technical Report MSR-TR-2003-91, Microsoft Research (MSR), December 2003.

[HG95] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[Hua08] Jinbo Huang. Universal booleanization of constraint models. In *Principles and Practice of Constraint Programming, 14th International Conference, CP'08*, volume 5202 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2008.

[ILO10] ILOG. Ibm ilog cp optimizer, 2010.

[JM87] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In *ICLP*, pages 196–218, 1987.

[Kau06] Henry A. Kautz. Deconstructing planning as satisfiability. In *Proceedings of the Twenty-first Conference on Artificial Intelligence, AAAI'06*, pages 1524–1526. AAAI Press, 2006.

[Kow74] R. A. Kowalski. Logic for problem solving. DCL Memo 75, Dept of AI, University of Edinburgh, March 1974.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Mes97] Pedro Meseguer. Interleaved depth-first search. In *IJCAI*, pages 1382–1387, 1997.

[MFM04] Mahajan, Fu, and Malik. Zchaff2004: An efficient SAT solver. In *International Conference on Theory and Applications of Satisfiability Testing, SAT'04*, volume 3542 of *LNCS*, pages 360–375. Springer, 2004.

[MH03] Laurent Michel and Pascal Van Hentenryck. Comet in context. In *PCK50: Proceedings of the Paris C. Kanellakis memorial workshop on Principles of computing & knowledge*, pages 95–107, New York, NY, USA, 2003. ACM.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[MR02] Leonardo De Moura and Harald Rueß. Lemmas on demand for satisfiability solvers. In *Proc. of the Fifth International Symposium on the Theory and Applications of Satisfiability Testing, SAT'02*, May 2002.

[MTW+99] Patrick Mills, Edward Tsang, Richard Williams, John Ford, James Borrett, and Wivenhoe Park. Eacl 1.5: An easy constraint optimisation programming language. Technical Report CSM-324, University of Essex, Colchester, U.K., 1999.

[MW98] Pedro Meseguer and Toby Walsh. Interleaved and discrepancy based search. In *ECAI*, pages 239–243, 1998.

[NO06] R. Nieuwenhuis and A. Oliveras. On SAT Modulo Theories and Optimization Problems. In *Theory and Applications of Satisfiability Testing, 9th International Conference, SAT'06*, volume 4121 of *LNCS*, pages 156–169. Springer, 2006.

[NORCR07] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Challenges in Satisfiability Modulo Theories. In *18th International Conference on Rewriting Techniques and Applications, RTA'07*, volume 4533 of *LNCS*, pages 2–18. Springer, 2007.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming, 13th International Conference, CP'07*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, November 1993.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *Proceedings of the Twenty-first Conference on Artificial Intelligence, AAAI'94*, pages 362–367, 1994.

[Ren10] Andrea Rendl. *Effective Compilation of Constraint Models*. PhD thesis, University of St Andrews, United Kingdom, 2010.

[Rou75] P. Roussel. *Prolog: Manuel de Reference et d'Utilisation Groupe d'Intelligence Artificielle*. Marseille-Luminy, 1975.

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.

[SS06] Hossein M. Sheini and Karem A. Sakallah. From propositional satisfiability to satisfiability modulo theories. In *Theory and Applications of Satisfiability Testing, 9th Intl. Conference, SAT'06*, volume 4121 of *LNCS*, pages 1–9. Springer, 2006.

[Wal75] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw Hill, 1975.

[Wal97] Toby Walsh. Depth-bounded discrepancy search. In *IJCAI*, pages 1388–1395, 1997.

[Wal00] Toby Walsh. SAT vs CSP. In *Principles and Practice of Constraint Programming, 6th International Conference, CP'00*, volume 1894 of *LNCS*, pages 441–456. Springer, 2000.

[Wie03] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, december 2003. Katholieke Universiteit Leuven. CW 371.

[Yok90] Makoto Yokoo. Distributed constraint satisfaction for DAI problem. In *The 10th International Workshop for DAI*, page 20, October 1990.

[ZLS04] Hantao Zhang, Dapeng Li, and Haiou Shen. A SAT based scheduler for tournament schedules. In *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT'04, Online Proceedings*, pages 191–196, 2004.