Universitat de Girona
Escola Politècnica Superior

Projecte fi de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Ampliació i millora del “Girona Optimization System”
TFG d’enginyeria informàtica

Document: Memòria

Alumne: David Pérez Sánchez

Tutor: Mateu Villaret Auselle i Jordi Coll Caballero
Departament: IMAE
Àrea: Llenguatges i sistemes informàtics

Convocatòria (mes/any): Setembre 2021

Universitat de Girona
Escola Politècnica Superior

PROJECTE FI DE GRAU

Ampliació i millora del “Girona Optimization System”

Autor:
David PÉREZ SÁNCHEZ

Setembre 2021

Grau en Enginyeria Informàtica

Tutors:
Mateu VILLARET AUSELLE
Miquel JORDI COLL CABALLERO

Agraïments

Per començar vull agrair molt especialment als meus tutors Mateu Villaret i Jordi Coll per la implicació i dedicació que han mostrat en totes les etapes del projecte. També vull donar les gràcies als professors Miquel Bofill i Josep Suy per l'atenció i ajut oferts.

Índex

1 Introducció **1**
1.1 Motivació 1
1.2 Objectiu 1

2 Viabilitat **3**

3 Metodologia **7**

4 Planificació **11**

5 Marc de treball i conceptes previs **13**
5.1 Compilador 13
5.1.1 Definició 13
5.1.2 Tipus 13
5.1.3 Estructura 15
5.2 Expressions regulars 21
5.3 Gramàtiques lliures de context 22
5.3.1 Ambigüitat 24
5.4 Gramàtica d'atributs 24
5.5 *Constraint Programming* 25
5.5.1 Codificació a SAT 27
5.5.2 Codificació MaxSAT 27

6 Requisits del sistema **29**
6.1 Funcionals 29
6.2 No funcionals 31

7 Estudi i decisions **33**
7.1 C++ 33
7.1.1 Gestió de memòria 34
7.2 Valgrind 34
7.3 Massif 35
7.4 perf 35
7.5 gdb 36
7.6 CMake 36
7.7 ANTLR4 37
7.7.1 Analitzador lèxic 37
7.7.2 Analitzador sintàctic 38

7.7.3 Analitzador semàntic i generació de codi 39
7.8 *Solvers* 40
7.9 DIMACS 40
7.9.1 Entrada 40
7.9.2 Sortida 41

8 Anàlisi i disseny del sistema 43
8.1 Estructura del programa 43
8.1.1 Compilador BUP 44
8.1.2 API LIA 44
8.1.3 *Solvers* 46
8.2 Correccions, millores i ampliacions 46
8.2.1 Escalabilitat 46
8.2.2 Rendiment 47
8.2.3 Predicats 50
8.2.4 Soft constraints 52
8.2.5 *Solvers* 54
8.2.6 *Debug* 55

9 Implementació i proves 57
9.1 Escalabilitat 57
9.2 Rendiment 58
9.2.1 Gestió de memòria 58
9.2.2 Temps de càlcul 61
9.2.3 Etapes de compilació 61
9.3 Predicats 62
9.3.1 Definició 62
9.3.2 Crida 70
9.3.3 *Operador sizeof()* 74
9.3.4 Taula de símbols 75
9.3.5 BUPFile 78
9.3.6 Errors 80
9.4 *Soft constraints* 82
9.4.1 Gramàtica 82
9.4.2 *Visitor* 82
9.4.3 *Encoding* 83
9.5 *Solvers* 86
9.5.1 Compilació 86
9.5.2 Procés de *solving* 87
9.5.3 Selecció de *solver* 88
9.6 *Debug* 89

9.7 Altres millores i correccions 92
9.7.1 Warnings 93
9.7.2 Prioritat dels operands 94
9.7.3 Operador *and* i *or* sobre llistes 94
9.8 Proves 95
9.8.1 Predicats 95
9.8.2 *Soft constraints* 100
9.8.3 *Solvers* 101
9.8.4 *Debug* 101

10 Implantació i resultats **103**
10.1 *Sequential Weight Counter* 103
10.1.1 SAT 103
10.1.2 MaxSAT 105
10.1.3 MaxSAT *Evaluations* 106
10.2 *Combinatorial Auctions* 108

11 Conclusions **111**

12 Treball futur **113**
12.1 Noves estructures de dades 113
12.2 Adaptació a nous formats DIMACS 113
12.3 Millora de l'eficiència 113
12.4 Adaptació per SAT-IT 113
12.5 Creació de llibreria de predicats 114
12.6 Generació de documentació LaTeX 114
12.7 Aplicació multiplataforma 114

Annex **115**

Manual d'usuari **121**

Bibliografia **125**

CAPÍTOL 1

Introducció

1.1 Motivació

Trobar la solució a un problema és la finalitat i objectiu de quasi tot procés. L'afany per descobrir aquesta solució és el que ha incentivat la creació de nombroses tècniques i paradigmes de programació. Dins aquests paradigmes es troba la programació amb restriccions o *Constraint Programming*, sovint emmarcada dins l'àmbit de la programació declarativa.

Aquest paradigma redueix la cerca de la solució a un problema combinatori a identificar l'assignació que satisfà les restriccions imposades sobre un conjunt de variables o, en altres paraules, trobar la possible solució al problema combinatori que resulti vàlida, entre totes les candidates.

La motivació pel desenvolupament d'aquest projecte radica en la necessitat de codificar les restriccions que defineixen els problemes d'una manera convenient. A la Universitat de Girona, concretament al grup de recerca de Lògica i Intel·ligència Artificial (LIA) del departament d'Informàtica, Matemàtica Aplicada i Estadística (IMAE), disposen d'un llenguatge declaratiu enfocat en el paradigma de la programació per restriccions anomenat BUP, que forma part del *Girona Optimization System* (GOS) [Generoso, 2020]. El sistema en qüestió és el resultat d'un anterior TFG que pretén suplir aquesta necessitat. Tot i ser funcional, tal com afirma el seu autor, es tracta d'una primera iteració i es troba en una fase inicial de desenvolupament. Això vol dir que GOS requereix algunes correccions, millores i ampliacions perquè sigui un sistema complet i funcional.

Així doncs, el propòsit és desenvolupar i millorar el sistema GOS per tal que pugui utilitzar-se en escenaris reals, és a dir, que sigui apte tant per l'àmbit de recerca com l'educatiu i, fins i tot, el professional.

1.2 Objectiu

L'objectiu principal d'aquest projecte és crear una eina que permeti modelar *Constraint Satisfaction Problems* (CSPs) d'una manera convenient, partint del Girona Optimization System.

Per assolir-lo es plantegen diverses correccions i ampliacions a implementar en el sistema. En quant a les correccions, es volen identificar i resoldre alguns

errors que provoquen que l'actual sistema falli o necessiti temps de còmput no raonables en determinades circumstàncies. També es pretenen polir certs aspectes de disseny, com la correcta inclusió de llibreries o l'apropiada gestió dels espais de noms entre altres i millorar la gestió de la memòria del procés on s'executa GOS.

Pel que fa a les ampliacions del sistema, caldrà fer abans una ampliació del llenguatge per dotar-lo de la gramàtica necessària. Les ampliacions es centren en implementar algunes funcionalitats enfocades a la facilitat d'ús de l'usuari, com la definició de predicats i la possibilitat d'incloure alters fitxers, i en completar el Girona Optimization System perquè suporti la optimització.

CAPÍTOL 2

Viabilitat

Aquest projecte estava emmarcat en l'àmbit acadèmic i realitzat per medis propis, de manera individual i en col·laboració amb el grup de recerca LIA del departament IMAE.

Per millorar i ampliar GOS va ser necessari disposar dels coneixements tècnics amb què estava dissenyat i implementat el sistema. Això seria indispensable per poder entendre i analitzar el sistema inicial i també per poder implementar els requeriments de manera adequada. A continuació s'enumeren els principals punts que van fer això possible:

- Coneixement teòric sobre el funcionament intern dels compiladors adquirit mitjançant la investigació i experimentació en el camp, a més de les bases adquirides a l'assignatura de Compiladors.

- Coneixement teòric sobre CSP adquirit a través de la interacció amb el grup de recerca LIA i a l'assignatura de PDA (Programació Declarativa, Aplicacions).

- Experiència en la utilització del llenguatge C++ i coneixença dels detalls interns de seu funcionament com, per exemple, la gestió de memòria, millors pràctiques o característiques del compilador.

- Accés als membres del grup de recerca LIA per consultes.

- Coneixement d'ANTLR4 i l'esquema de disseny de compiladors proposa aquesta llibreria, adquirit mitjançant experimentació i investigació pròpia, a més de les bases adquirides a l'assignatura de Compiladors.

Addicionalment vaig necessitar una infraestructura per desenvolupar el projecte. Per això vaig necessitar un ordinador que consistiria en un portàtil propi. Com que GOS estava ideat per compilar-se i executar-se en sistemes operatius basats en Linux, havia de disposar necessàriament al meu portàtil aquesta plataforma. El problema és que en el moment de desenvolupar el projecte no existien distribucions que suportessin el maquinari, és a dir, cap sistema operatiu basat en Linus no tenia els *drivers* necessaris per funcionar en el portàtil. Per tant, vaig haver de virtualitzar un Ubuntu 20.04.4 LTS dins el Windows 11 que tenia instal·lat l'ordinador. Per fer-ho, vaig utilitzar el WSL2 (*Windows Subsystem*

Linux 2) que ofereix Microsoft en els seus sistemes operatius Windows. Això és perquè consisteix en un sistema de virtualització més eficient que els convencionals i, per l'execució de tasques intensives, era el més adient.

Així doncs, els elements que van fer possible disposar d'una infraestructura de desenvolupament adequada van ser:

- Tenir en propietat un portàtil prou potent amb un sistema operatiu basat en Linux virtualitzat mitjançant WSL2 dins el Windows 11.
- Existència de *solvers* de codi obert com el Glucose, MiniSat o OpenWBO.
- Accés a editors i entorns de desenvolupament gratuïts en l'àmbit acadèmic com VIM o JetBrains CLion.
- Accés a les llibreries necessàries per el desenvolupament del projecte com ANTLR4 o l'API del grup LIA.
- Existència d'eines com LaTeX, CMake, Git, etc.

Pel que fa a la viabilitat econòmica, com es tractava d'un projecte individual i realitzat per medis propis, es va procurar minimitzar el cost mitjançant la utilització de programari lliure i gratuït. A continuació es fa una estimació del cost del projecte, tenint en compte tant els costos humans i el temps necessari per realitzar el desenvolupament i també els elements de *software* i equipament utilitzats.

Els costos humans estan calculats suposant un treball de 4 hores al dia i 5 dies de treball a la setmana. Els salaris suposats són:

- Programador → 15 €/h
- Analista → 20 €/h
- Dissenyador especialitzat → 25 €/h

Tasca	**Treballador**	**Hores**	**Cost**
Anàlisi GOS inicial	Analista	60	1200
Estudi conceptes/llibreries	Dissenyador i Programador	20	800
Disseny de millores	Dissenyador	48	1200
Millores estructurals	Analista i Programador	52	1820
Millores de rendiment	Analista i Programador	52	1820
Ampliació predicats	Dissenyador i Programador	52	2080
Ampliació *soft constraints*	Dissenyador i Programador	52	2080
Desacoblament dels *solvers*	Programador	52	780
Informació de *debug*	Programador	52	780
Altres millores i correccions	Programador	30	450
		Total	**13010€**

Els costos d'equipament són els següents:

Equipament	**Cost**
Asus UM425UAZ	900
ANTLR4	0
JetBrains CLion Educative License	0
Vim	0
Github	0
Diagrams.net	0
MiniSAT	0
Glucose	0
OpenWBO	0
LaTeX	0
GDB	0
Altres eines *open source*	0
Total	**900€**

CAPÍTOL 3

Metodologia

La metodologia seguida pel desenvolupament del projecte està basada en SCRUM. És una tècnica o metodologia de treball d'equips de tipus Agile, que consisteix en partir el procés de desenvolupament en parts diferenciades. El funcionament d'aquesta metodologia es basa en etapes, tal com es mostra a la figura 3.1. A continuació se'n descriuen, per ordre cronològic, les principals [Drumond, nd]:

- ***Backlog refinement*** → S'organitza el *backlog* o conjunt d'històries pendents i es prioritzen les tasques més importants.
- ***Sprint planning*** → Es realitza abans d'iniciar un *sprint* i és on es decideix quines històries es desenvoluparan durant el temps que dura el *sprint*.
- ***Sprint*** → Són intervals de temps fixes on es duu a terme el desenvolupament de les tasques previstes. Un *sprint* pot tenir duracions diverses, però acostumen a ser d'entre una i quatre setmanes.
- ***Daily stand up*** → Reunió diària de curta durada on es planeja la feina pel dia i es contextualitza amb l'equip què està desenvolupant cada membre.
- ***Sprint review*** → Un cop acabat el *sprint*, es fa una crítica de com ha anat, és a dir, es presenta la feina feta i s'analitza si s'han complert els objectius i tasques previstes.
- ***Sprint retorspective*** → L'equip fa una autocrítica de com ha estat la dinàmica de treball i es tracten problemes diversos que hagin pogut sorgir.

En SCRUM també hi ha rols de treball que adopten els membres de l'equip:

- ***Product owner*** → S'encarrega de gestionar el *backlog*, repartir les tasques, guiar a l'equip de desenvolupament i decidir els requeriments i prioritats. Es preocupa del producte.
- ***Scrum master*** → Assessoren els equips i els *product owners* en el procés de SCRUM. Es preocupa de la logística de la metodologia i vetlla perquè la dinàmica de treball sigui apropiada.
- ***Development team*** → S'encarreguen d'implementar les històries previstes i participen en les reunions.

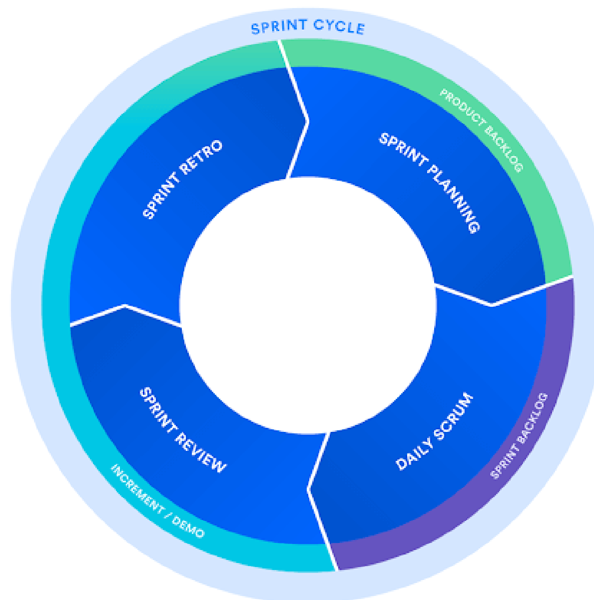

Figura 3.1: Etapes d'un *sprint* segons la metodologia SCRUM [Drumond, nd]

Donat que el projecte és individual, no s'ha pogut implementar al complet la metodologia. Tot i això, s'ha procurat ser-hi fidel el màxim possible. Les millores o ampliacions del sistema s'han agrupat en *sprints* acotats. Les reunions prèvies i posteriors a cada *sprint*, s'han dut a terme agrupades en una sola reunió, feta conjuntament amb membres del grup de recerca. En aquestes reunions es plantejaven tots els conceptes previstos per SCRUM abans d'iniciar un *sprint* i s'analitzaven els resultats de l'anterior.

El paper de *product owner* l'han adoptat els tutors del treball i les tasques de logística que duu a terme el *scrum master* més les de desenvolupament del *development team* les he adoptat jo.

En resum, el desenvolupament del projecte s'ha dut a terme en *sprints*, un per cada millora o ampliació prevista. Abans de començar-ne un i coincidint amb la finalització de l'anterior, s'ha fet una reunió amb els membres del grup de recerca LIA i els tutors del treball per mostrar les noves funcionalitats i comentar si s'adaptaven als requeriments. Addicionalment, es planejava el següent *sprint* i es refinava el *backlog*, és a dir, s'afegien nous requeriments que sorgissin a mida que s'anava desenvolupant el projecte i es modificaven els existents en cas que fos necessari.

Pel que fa a la dinàmica de treball, s'ha optat per seguir un dels models

que proposa Git. En concret, l'anomenat *feature branching*, que consisteix en crear una branca de desenvolupament per cada nova funcionalitat que es vulgui afegir, tal com es mostra a l'esquema de la figura 3.2. Aquesta branca es crea a partir de la branca principal o *master* i s'integra a la mateixa un cop acabada la funcionalitat. Els avantatges d'aquesta metodologia són que *master* sempre té un codi complet i sense codi a mig implementar. Addicionalment, permet que més d'un desenvolupador treballi en una mateixa funcionalitat. En el cas d'aquest projecte, el *feature branching* és molt adequat donat que les millores i ampliacions estan molt acotades.

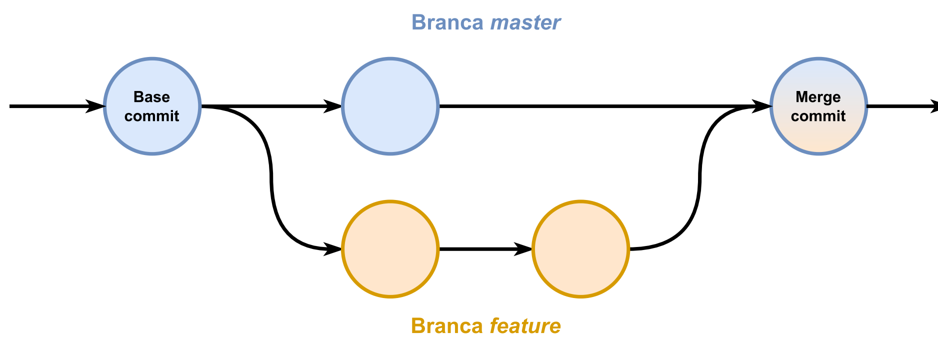

Figura 3.2: Esquema del funcionament de la dinàmica de treball amb Git anomenada *feature branching*

CAPÍTOL 4
Planificació

Com que la planificació es va realitzar seguint la metodologia SCRUM, el desenvolupament del projecte es va fraccionar en *sprints*. Cadascun agrupava les tasques necessàries per implementar una millora o ampliació del sistema prevista. Addicionalment, es van planificar dos períodes més per l'estudi, anàlisi i comprensió del sistema inicial i un altre per la confecció del informe i obtenció de resultats.

L'inici del projecte estava previst el dia 1 de març per realitzar l'entrega a la segona convocatòria, corresponent al mes de juny. Tot i això, quan es va fer la planificació inicial, es va veure que hi havia molta feina en l'estudi i comprensió del sistema, donada la complexitat de la implementació del sistema. Per aquesta raó es va optar per modificar la data de finalització per la convocatòria del setembre, concretament el dia 5. A continuació s'especifiquen les etapes de desenvolupament previstes, juntament amb una breu descripció i una estimació del temps requerit.

- **Estudi i anàlisi del sistema inicial (31d)** → Donat que aquest projecte es tractava de la continuació d'un altre projecte, era indispensable un estudi, comprensió del codi i funcionalitats existents per poder implementar de manera correcta les millores previstes. Per aquesta raó es preveia dedicar una etapa sencera a la lectura del codi i l'informe del TFG anterior. Addicionalment, es va planejar arreglar algun error conegut per incentivar l'exploració de tot el codi durant aquesta etapa inicial. Les correccions serien:
  - Trobar i solucionar l'error que provocava que el sistema funcionés extremadament lent en determinades situacions.
  - Millores en la correctesa de l'ús del llenguatge C++ per tal que el sistema fos més escalable.
- **Desenvolupament de les millores i ampliacions (6x21d)** → Cadascuna de les tasques es portaria a terme en *sprints* diferenciats amb una duració de 3 setmanes. També es realitzarien les tasques addicionals que preveu la metodologia SCRUM mitjançant reunions amb membres del grup de recerca. Els *sprints* que es portarien a terme serien:

  - **Escalabilitat** → Correcció dels errors en l'ús del llenguatge i l'estructura del projecte.
  - **Rendiment** → Identificació, anàlisi i millora dels problemes de rendiment i pèrdues de memòria de l'aplicació.
  - **Predicats** → Ampliació de les funcionalitats del llenguatge BUP per permetre la implementació de predicats personalitzats.
  - ***Soft constraints*** → Implementació de la possibilitat d'optimitzar problemes mitjançant l'anotació de pesos a les clàusules.
  - ***Solvers*** → Desacoblament de l'etapa de *solving* o cerca d'una solució a la fórmula compilada per dotar de més flexibilitat al sistema GOS.
  - ***Debug*** → Afegir l'opció de mostrar informació de *debug* als fitxer de sortida DIMACS per l'aplicació docent del sistema.

- **Obtenció de resultats i confecció de l'informe (31d)** → Un cop finalitzat el projecte, provar el sistema GOS amb les noves millores i ampliacions amb aplicacions reals i redactar la memòria del projecte.

	Març				Abril				Maig				Juny			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Estudi i anàlisi	■	■	■	■												
Escalabilitat					■	■	■									
Rendiment								■	■	■						
Predicats											■	■	■			
Soft constraints														■	■	■
Solvers																
Debug																
Resultats i informe																

	Juliol				Agost				Setembre			
	17	18	19	20	21	22	23	24	25	26	27	28
Estudi i anàlisi												
Escalabilitat												
Rendiment												
Predicats												
Soft constraints												
Solvers	■	■	■									
Debug				■	■	■						
Resultats i informe						■	■	■	■			

Figura 4.1: Diagrama de Gantt que representa la planificació de temps del projecte

CAPÍTOL 5

Marc de treball i conceptes previs

5.1 Compilador

5.1.1 Definició

Un compilador consisteix bàsicament en un programa capaç de traduir programes escrits en un llenguatge a codi màquina o codi objecte. Llavors aquest codi pot ser executat per l'arquitectura específica per la que ha estat compilat. [Muchnick, 1997] Tot i això, el concepte de compilador s'ha generalitzat per passar a considerar-se un traductor de codi. Gràcies a aquesta generalització n'existeixen nombrosos tipus, que s'expliquen a la subsecció 5.1.2.

Tot i la diferenciació per tipologies, tots els compiladors comparteixen uns principis bàsics. A alt nivell, tenen tots una mateixa estructura, que consisteix en dividir el procés de traducció en, com a mínim, quatre etapes: anàlisis lèxic, anàlisis sintàctic, anàlisis semàntic i generació de codi (veure 5.1.3).

5.1.2 Tipus

Tal com s'ha comentat a l'apartat anterior, si generalitzem el concepte de compilador, es defineix com a traductor de codi. Segons com, quan i a quin codi objectiu es vulgui traduir, els compiladors es poden dividir en grups.

5.1.2.1 Compiladors *single-pass* vers *multi-pass*

La traducció del programa font es realitza en etapes encadenades linealment i només es té el codi desitjat un cop acabades totes les passes. Si es duguessin a terme ininterrompudament (compilador d'una passada o *single-pass compiler*), vol dir que s'hauria d'escriure un compilador per cada arquitectura perquè, tal com es comenta a 5.1.3, hi ha etapes com la generació de codi (veure 5.1.3.4) que són específiques per l'arquitectura del sistema. Per aquesta raó els compiladors actuals tendeixen a ser compiladors de múltiples passades o *multi-pass compilers*.

Aquests darrers parteixen el procés de traducció per poder desacoblar la part que depèn del sistema de la part independent. Tal com mostra la figura 5.1, es diferencien dos parts abstractes del procés de compilació: l'anàlisi del

codi i la síntesi. Per poder desacoblar aquestes dues parts, es requereix un pas addicional en el proces de compilació, que és la generació de codi intermedi (veure 5.1.3.5).

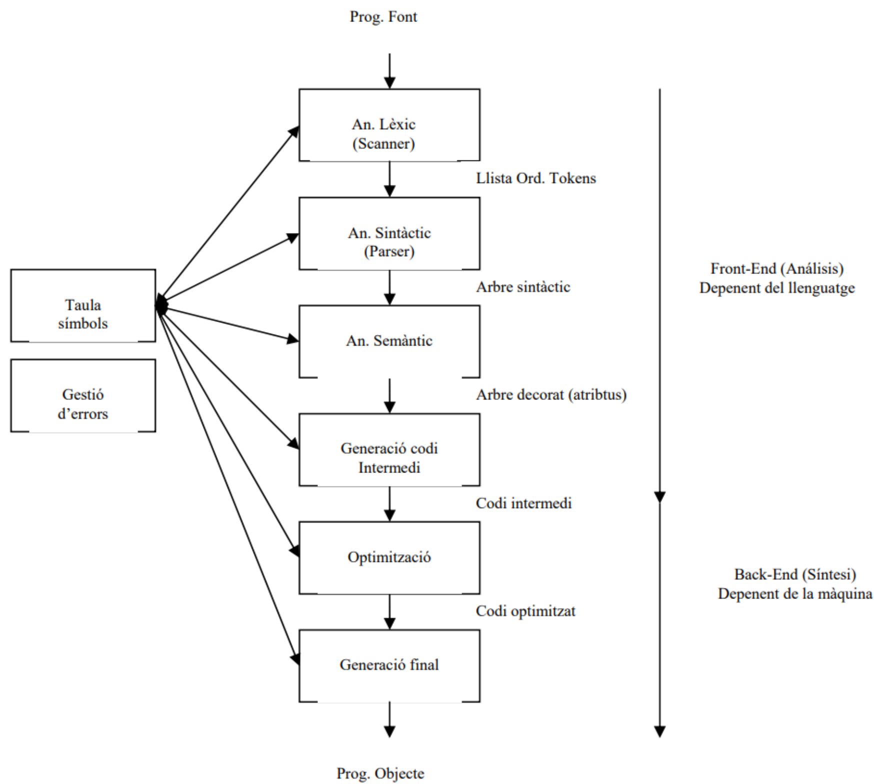

Figura 5.1: Etapes de compilació agrupades segons si corresponen a l'anàlisi del codi font i la síntesi del codi objectiu [Suy, 2022]

5.1.2.2 Compilador vers intèrpret

Suposant que l'objectiu de la compilació és obtenir un codi executable, els compiladors tradueixen un programa escrit en un llenguatge d'alt nivell a codi màquina. Aquest procés es duu a terme amb una temporalitat diferent a l'execució. Tot i això, existeixen una modalitat de compiladors que van executant les línies de codi a mida que les tradueixen que s'anomenen intèrprets.

En el cas dels compiladors, el codi font es pot traduir directament a un executable. Llavors aquest es podrà executar tants cops com es vulgui. En canvi, els intèrprets hauran de traduir tot el codi font cada vegada que es vulgui executar el programa.

5.1.2.3 Compilador vers transpilador

Existeix una modalitat de traducció que no pretén obtenir un executable com a resultat del procés de compilació, sinó que l'objectiu és traduir un programa escrit en un llenguatge d'alt nivell a un altre llenguatge del mateix estil. Aquest tipus de compilador s'anomena transpilador.

5.1.3 Estructura

Tot i la gran diversitat de compiladors existent (veure 5.1.2), el procés de traducció que es duu a terme per convertir un programa font en un altre codi o programa comparteix una sèrie d'etapes i estructures auxiliars indispensables.

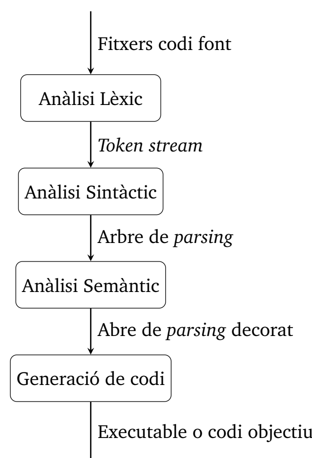

Figura 5.2: Etapes de compilació bàsiques

5.1.3.1 Anàlisi lèxic

El compilador rep el codi del programa font com a entrada en forma de *string* o cadena de caràcters. El paper de l'anàlisi lèxic és interpretar i separar aquesta entrada en *tokens* o subcadenes que corresponen a les diferents unitats lèxiques del codi.

Els *tokens* s'identifiquen segons unes regles o patrons lèxics. Aquests patrons s'expressen utilitzant expressions regulars (veure 5.2) i consisteixen en una descripció de quins caràcters poden formar les diferents unitats lèxiques del llenguatge. Un cop identificats els *tokens*, s'assigna a cadascun una sèrie d'atributs bàsics (veure 5.4) com, per exemple, el número de línia i posició o el lexema. Cal destacar que també existeixen altres atributs que només tenen sentit per certs tipus de *tokens* com, per exemple, el valor enter numèric d'un número.

Si apliquéssim l'anàlisi lèxic a la següent línia de codi escrita en pseudocodi:

```
1   var i: boolea := 1+2*3 = 9;
```

Utilitzant patrons definits amb expressions regulars de l'estil:

```
1   IDENT    =  [a-zA-Z][a-zA-Z0-9]*
2   NUM      =  [1-9][0-9]* | 0
3   PC_VAR   =  var
4   DOSPUNTS =  :
5   PC_BOOLEA = boolea
6   ASSIGN   =  :=
7   OP_SUM   =  +
8   OP_MUL   =  *
9   OP_IGU   =  =
10  PUNTCOMA =  ;
11  ...
```

Obtindríem el següent *token stream* (ignorant els espais innecessaris), on cadascun tindria associats els attributs pertinents:

```
1   PC_VAR IDENT DOSPUNTS PC_BOOLEA ASSIGN NUM OP_SUM NUM OP_MUL NUM OP_IGU NUM
        PUNTCOMA
```

5.1.3.2 Anàlisi sintàctic

En l'etapa de l'anàlisi sintàctic (també anomenada de *parsing*) s'agafa com a entrada el *token stream* generat a l'anàlisi lèxic i es processa per determinar-ne l'estructura. En altres paraules, es comprova que les unitats lèxiques del programa es trobin en una posició coherent.

L'estructura sintàctica d'un programa es pot descriure mitjançant regles gramaticals d'una CFG (veure 5.3). Basant-se en aquestes regles, el *parser* genera un arbre de *parsing* o arbre de derivació. Suposem, per exemple, el *token stream* generat a partir de l'exemple proposat a l'apartat 5.1.3.1:

```
1   PC_VAR IDENT DOSPUNTS PC_BOOLEA ASSIGN NUM OP_SUM NUM OP_MUL NUM OP_IGU NUM
        PUNTCOMA
```

Utilitzant les regles gramaticals corresponents a una declaració i assignació en pseudocodi:

```
1   declaracio  = modif IDENT DOSPUNTS tipBasic
2   modif       = PC_VAR | PC_CONST
3   tipBasic    = PC_BOOLEA | PC_ENTER | PC_REAL | PC_CAR
4   assignacio  = declaracio ASSIGN exprRel
5   exprRel     = exprSumRest (opRel exprSumRest)*
6   exprSumRes  = exprMulDiv (opSumRes exprMulDiv)*
```

```
 7  exprMulDiv  = exprNeg (opMulDiv exprNeg)*
 8  exprNeg     = opNeg exprBase | exprBase
 9  exprBase    = valBasic | PAREN_ESQ exprRel PAREN_DRE
10  valBasic    = NUM | CAR | REAL | BOOLEA
11  opRel       = OP_IGU | OP_DIF | OP_MAJ | OP_MEN | OP_MAJ_IGU | OP_MEN_IGU
12  opSumRes    = OP_SUM | OP_RES
13  opMulDiv    = OP_MUL | OP_DIV
14  opNeg       = OP_NEG
```

L'arbre de *parsing* resultant es mostra a la figura 5.3.

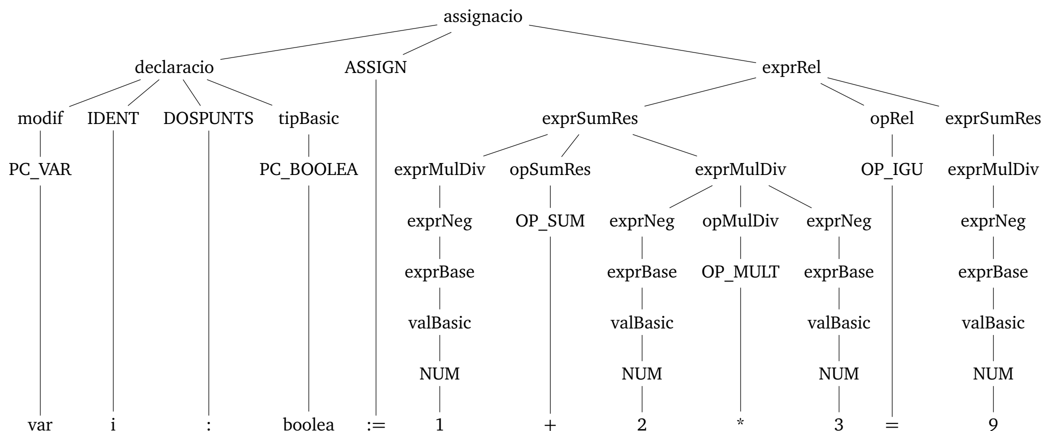

Figura 5.3: Arbre de *parsing* o derivació corresponent a "var i: boolea := 1+2*3 = 9;"

A més de generar l'arbre de *parsing*, en l'etapa de l'anàlisi sintàctic, es detecten errors estructurals com *tokens* fora de lloc o blocs de llenguatge mal construïts, entre altres.

5.1.3.3 Anàlisi semàntic

L'anàlisi semàntic s'encarrega de dotar de significat als diferents elements del codi font i verificar-ne la coherència semàntica basant-se en unes regles semàntiques. L'entrada d'aquesta etapa és l'arbre de *parsing* generat a l'anàlisi sintàctic. A partir de l'arbre, es realitzen les comprovacions semàntiques amb l'ajut de la taula de símbols (veure 5.1.3.6).

Les tasques principals que es solen dur a terme en aquesta etapa son [Suy, 2022]:

- **Ús apropiat dels identificadors** → Cal haver declarat els identificadors (per exemple, variables o constants) abans d'utilitzar-se. Aquest control es realitza amb l'ajut de la taula de símbols (veure 5.1.3.6) on, cada cop que es declara un identificador, es crea un registre nou a la taula i així tenir constància de la seva declaració.

- **Control de tipus** → Les expressions que s'avaluen al llarg del codi font han tenir un tipus determinat segons el lloc on es troben. Per exemple, en una estructura condicional, l'expressió de la condició ha de ser de tipus booleà. També es comprova que no es realitzin operacions incorrectes, com realitzar operacions aritmètiques entre tipus incompatibles (com sumar un booleà i un nombre real). Cal destacar que segons el disseny del llenguatge, es poden fer conversions de tipus com, per exemple, en cas de sumar un enter i un real, convertir l'enter a real.

- **Inferència de tipus** → Per poder realitzar el control de tipus, cal conèixer el tipus de les expressions. Seguint les regles d'inferència de tipus i les regles semàntiques, el compilador dedueix el tipus mitjançant l'exploració de l'arbre de derivació. Per exemple, suposem l'expressió `1+2 < 3` i que el llenguatge suporta les operacions típiques d'un llenguatge modern. La suma `1+2` s'avalua amb tipus enter (perquè tots dos operands són enters) i el número `3` també com a enter (perquè és un literal numèric sense decimals). Llavors el resultat de l'operació relacional `<` entre enters s'avalua com a booleà i el tipus general de l'expressió és booleà.

- **Control de crides** → En cas que el llenguatge permeti funcions, es comprova que s'hagi declarat una funció amb signatura compatible abans de cridar-se i que la crida es faci correctament.

- **Comprovacions posposades a l'anàlisi sintàctic** → Existeixen comprovacions que, tot i poder-se dur a terme a l'etapa d'anàlisi sintàctic, podrien complicar molt la gramàtica del llenguatge. Si suposem que el llenguatge en què està escrit el codi font suporta el tipus "dataçom a tipus bàsic, caldria comprovar la correctesa de les dates, tenint en compte el nombre de dies irregulars als mesos, anys de traspàs, etc. Tot i que és possible escriure les regles gramaticals necessàries, fer aquest control semànticament resulta més senzill.

Així doncs, en l'anàlisi semàntic, es realitza el control d'errors amb l'ajut de la taula de símbols (veure 5.1.3.6) i dels atributs sintetitzats i heretats calculats mitjançant l'exploració de l'arbre i l'aplicació de les regles semàntiques (veure 5.4). Aquests atributs addicionals s'incorporen a l'arbre de *parsing* amb els atributs incorporats a l'anàlisi lèxic. Alguns exemples d'atributs calculats en aquesta etapa poden ser el valor i tipus de les expressions. [Suy, 2022]

5.1.3.4 Generació de codi final

Aquesta és l'etapa final del procés de compilació si es suposa una compilació simple, sense etapes addicionals (veure 5.1.3.5).

La tasca d'aquest pas és generar el codi resultant del procés de compilació. Per fer-ho s'explora l'arbre de *parsing* decorat resultant d'aplicar l'anàlisi semàntic amb la finalitat de generar les instruccions pertinents codificades en el llenguatge objectiu.

En aquesta etapa també es sol generar la informació de depuració o *debugging* mitjançant la taula de símbols i l'arbre de *parsing* decorat.

5.1.3.5 Etapes addicionals

En els apartats anteriors es descriu el procés mínim i indispensable per poder compilar programes. En la majoria dels compiladors moderns però, s'apliquen etapes addicionals com les que es descriuen a continuació. La figura 5.4 il·lustra un procés de compilació complet.

- **Generació de codi intermedi** → Per poder desacoblar la part del procés de compilació depenent del sistema de la part independent, cal generar un codi intermedi. Es tracta d'un codi molt proper al codi màquina però que no es pot executar directament normalment anomenat codi objecte. Aquest es pot generar independentment del sistema on es compili i serveix com a entrada a la darrera part de compilació, que sí depèn del sistema. Aquesta metodologia és la que utilitzen els compiladors *multi-pass* (veure 5.1.2.1).

- **Optimització** → La optimització del codi és un procés opcional que ajuda a millorar l'eficiència del codi i reduir-ne els requeriments. Les millores que es poden aplicar poden ser totalment independents del sistema on es compili i dependre només de com s'ha programat el programa. Alguns exemples serien eliminar variables declarades però no utilitzades o replicar el codi de bucles senzills per evitar l'*overhead*[1] d'aquest tipus d'estructures. En canvi, existeixen altres millores que depenen del sistema, com seria la utilització d'instruccions específiques per un tipus de processador.

- **Preprocessat** → Certs llenguatges permeten incloure directives en el codi que no formen part de la gramàtica del llenguatge en qüestió, sinó que pertanyen al conjunt de directives suportades pel preprocessador. Aquestes directives són avaluades pel preprocessador abans del procés de compilació i permeten mecanismes com la definició i utilització de *macros*, inclusió d'altres fitxers, compilació condicional de seccions de codi o l'aplicació de regles específiques de compilació en algunes seccions de codi. [IBM, 2021]

[1]Recursos extra invertits en la gestió de l'execució d'un procés.

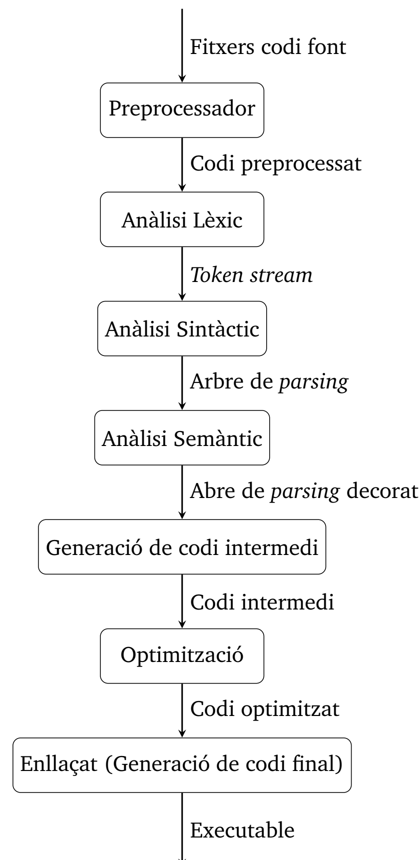

Figura 5.4: Compilador *multi-pass* amb totes les etapes mínimes i opcionals (inspirat en el compilador del llenguatge C++)

5.1.3.6 Taula de símbols

La taula de símbols és una estructura de dades utilitzada en el procés de compilació per associar a cada símbol del codi font un contingut semàntic. La informació guardada s'estructura en forma de registres que contenen totes les dades necessàries per fer les comprovacions d'errors semàntiques i la generació de codi.

Aquesta taula ha de permetre inserir nous registres, cercar-ne d'existents segons un identificador del símbol i modificar-ne els atributs en cas de ser necessari. Una possible implementació de la taula de símbols podria ser en forma de *map* o diccionari amb registres per cada tipus de símbol indexats per el lexema del símbol o signatura de la funció.

La informació que sol emmagatzemar-se als registres són, per exemple [Suy, 2022]:

- Nom o lexema del símbol. Acostuma a ser el mètode d'indexació dels re-

gistres.

- Adreça de memòria on es troba el valor de la variable o constant.
- Tipus del símbol. Serveix, per exemple, per fer el control d'errors o la inferència de tipus en l'anàlisi semàntic (veure 5.1.3.3).
- Posició del símbol en el codi. Normalment en la forma de número de línia i posició dins la mateixa. Serveix per mostrar missatges d'errors convenients per l'usuari i per poder generar la informació de depuració.

Segons si el llenguatge funciona amb àmbit d'execució, la taula de símbols ha de permetre reconèixer en quin àmbit ha estat declarat cada símbol i des de quins pot ser utilitzat. Aquesta funcionalitat pot implementar-se tenint una taula de símbols diferent per cada àmbit (per exemple, amb una pila de taules) o organitzant la taula en blocs o *frames*.

5.2 Expressions regulars

Una expressió regular és una expressió que descriu un llenguatge regular. Un llenguatge és conjunt de *strings* o cadenes de caràcters i es considera regular si un DFA o autòmat finit determinista és capaç de reconèixer-lo.

Un DFA consisteix en una màquina d'estats finita capaç de reconèixer *strings*. Està format per un conjunt d'estats (on n'hi ha un d'inicial i un conjunt de finals) i unes funcions o regles de transició que defineixen el canvi entre estats depenent del símbol de l'alfabet (propi de l'autòmat) que es llegeix com a entrada. Es considera que un DFA reconeix un *string* si després de consumir un a un tots els símbols que formen el *string* i aplicant les regles de transició corresponents a cada pas, l'autòmat acaba en un estat acceptador. Altrament es considera que el DFA rebutja el *string*. Llavors, el conjunt de tots els *strings* que reconeix el DFA es diu que és el llenguatge que regoneix el DFA [Bofill, 2020].

La figura 5.5 mostra un exemple d'aquest tipus d'autòmat. Està format per el conjunt d'estats $\{q_1, q_2, q_3\}$, on l'estat inicial és q_1 i el conjunt d'estats finals és $\{q_2\}$. Les regles de transició entre estats són:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

El llenguatge que reconeix aquest DFA és el conjunt de tots els *strings* que acaben en 1 o amb un nombre parell de 0 després de, com a mínim, un 1.

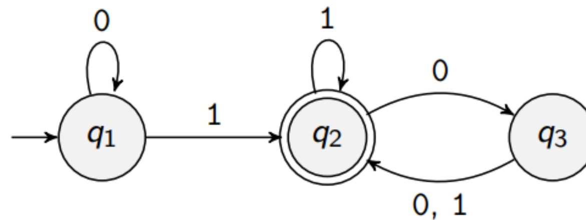

Figura 5.5: Exemple d'un DFA [Bofill, 2020].

Una altra manera d'entendre les expressions regulars equivalent a la definició anterior és com un patró que coincideix amb un *string*. Llavors el llenguatge d'una expressió regular són tots aquells *strings* que encaixen amb el patró.

Aquest patró es construeix amb operacions regulars entre símbols de l'alfabet que accepta l'expressió regular. Per tant, s'entén que R és una expressió regular amb alfabet A si R és [Bofill, 2020]:

- Un **símbol** dins l'alfabet A.
- Un ***string* buit** o *string* sense símbols anomenat ε.
- Un **conjunt buit** de *strings* $\emptyset$.
- Una **unió d'expressions regulars** de l'estil $A \cup B$, on A i B són expressions regulars.
- Una **concatenació d'expressions regulars** de l'estil $A \circ B$, on A i B són expressions regulars. La concatenació sol expressar-se sense l'operador $\circ$ per qüestions de simplicitat.
- La **clausura de Kleene** aplicada sobre una expressió regular. Per exemple A^*, on A és una expressió regular. Aquest operador és l'equivalent a concatenar A cap o n vegades seguides.

Un exemple d'expressió regular, equivalent al DFA de la figura 5.5 seria:

$$0^*1(1^* \cup (01)^* \cup (00)^*)$$

5.3 Gramàtiques lliures de context

Les expressions regulars o DFAs (veure 5.2) són eines molt útils i àmpliament aplicades en molts àmbits. Tot i això, tenen certes limitacions i és que, tal com s'expressa a la seva definició, són capaces de reconèixer llenguatges regulars. Això vols dir que només poden descriure un subconjunt de tots els llenguatges existents.

Per exemple, un DFA no seria capaç de descriure el llenguatge que comprèn tots els *strings* amb igual nombre de zeros que d'uns de manera ordenada (per exemple "01", "0011", "000111", etc.), formalment descrit com $L = \{0^n 1^n \mid n \geq 0\}$ (*strings*. Això és perquè, tal com indica el seu nom, un autòmat **finit** determinista té memòria finita i per poder descriure el llenguatge indicat necessitaria memòria il·limitada i, per tant, es pot afirmar que es tracta d'un llenguatge irregular. En l'exemple descrit s'ha justificat de manera intuïtiva perquè el llenguatge L no és regular. La demostració també es podria fer de manera formal mitjançant el *Pigeonhole Principle* o el *Pumping Lemma*, però queda fora de l'objectiu d'aquest apartat.

Els llenguatges lliures de context o **CFL** (*Context Free Language*) consisteixen un conjunt de llenguatges no regulars que inclou els llenguatges regulars. De la mateixa manera que els DFAs o les expressions regulars descriuen un llenguatge regular, una CFL és descrita per una gramàtica lliure de context o CFG (*Context Free Grammar*). Per tant, es pot entendre que les CFG són més potents que un DFA o expressió regular, entenent potència com a la capacitat de reconèixer llenguatges [Bofill, 2020].

Una **CFG** està formada per un conjunt de variables (on una és la variable inicial), un conjunt de símbols terminals i un conjunt de regles de producció o derivació. Per exemple, la gramàtica que descriu les operacions aritmètiques bàsiques tindria les següents regles de producció :

$$
\begin{aligned}
&S \rightarrow E \\
&E \rightarrow num \mid (E) \\
&E \rightarrow E + E \\
&E \rightarrow E - E \\
&E \rightarrow E * E \\
&E \rightarrow E / E
\end{aligned}
$$

On el conjunt de variables seria $\{S, E\}$, la variable inicial seria S, i els símbols terminals serien *num* (qualsevol número).

Utilitzant les regles de producció d'una gramàtica es poden generar tots els *strings* que és capaç de reconèixer la gramàtica. Això es pot fer mitjançant l'aplicació successiva de regles de derivació fins a tenir un *string* sense variables. Per exemple, per produir el *string* 1+2*3 amb la gramàtica anterior, la derivació possible seria:

$$S \rightarrow E \rightarrow E + E \rightarrow E + E * E \rightarrow num + num * num \rightarrow 1 + 2 * 3$$

5.3.1 Ambigüitat

Una gramàtica ambigua és aquella que permet generar o derivar un *string* de maneres diferents. Si agafem com exemple la gramàtica de l'apartat anterior que descriu les operacions aritmètiques bàsiques, el *string* 1+2*3 es pot generar amb una altra derivació diferent a la ja expressada. Expressant ambdues derivacions en forma d'arbre es pot observar que, tot i tenir una estructura diferent acaben generant el mateix *string*, tal com es mostra a la figura 5.3 [Bofill, 2020].

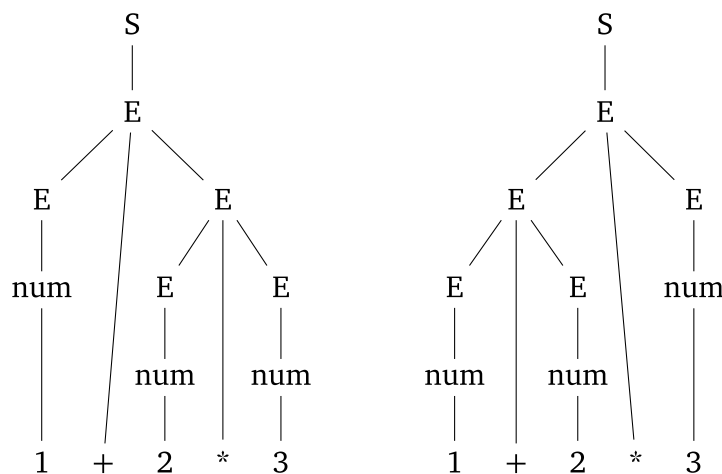

Figura 5.6: Dues derivacions diferents per al *string* "1+2*3", segons una gramàtica ambigua

5.4 Gramàtica d'atributs

Les gramàtiques d'atributs són una extensió de les gramàtiques lliures de context (veure 5.3) que incorporen la possibilitat d'associar dades (atributs) a les variables i símbols. Per tant, una gramàtica d'atributs es defineix com una CFG però amb alguns elements addicionals [Suy, 2022]:

- Cada variable o símbol té associat un conjunt d'atributs.
- Cada regla de producció o derivació té associat un conjunt de funcions que són les que determinen com es computen els atributs.
- Cada regla té un conjunt de predicats o funcions booleanes que comproven la consistència dels atributs.

Els atributs poden ser de dos tipus: sintetitzats i heretats. Per poder definir-los, suposem que les regles de producció d'una variable A tenen la forma de $A \rightarrow \alpha\beta$, on α és el conjunt de símbols (nodes terminals) i β és el conjunt de variables (nodes no terminals). Llavors, aquests atributs es defineixen com:

- **Sintetitzats** → Els atributs sintetitzats d'una variable que apareix a la part esquerra d'una regla de producció són aquells aquells que es calculen només a partir dels atributs d'elements que es troben a la part dreta de la regla. Per exemple, els atributs sintetitzats de A serien aquells computats mitjançant atributs corresponents a d'α i β. En altres paraules, suposant la representació d'arbre d'una derivació, els atributs sintetitzats són aquells que es calculen utilitzant valors només dels fills d'un node. Cal destacar que els símbols (no variables) o nodes terminals poden tenir atributs sintetitzats però, com que no apareixen mai a la part esquerra d'una regla de producció, són assignats i no pas calculats.

- **Heretats** → Els atributs heretats d'una variable que apareix a la part dreta d'una regla de producció són aquells que es calculen a partir d'atributs corresponents als altres elements de la part dreta de la regla o dels atributs de la variable de la part esquerra. Per exemple, suposem que B és una variable que pertany al conjunt β. Llavors els atributs heretats de B són aquells calculats a partir dels atributs de A, els símbols de α o les variables de $\beta-\{B\}$. Si suposem la representació en arbre, els atributs heretats d'un node estarien calculats a partir dels nodes germans o del node pare.

5.5 *Constraint Programming*

La programació amb restriccions o *Constraint Programming* (CP) és un paradigma de programació que s'utilitza sobretot per la cerca de solucions a problemes combinatoris, és a dir, amb moltes possibles solucions candidates o, el que és el mateix, un espai de cerca molt gran. Donat que es tracta d'un tipus de programació declarativa, la filosofia d'aquest paradigma és descriure les característiques del problema a resoldre d'una manera expressiva però sense donar detalls de com o quin procés seguir per trobar la solució.

Un *constraint* o restricció consisteix en una relació lògica sobre un conjunt de variables. Cada variable té un domini o conjunt dels possibles valors que pot prendre. Les restriccions limiten el domini de les variables i determinen quins valors poden tenir a la vegada. Per exemple, suposem que tenim dues variables x i y amb els dominis $D_x = \{1,2,3\}$ i $D_y = \{1,2,3\}$. Si imposem la restricció $x < y$ llavors els dominis de les variables es redueixen a $D_{x'} = \{1,2\}$ i $D_{y'} = \{2,3\}$.

El propòsit de la programació amb restriccions és resoldre problemes descrits amb *constraints*. Una modalitat de problemes que contempla aquest paradigma són els problemes de satisfacció de restriccions o CSPs (*Constraint Satisfaction Problem*). Aquests es descriuen com [Barták, nd]: :

- Un conjunt finit de variables.

- Un conjunt finit de possibles valors que poden adoptar cadascuna de les variables (dominis finits).

- Un conjunt finit de *constraints* que restringeix els valors del seu domini que prenen les variables.

La solució a un CSP és l'assignació de variables amb els valors del seu domini que satisfà totes les restriccions. En cas d'existir alguna solució, es considera que el problema és satisfactible i, altrament, es considera insatisfactible. Segons les característiques del problema poden existir una o més solucions. La cerca de la solució a un CSP és un problema combinatori de complexitat NP. En concret, consisteix en un problema NP-complet

Figura 5.7: Exemple de CSP que consisteix en la coloració d'un mapa evitant repetir colors en països adjacents. Les variables serien els diferents països del mapa. Totes les variables tindrien el mateix domini, que consisteix en els colors disponibles. Les restriccions serien que cada país tingués un color i que dos països que comparteixen frontera no tinguin el mateix color. [D-Wave Ocean, nd]

Existeixen diverses tècniques per expressar o modelar CSP i resoldre'ls. En aquest projecte s'exploren les metodologies de codificació a SAT i MaxSAT, explicades en els següents apartats.

5.5.1 Codificació a SAT

Aquesta metodologia consisteix en reduir un CSP a un problema de satisfactibilitat booleana (SAT). Un problema SAT està format també per un conjunt finit de variables, però en aquest cas són variables booleanes (literals), és a dir, amb domini definit pel conjunt de valors $\{true, false\}$. Les restriccions es codifiquen en fórmules proposicionals, és a dir, amb una combinació de variables booleanes i operadors lògics. Els problemes CSP es poden convertir a SAT en temps polinòmic[2], ja que SAT és NP-complet.

Les fórmules acostumen a expressar-se en *Conjunctive Normal Form* (CNF), que consisteix en una conjunció de clàusules. Una clàusula és una disjunció de literals positius o negatius. Per exemple, la fórmula $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3 \vee \neg x_5) \wedge (x_2 \vee \neg x_4)$ està en format CNF [Biere et al., 2009].

La solució a un problema SAT consisteix en trobar una assignació de valors de veritat a les variables de la fórmula que satisfaci totes les clàusules. Per exemple, una solució a la fórmula anterior, seria l'assignació:

$$\{x_1 \rightarrow true,\ x_2 \rightarrow false,\ x_3 \rightarrow true,\ x_4 \rightarrow false,\ x_5 \rightarrow true\}$$

5.5.2 Codificació MaxSAT

La resolució d'un CSP reduït a SAT dóna com a resultat una assignació de les variables de la fórmula que satisfà totes les clàusules. En alguns casos aquesta solució pot no ser única i pot ser que es desitgi trobar quina és millor segons algun criteri. També és possible que no existeixi una solució per un problema SAT però que, en cas que es poguessin relaxar les restriccions imposades i es pogués violar alguna clàusula, sí existiria una solució acceptable. D'aquest tipus de problemes se'n diu problemes d'optimització. La reducció de problemes a MaxSAT permet abordar problemes d'optimització. MaxSAT és una variant de SAT on l'objectiu és trobar una assignació que satisfaci el màxim nombre de clàusules.

Existeixen diverses extensions de problemes MaxSAT. Una d'elles és *Weigthed Partial* MaxSAT. Aquesta consisteix en assignar un pes a algunes clàusules. El pes representa el cost que comporta no satisfer-la. Per tant, en aquesta variant, hi ha clàusules sense pes (*hard clauses*) i amb pes (*soft clauses*). Les *hard clauses* és obligatori satisfer-les mentre que de les *soft clauses* se'n vol satisfer un conjunt que permeti que el cost de les clàusules no satisfetes sigui mínim o, dit d'una altra manera, que la suma de costos de les clàusules satisfetes sigui màxim.

Una altra variant és *Weigthed* MaxSAT. Aquest segueix el mateix principi que *Weigthed Partial* MaxSAT, però només permet *soft clauses*. Per tant, les *hard clau-*

[2] En particular, els problemes CSP de domini finit.

ses es codifiquen com a *soft clauses* amb un pes suficientment alt perquè siguin d'obligat compliment [Biere et al., 2009]. A continuació es mostra un exemple de *Weigthed* MaxSAT, on s'anoten els pesos amb un símbol "@":

$$\begin{array}{rl} x_1 & @4 \\ \neg x_1 \vee \neg x_2 & @2 \\ x_2 \vee \neg x_1 & @1 \end{array}$$

En aquest exemple no es poden satisfer totes les clàusules. Per tant, la solució òptima és $\{x_1 \rightarrow true,\ x_2 \rightarrow false\}$ i té un cost de 1, ja que no s'ha satisfet la tercera clàusula.

CAPÍTOL 6

Requisits del sistema

6.1 Funcionals

Donat que aquest treball era la continuació d'un projecte en desenvolupament actiu al grup de recerca LIA, els requisits funcionals es van determinar conjuntament amb els membres del grup. Els requeriments inicials establerts van ser que un usuari pogués:

- Modelar CSPs mitjançant el llenguatge BUP d'una manera convenient a SAT i MaxSAT.
- Definir i utilitzar predicats definits per l'usuari amb una sintaxi senzilla. Els predicats podrien rebre qualsevol tipus de dades per paràmetre, a més de suportar referències creuades, definició de variables locals, recursivitat i sobrecàrrega segons la signatura.
- Utilitzar altres *solvers* (a més dels suportats fins llavors per GOS).
- Obtenir informació útil per entendre el procés de resolució de les fórmules per donar-li aplicabilitat docent.
- Obtenir la solució a un problema en un temps raonable.
- Obtenir la codificacio en DIMACS de la fórmula generada.
- Rebre missatges d'error convenients per entendre els possibles errors de programació comesos.
- Utilitzar GOS per la docència.

A més d'aquests requisits orientats a la usabilitat per part de l'usuari, també es van establir els següents requeriments que concretaven detalls d'implementació:

- Identificar i arreglar el problema que ocasionava que el sistema GOS trigués una quantitat de temps no raonable en resoldre certes instàncies.

- Corregir el problema que provocava que una execució llarga de GOS fes alentir el sistema operatiu o ocupés massa memòria.

- Desacoblar el procés de *solving* de GOS per aportar més flexibilitat al sistema.

- Poder escollir el *solver* a utilitzar mitjançant arguments en la crida de l'executable GOS.

- Poder optimitzar amb el *Girona Optimization System* mitjançant la definició de *soft constraints* amb una sintaxi senzilla. Només es podria anotar pesos a clàusules independents, no es realitzaria reificació ni cap procés addicional intern que modifiqués la fórmula per no ocultar res a l'usuari.

- Obtenir amb la codificació en DIMACS informació rellevant del model escrit amb BUP.

- Fer que GOS fos un sistema més estable i complet per tal que es pogués aplicar en situacions reals.

Addicionalment, durant el desenvolupament del projecte, com a resultat de les reunions SCRUM (veure 3) amb el grup de recerca, van sorgir alguns requeriments més específics:

- Definir predicats en fitxers externs i poder-los incloure en el codi del model mitjançant sentències d'inclusió. Les inclusions podrien ser en cadena, és a dir, un fitxer inclòs podria incloure altres fitxers. A més, s'hauria d'evitar incloure el mateix fitxer dues vegades.

- Necessitat d'un operador per obtenir la mida d'una estructura de dades de BUP. Això és perquè dins un predicat no es pot conèixer les dimensions d'un vector.

- Poder realitzar anotacions en forma de comentaris en la codificació en DIMACS de la fórmula amb finalitats docents i de *debug*.

- Obtenir en la codificació DIMACS la correspondència de les variables declarades amb les mostrades en la codificació de la fórmula.

- Operador *lor* i *land* per poder aplicar els operadors lògics *or* i *and* a tots els elements d'una llista.

- Correcció de la prioritat en l'avaluació de les expressions en BUP.

6.2 No funcionals

Els requeriments no funcionals van mantenir-se amb els ja definits en el projecte GOS. Per tant, els requisits de *hardware* serien:

- Poder-se compilar i executar en sistemes operatius Windows moderns i qualsevol distribució Linux prou popular.
- Consumir pocs recursos del maquinari per tal que fos utilitzable en la gran majoria de dispositius.

Els requeriments de *software* serien més que els requerits pel sistema GOS inicial, donat que s'afegirien noves funcionalitats i es requeriria programari per analitzar el rendiment de l'aplicació:

- *Another Tool for Language Recognition 4* (ANTLR4)
- CMake
- *GNU Compiler Collection* (GCC)
- Valgrind
- hotspot
- perf
- SMTAPI del grup de recerca LIA.
- MiniSat
- Glucose
- OpenWbo

CAPÍTOL 7
Estudi i decisions

Donat que l'objectiu d'aquest treball és la millora i ampliació del Girona Optimization System i el seu llenguatge propi BUP, es tracta d'una continuació d'un projecte i no pas de la creació d'un nou. Per aquesta raó, s'han procurat respectar les decisions de disseny i el codi existent el màxim possible. Això provoca que gran part dels components del sistema s'hagin mantingut respecte a la versió anterior. En els següents apartats s'expliquen els principals llenguatges, eines i llibreries que s'ha utilitzat per desenvolupar el treball.

7.1 C++

El llenguatge de programació C++ va ser creat per Bjarne Stroustrup l'any 1970 com a treball de la seva tesis doctoral. Està basat en el llenguatge C, però incorpora un canvi de paradigma respecte al seu predecessor, la orientació a objectes. Els trets més representatius del C++ són [Albatross, nd]:

- És un llenguatge compilat (veure 5.1.2), pel ofereix una ràpida execució.
- Té un tipatge fort, cosa que facilita el control d'errors, però també ofereix mecanismes per alterar-ne el comportament.
- És un llenguatge obert i estandarditzat.
- Àmpliament utilitzat en ambients professionals.
- Disposa d'un gran ventall de llibreries.
- Ofereix al programador gran flexibilitat i control de l'execució i recursos.

Un dels trets que diferencia al C++ de llenguatges del mateix estil és el darrer punt comentat, la seva gran flexibilitat. Per poder oferir aquest control al programador, s'estalvia moltes verificacions que passen a ser responsabilitat de qui implementa el programa. Un dels casos més notables és la gestió de la memòria.

En els darrers estàndards s'han incorporat noves funcionalitats al llenguatge que faciliten l'ús del llenguatge i afegeixen noves funcionalitats. En aquest projecte s'utilitza la versió 17, ja que incorpora la llibreria `filesystem` per una

fàcil gestió dels directoris i fitxers del sistema. A més, tot i que ja existia des de versions anteriors, incorpora els *smart pointers* per una gestió convenient de la memòria.

7.1.1 Gestió de memòria

Els programes, generalment, s'executen dins un context o àmbit d'execució que, en els sistemes operatius moderns, prenen el nom de processos. Un procés posa a disposició de l'executable (programa compilat) els recursos necessaris pel correcte funcionament. Un d'aquests recursos és la pila o *stack* del procés, un espai de memòria exclusiu perquè el programa hi guardi les dades i instruccions necessàries per l'execució.

El llenguatge C++ permet utilitzar el *stack* (memòria estàtica) per guardar-hi variables o constants amb un temps de vida limitat. Aquest temps està determinat per l'àmbit on es declara la variable o constant en qüestió. Addicionalment ofereix la possibilitat d'utilitzar la pila del sistema o *heap* (memòria dinàmica), un mecanisme per evitar aquesta limitació del temps de vida.

El problema d'utilitzar el *heap* és que la responsabilitat de la gestió de la memòria passa a mans del programador. Si es reserva memòria a la pila del sistema (amb l'operador `new`) cal alliberar-la explícitament (amb l'operador `delete`). En contraposició, si s'utilitza el *stack* la memòria queda alliberada un cop es finalitza l'àmbit d'execució.

La gestió de la memòria estàtica resulta molt ràpida. Per exemple, declarar o esborrar una variable al *stack* implica només incrementar o decrementar el *stack pointer* de la pila del procés. En canvi, la gestió de la memòria dinàmica és molt més costosa, ja que és un espai de memòria on poden accedir tots els processos del sistema, possiblement de manera concurrent, i cal assegurar la consistència de les dades.

Si es fa un ús inapropiat del *heap* es poden produir pèrdues de memòria o *memory leaks*. Un exemple seria reservar memòria a la pila del sistema sense esborrar-la quan ja no és necessària. Això fa que el procés cada cop ocupi més memòria innecessàriament causant alentiment i, en els pitjors dels casos, provocar que el sistema es quedi sense memòria, entri en estat de *thrashing* i col·lapsi.

7.2 Valgrind

Valgrind és un conjunt d'eines que serveixen trobar les pèrdues de memòria en un programa i analitzar el rendiment d'aplicacions. L'eina per defecte (i la més utilitzada) és Memcheck, la qual analitza els *memory leaks* d'un procés en

execució. Una pèrdua de memòria o *memory leak* es produeix per un error de programació o una mala gestió de la memòria del procés (veure 7.1.1). Aquests errors es troben potencialment en aplicacions programades amb llenguatges que ofereixen a l'usuari la possibilitat de gestionar la memòria com, per exemple, C/C++. Altres llenguatges com C# o Java no tenen aquest problema, ja que disposen d'un sistema anomenat *Garbage Collector* que s'encarrega d'alliberar la memòria que ja no està en ús.

Aquesta aplicació treballa directament amb els executables i està dissenyada per ser el menys intrusiva possible. Per fer-la servir no és necessari modificar cap part del programa que estem analitzant, només cal compilar el programa amb prou informació de *debug* per poder seguir la traça de les operacions de gestió de memòria. Aquesta informació la consumeix l'eina Memcheck. Amb els compiladors *gcc* o *g++* aquesta informació s'aconsegueix compilant amb l'opció `-g` [Valgrind, ndb]. Per exemple, la següent comanda compilaria un programa amb la informació requerida:

```
1  gcc -g [nom del nostre codi font.cpp] -o [nom del nostre programa]
```

7.3 Massif

Massif és en una de les eines que inclou Valgrind (veure 7.2). Consisteix en un analitzador de la pila del sistema o *heap*, tot i que també és capaç de recollir estadístiques de la pila del procés o *stack*. Ofereix a l'usuari informació sobre la quantitat de memòria que utilitza un programa al llarg de la seva execució [Valgrind, nda].

7.4 perf

El *profiler perf* consisteix en un analitzador dels *performance counters* o comptadors de rendiment de la CPU. Aquests comptadors són registres que compten els *events* o successos de *hardare* que succeeixen dins la CPU durant l'execució d'un programa. Alguns dels *events* que es poden enregistrar són les fallades de memòria *cache* o quines instruccions s'han executat. Aquesta aplicació ajuda a l'usuari a identificar problemes de rendiment en el codi capturant informació sobre l'execució en determinats punts del programa [Perf Wiki, 2022].

7.5 gdb

El *GNU Project Debugger* (GDB) és una eina que serveix per analitzar i inspeccionar el comportament d'un programa durant la seva execució o saber què estava fent en el moment en què es produeix un error. Permet quatre funcionalitats principals [GDB Developers, nd]:

- Executar el programa a analitzar especificant qualsevol directiva especial que pugui alterar la seva execució.
- Parar l'execució del programa en qualsevol moment utilitzant els *breakpoints*.
- Examinar l'estat del programa en el moment en què s'ha parat l'execució. La informació disponible, entre altres, és el contingut dels registres de la CPU i de la memòria o la pila de la traça d'execució o *stack trace*.
- Modificar parts del programa i canviar l'ordre d'execució de les instruccions.

7.6 CMake

CMake és un conjunt d'eines multi-plataforma i *open-source*[1] que serveixen per compilar, provar i construir paquets de programari. Va ser creat amb l'objectiu de ser un sistema per la gestió del procés de creació de *software* flexible, extensible i independent del compilador, llenguatge o sistema operatiu utilitzat.

El seu funcionament es basa en uns fitxers de configuració que descriuen com generar el *software* desitjat per cada entorn concret (per exemple, *makefiles*). Aquests fitxers permeten realitzar tot tipus de tasques com cercar fitxers, construir llibreries (dinàmiques i estàtiques) o executables, definir dependències entre projectes i fitxers, utilitzar estructures condicionals, executar comandes externes, etc [CMake, nd].

A continuació es mostra un exemple bàsic del fitxer `CMakeLists.txt`. En aquest s'hi especifica una versió del sistema CMake mínima requerida per poder executar el fitxer. Llavors es crea un projecte anomenat `Tutorial` i s'especifica que es troba a la versió `1.0`. Finalment s'afegeix el fitxer de codi font `tutorial.cxx` al projecte.

```
1  cmake_minimum_required(VERSION 3.10)
2
3  # set the project name
4  project(Tutorial VERSION 1.0)
```

[1] Tipus de *software* que pot ser llegit, editat i distribuït per qualsevol.

```
5
6 # add the executable
7 add_executable(Tutorial tutorial.cxx)
```

7.7 ANTLR4

El ANTLR4 (*ANother Tool for Language Recognition*) és la versió número 4 d'una eina utilitzada per generar analitzadors lèxics, sintàctics, semàntics i generadors de codi capaços de compilar, traduir, llegir o executar text estructurat o fitxers binaris [Parr, nd]. L'eina utilitza fitxers amb extensió `.g4` on s'hi especifica la gramàtica del llenguatge.

7.7.1 Analitzador lèxic

Per llegir un text estructurat i extreure'n el *token stream* (veure 5.1.3.1) s'ha d'especificar com identificar els diferents tipus de *tokens*. Per fer-ho, ANTLR permet sentències de l'estil `Ident_token : expr_reg`, on `ident_token` és un nom (començat per majúscules) per identificar el tipus de *token* i `expr_reg` és l'expressió regular (veure 5.2) que especifica el patró per reconèixer el *token*. La sintaxi de les expressions regulars a ANTLR es mostra a la figura 7.1.

Expressió	**Interpretació**
'c'	Reconeix el caràcter c
(e)	Reconeix e
e_1 e_2	Reconeix e_1 seguida de e_2
$e_1 \| e_2$	Reconeix o bé e_1 o bé e_2
'c_1'..'c_2'	Reconeix tots els caràcters amb codi ASCII entre c_1 i c_2 (inclosos)
~'c'	Reconeix un caràcter diferent de c
~('c_1'|'c_2'|'c_3')	Reconeix un caràcter diferent de c_1, c_2 i c_3
~('c_1'..'c_2')	Reconeix un caràcter diferent de 'c_1'..'c_2'
e?	Reconeix e o res
e*	Reconeix zero o més ocurrències de e
e+	Reconeix una o més ocurrències de e
'*abcde*'	Reconeix l'string abcde
'\''	Reconeix apòstrof
'\n'	Reconeix línia nova
'\r'	Reconeix retorn de carro
'\t'	Reconeix tabulador
' '	Reconeix espai
EOF o '@'	Final de fitxer
.	Qualsevol caràcter

Figura 7.1: Sintaxi de les regles lèxiques en ANTLR [Suy, 2022]

A més, permet funcionalitats addicionals com descartar-ne els innecessaris amb la instrucció `skip` per evitar que s'enviïn a l'analitzador lèxic. També ofereix la possibilitat de definir `fragment` s, que consisteixen en sub-expressions regular que es poden utilitzar en altres regles lèxiques per poder generalitzar parts d'expressions comunes.

Si adaptem l'exemple de l'apartat 5.1.3.1 sobre l'assignació d'una variable en pseudocodi, les regles lèxiques expressades en ANTLR serien:

```
1  fragment DIGIT : '1'..'9';
2  fragment ZERODIGIT : '0'..'9';
3  fragment CHAR : 'a'..'z' | 'A'..'Z';
4
5  TK_WS    : ( ' ' | '\t' | '\n' | '\r' ) -> skip;
6  NUM      :  (DIGIT (DIGIT|'0')*) | '0';
7  PC_VAR   :  'var';
8  DOSPUNTS :  ':';
9  PC_BOOLEA : 'boolea';
10 ASSIGN   :  ':=';
11 OP_SUM   :  '+';
12 OP_MUL   :  '*';
13 OP_IGU   :  '=';
14 PUNTCOMA :  ';';
15 IDENT    :  CHAR (CHAR DIGIT)*;
```

Cal destacar que la regla corresponent a `IDENT` s'ha mogut al final de l'especificació de l'anàlisi lèxic. Això és perquè ANTLR intenta reconèixer els *tokens* provant les regles lèxiques segons l'ordre en què han estat definides. Per exemple, la paraula `boolea` , és reconeguda per la regla `PC_BOOLEA` i `IDENT` , però com la regla `PC_BOOLEA` apareix abans, serà identificada com a token de tipus `PC_BOOLEA` .

A més, en cas que una regla reconegués un subconjunt d'una altra regla, ANTLR sempre reconeix el *token* més llarg possible. Per exemple, la paraula `variable` podria ser reconeguda com a `PC_VAR IDENT` o `IDENT` , però ANTLR es quedaria amb la segona opció [Suy, 2022].

7.7.2 Analitzador sintàctic

Per especificar les regles sintàctiques (veure 5.1.3.2) de la gramàtica s'utilitzen sentències de l'estil `ident_regla : expr1 | expr2 ...` on `ident_regla` és el nom de la regla de producció (començat amb minúscula) i `exprN` són les diferents estructures que pot adoptar la regla de producció, separades per l'operador `|` . La sintaxi de les expressions correspon a una gramàtica lliure de context (veure 5.3) amb notació EBNF[2] es mostra a la figura 7.2.

[2]Extended Backus-Naur Form

Expressió	Interpretació
T	Reconeix el token T
(e)	Reconeix e
$e_1\ e_2$	Reconeix e_1 seguida de e_2
$e_1 \vert e_2$	Reconeix o bé e_1 o bé e_2
$e?$	Reconeix e o res
$e*$	Reconeix zero o més ocurrències de e
$e+$	Reconeix una o més ocurrències de e
EOF o '@'	Final de fitxer

Figura 7.2: Sintaxi de les regles sintàctiques en ANTLR [Suy, 2022]

Si adaptem l'exemple de l'apartat 5.1.3.2 sobre l'assignació d'una variable en pseudocodi, les regles sintàctiques expressades en ANTLR serien:

```
1  declaracio  : modif IDENT DOSPUNTS tipBasic
2  modif       : PC_VAR | PC_CONST
3  tipBasic    : PC_BOOLEA | PC_ENTER | PC_REAL | PC_CAR
4  assignacio  : declaracio ASSIGN exprRel
5  exprRel     : exprSumRest (opRel exprSumRest)*
6  exprSumRes  : exprMulDiv (opSumRes exprMulDiv)*
7  exprMulDiv  : exprNeg (opMulDiv exprNeg)*
8  exprNeg     : opNeg exprBase | exprBase
9  exprBase    : valBasic | PAREN_ESQ exprRel PAREN_DRE
10 valBasic    : NUM | CAR | REAL | BOOLEA
11 opRel       : OP_IGU | OP_DIF | OP_MAJ | OP_MEN | OP_MAJ_IGU | OP_MEN_IGU
12 opSumRes    : OP_SUM | OP_RES
13 opMulDiv    : OP_MUL | OP_DIV
14 opNeg       : OP_NEG
```

7.7.3 Analitzador semàntic i generació de codi

El resultat de l'etapa de *parsing* o anàlisi sintàctic és una estructura de dades en forma d'arbre o arbre de *parsing*. En aquest punt s'ha d'explorar l'arbre per realitzar l'anàlisi semàntic i generació de codi. Tot i que l'usuari pot fer-ho manualment, ANTLR4 ofereix dos mètodes principals per facilitar la implementació d'aquesta exploració de l'arbre [Tomassetti, Gabriele, nd]:

- **Listeners** → L'arbre es visita seguint l'algorisme d'exploració DFS (*Depth-First Algorithm*). En el moment en què es visita un node de l'arbre, s'executa una funció definida per l'usuari.

- **Visitors** → Per defecte, l'arbre es visita seguint l'algorisme DFS però l'usuari pot alterar l'ordre d'exploració mitjançant la definició de mètodes per cada regla sintàctica.

Aquestes funcions o mètodes definides per l'usuari poden estar implementades en diferents llenguatges de programació d'alt nivell. L'ANTLR4 suporta

llenguatges populars com, per exemple, C++, Java o Python.

7.8 *Solvers*

Un *solver* és un programa o peça de *software* especialitzada en la resolució de problemes expressats en alguna codificació determinada. Es solen especialitzar en determinades codificacions. En aquest treball es tracten *solvers* per SAT (veure 5.5.1) i MaxSAT (veure 5.5.2).

Els SAT *solvers* actuals es basen sobretot en algorismes de *backtracking* amb cost exponencial (en el pitjor dels casos). Això és perquè es tracta de problemes de tipus *NP-hard* i, fins a la data de redacció d'aquesta memòria, no existeixen tècniques més eficients. Tot i això, s'utilitzen altres estratègies per tractar de millorar el temps de resolució. Aquestes estratègies consisteixen en combinar mecanismes d'inferència i implementacions molt eficients del procés de resolució. Una de les implementacions que dóna millora resultats actualment és *Conflict-Driven Clause Learning* (CDCL).

Pel que fa als MaxSAT *solvers*, actualment la majoria implementen diferents estratègies per trobar una solució òptima mitjançant la resolució d'una seqüència de fórmules SAT [Li et al., 2022].

7.9 DIMACS

El format DIMACS és un mètode estàndard de representació de les dades d'entrada i de sortida d'un *solver*. Es tracta d'un format àmpliament utilitzat i amb nombroses versions, amb expansions i retalls. A continuació s'explica el format estàndard utilitzat per SAT i MaxSAT *solvers* en l'especificació del format de l'any 2019 de *MaxSAT Evaluation* [Bacchus et al., nd].

7.9.1 Entrada

El format d'entrada consisteix en una representació en ASCII d'una fórmula expressada com a CNF (veure 5.5.1). El fitxer es divideix en dos blocs principals [Banbic, 1993]:

- **Preàmbul** → Conté informació sobre la instància. Cada línia que el forma està iniciada per un caràcter clau seguit d'un espai, que identifica el tipus d'informació que conté la línia. Els tipus segons la lletra són:
  - **c** → Comentaris. Són anotacions llegibles per l'usuari i ignorades pel *solver*.

- **p** → Problema. Hi ha d'haver una sola línia d'aquest tipus per fitxer. Descriu les característiques de les clàusules. El format és:

```
1  p FORMAT NVARIABLES NCLAUSULES
```

  El `FORMAT` pot ser *wcnf* o *cnf*, depenent si les clàusules porten pesos associats o no. Els camps `NVARIABLES` i `NCLAUSES` indiquen, respectivament, el nombre de variables i el nombre de clàusules que té la instància.

- **Clàusules** → Aquest bloc ha d'aparèixer immediatament després de la línia de problema. Conté les clàusules de la fórmula (una per línia i acabades en 0) en format CNF o WCNF. Les variables que les conformen estan codificades amb un enter entre 1 i `NVARIABLES` i separades per espais. En cas que la variable estigui negada, porta un signe negatiu a davant. Si el format és WCNF, a cada línia i davant de la clàusula s'hi indica el pes corresponent a la clàusula.

7.9.2 Sortida

El format de sortida és molt semblant al d'entrada, però amb tipus de línies diferents. A continuació s'indiquen els tipus de línies que apareixen en un fitxer de sortida segons la lletra que les identifica [MaxSAT Evaluations, 2021]:

- **c** → Comentaris. Poden utilitzar-se per expressar informació addicional no relacionada directament amb la solució (com estadístiques del procés de *solving*).
- **s** → Solució. Resultat del procés de *solving*. Pot tenir els següents valors, depenent si es tracta d'un problema MaxSAT (WCNF) o SAT (CNF):
  - **MaxSAT**
    - **OPTIMUM FOUND**
    - **UNSATISFIABLE**
    - **UNKNOWN**
  - **SAT**
    - **SATISFIABLE**
    - **UNSATISFIABLE**
    - **UNKNOWN**
- **o** → Cost de la solució. Només en cas de ser un problema MaxSAT. Indica el cost de la solució òptima trobada amb un enter.

- **v** → Valors de les variables. Assignació de valors de les variables en cas de trobar solució.

CAPÍTOL 8

Anàlisi i disseny del sistema

En aquesta secció s'explica l'anàlisi del sistema existent i el disseny de les millores i ampliacions previstes. Cal destacar la gran importància de l'etapa d'anàlisi, ja que incorporar noves funcionalitats de manera correcta en un projecte existent requereix d'abundant estudi i comprensió del codi.

8.1 Estructura del programa

El sistema GOS inicial està dissenyat per poder resoldre instàncies de problemes codificats amb el llenguatge propi BUP. Tal com es mostra a la figura 8.1, el sistema rep com a entrada el model o codificació del problema descrit amb llenguatge BUP i les dades de la instància en un fitxer JSON. Llavors, el compilador de BUP processa ambdós fitxers per generar una fórmula SAT utilitzant l'API del grup del grup de recerca LIA. A continuació aquesta API pot o bé mostrar per pantalla la fórmula en format DIMACS o passar-la a un SAT *solver* integrat. En el darrer cas, un cop resolta la fórmula pel *solver*, s'acaba mostrant el resultat del programa per pantalla.

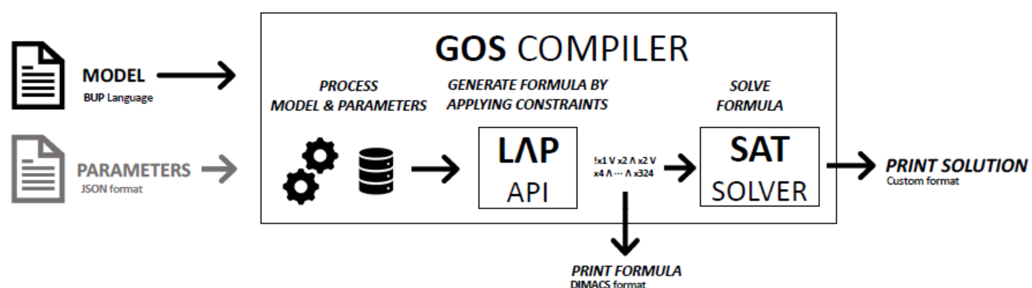

Figura 8.1: Esquema general de funcionament del sistema GOS inicial [Generoso, 2020]

Així doncs, el sistema està compost per tres components principals:

- Compilador del llenguatge BUP
- API del grup del grup de recerca LIA
- API dels *solvers* inclosos en el sistema

8.1.1 Compilador BUP

El compilador rep com a entrada el model d'un problema CSP codificat en BUP i les dades corresponents a una instància concreta especificades en format JSON. La seva tasca és traduir aquesta informació a una formula CNF (veure 5.5.1), que seria l'equivalent al codi intermedi d'un compilador convencional (veure 5.1.3.2). Llavors el sistema permet acabar la compilació i generar un fitxer en format DIMACS o també pot passar la fórmula a un *SAT solver* per resoldre-la i mostrar-ne el resultat.

Tal com es mostra al diagrama de la figura 8.2, el procés de compilació comença amb l'entrada del model escrit amb llenguatge BUP i les dades en format JSON. Primer es llegeixen les dades seguint un procés de compilació parcial. És a dir, donada la gramàtica corresponent a un text estructurat en format JSON, el sistema GOS genera l'arbre de *parsing* corresponent al fitxer de la instància. Llavors recórre l'arbre utilitzant el *visitor* `GOSJSONInputVisitor` per generar una estructura de dades equivalent que es pugui tractar en C++.

Un cop llegida la instància es passa a compilar el model. L'estratègia que es segueix és generar un arbre de *parsing* del codi BUP d'entrada i visitar-ne el subarbre requerit. Es visiten, per ordre, el bloc de *viewpoint* i *constraints* per generar una `SMTFormula`. Si s'ha escollit mostrar la fórmula, es codifica en DIMACS mitjançant l'API i es mostra per pantalla. Altrament, l'API la resol amb el *solver* especificat. Llavors, en cas d'haver especificat un bloc de *output* en el fitxer BUP, es genera el text que a mostrar visitant el bloc d'*output*. Altrament es mostra que la fórmula és satisfactible.

8.1.2 API LIA

Per la manipulació de fórmules CNF, la codificació a DIMACS i la comunicació amb els *solvers* s'utilitza la llibreria del grup de recerca LIA, la qual ofereix una API molt completa anomenada SMTAPI. Cal destacar que aquesta llibreria suporta més funcionalitats que les necessàries per aquest projecte, com la definició de clàusules SMT.

Per utilitzar aquesta llibreria, primer cal definir un *encoding*. Per fer-ho, s'ha de crear una classe que implementi els mètodes de la classe abstracta `Encoding`, la qual defineix les funcions necessàries per operar amb una fórmula. En el cas de GOS, aquesta classe s'anomena `GOSEncoding`.

Llavors cal crear una instància d'una `SMTFormula` i emplenar-la de clàusules per poder instanciar el *GOSEncoding*. A continuació, el `BasicController` s'encarrega de configurar el procés de resolució de la fórmula segons els paràmetres entrats i l'executa.

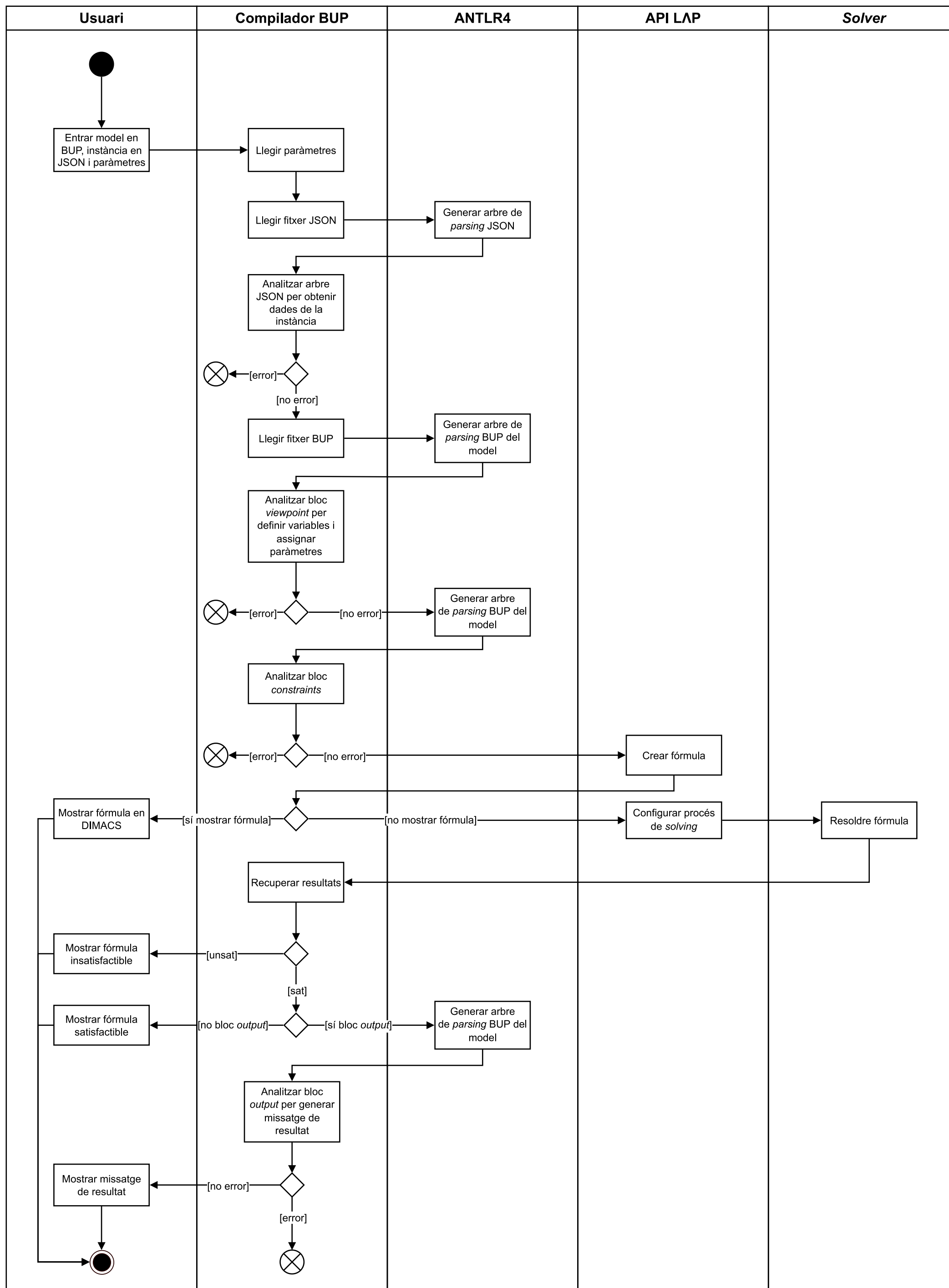

Figura 8.2: Diagrama de flux del funcionament del sistema GOS inicial

8.1.3 *Solvers*

L'API suporta per defecte la interacció amb dos *solvers*: minisat i glucose. En l'especificació de la llibreria s'indica que n'hauria de suportar més, però no estan implementats.

8.2 Correccions, millores i ampliacions

8.2.1 Escalabilitat

Durant l'etapa d'estudi i comprensió de la implementació del sistema GOS inicial, vaig observar que el codi font del programa estava contingut en diferents fitxers amb extensió `.h` i en `.g4`. Els primers corresponen a la implementació en C++ de l'anàlisi semàntic i generació de codi dels compiladors dels llenguatges BUP i JSON. Els segons són la descripció de la gramàtica dels dos llenguatges amb la sintaxi de ANLTR4.

Pel que fa a la implementació en C++, tot el codi estava escrit en fitxers capçalera o *headers* (`.h`), és a dir, no s'utilitzava l'esquema típic de separar en fitxers capçalera i implementació la declaració de la implementació per les classes i mètodes (`.h` i `.cpp`). Utilitzant aquesta aproximació s'ha de ser molt curós amb l'ús del llenguatge C++ per tal que el codi sigui utilitzable com a mòdul en altres projectes i perquè no tingui problemes d'escalabilitat.

Alguns dels problemes identificats van ser la utilització de les sentències `using namespace`. Aquesta utilitat de C++ permet fer visibles tots els identificadors d'un *namespace* en l'àmbit on s'utilitza la sentència, cosa que implica que tots aquests noms es puguin utilitzar o cridar sense haver de mencionar l'espai de noms al que pertanyen. La funcionalitat pot ser útil a l'hora de programar per estalviar-se haver d'escriure l'identificador de l'espai de noms cada vegada que s'utilitza algun membre del *namespace*. Tot i que pot ser útil en fitxers d'implementació `.cpp`, es considera mala pràctica fer-la servir en fitxers *header* `.h`. Això és perquè qualsevol usuari que inclogui un fitxer en un projecte amb la directiva `#include` també estarà fent involuntàriament tots els `using namespace` que hi hagi en aquell fitxer capçalera. Per exemple, suposem que un usuari de l'aplicació GOS té el fitxer `myVector.h` en el seu projecte, on s'implementa una classe amb nom `vector`. Llavors, en la implementació hauria d'incloure el fitxer `GOSCompiler.h` i el fitxer `myVector.h`. Com que en el primer fitxer s'utilitza la sentència `using namespace std;`, s'estaria creant un conflicte d'identificadors (ambigüitat) entre la classe `vector` de l'usuari i la classe `std::vector`. Aquests problemes poden ser un inconvenient que, en

el millor dels casos, pot fer fallar la compilació del programa de l'usuari.

Relacionat amb el problema anterior vaig trobar que no s'utilitzava cap *namespace* per contenir tots els identificadors relatius a l'aplicació GOS. Això pot causar problemes semblants als descrits en el paràgraf anterior.

Un altre inconvenient identificat va ser la utilització d'identificadors sense haver inclòs la capçalera on estan definits. Un exemple seria la utilització de la classe `string` sense haver escrit prèviament en el fitxer `#include <string>`. La omissió d'aquests *includes* és possible gràcies a que en alguns dels fitxers inclosos ja hagi estat inclòs el *header* on l'identificador està definit. El problema és si en algun moment del desenvolupament, ampliació o manteniment de l'aplicació es canvia l'estructura de fitxers, causant que canvii la cadena d'*includes* i que no compili l'aplicació.

Les correccions que es durien a terme serien solucionar els problemes amb els *namespaces*, esborrant totes les sentències `using namespace` i utilitzant *fully qualified names*[1] per referenciar els identificadors. Cal destacar que una altra manera de solucionar aquest problema podria haver estat reescriure tota l'aplicació per seguir l'estructura capçalera-implementació però aquesta idea es va descartar per la gran quantitat de feina que comportava i, a més, perquè l'estructura amb *headers* no resulta incorrecta si s'utilitza de manera adequada. També s'estructuraria en *namespaces* tota la implementació relacionada amb el sistema GOS. Finalment s'inclourien a cada fitxer les capçaleres necessàries per tots els identificadors utilitzats.

8.2.2 Rendiment

Un dels requeriments del projecte era identificar quin era el problema de que causava que el sistema GOS trigués una quantitat de temps exagerada en certes instàncies. També, de manera més global, es pretenia millorar si era possible el rendiment general de l'aplicació.

Per trobar el problema de rendiment, en primer lloc vaig estudiar el codi i la implementació del sistema. Així va ser com vaig adonar-me que es feia un ús abundant (i a vegades innecessari) del *heap*. Tal com s'explica a la secció 7.1.1, l'ús del *heap* és força costós en quant a temps de còmput. Per tant, vaig voler esbrinar si aquest ús abusiu de la memòria dinàmica estava alentint el programa.

Utilitzant l'eina Massif (veure 7.3) vaig analitzar l'ús del *heap* i el *stack* durant una execució de curta durada, ja que la utilització d'aquest tipus d'eines incrementa el temps d'execució considerablement. Concretament vaig utilitzar el model `nonogram.bup` i la instància `nono_duck.json`. Per poder analitzar

[1] Identificador precedit pel nom del *namespace* on es troba definit. Per exemple *std::string*.

l'ús de memòria correctament, vaig haver de compilar l'aplicació en mode *debug*. La comanda executada va ser:

```
1  valgrind --tool=massif --stacks=yes ./CSP2SAT ../input/nonogram.bup ../input/
       nono\_duck.json
```

Un cop acabada l'execució es va generar un fitxer de resultats que, amb l'ajut de l'eina `ms_print`, vaig extreure'n les estadístiques. Les dades obtingudes estan representades a la figura 8.3, on es mostra la quantitat de memòria ocupada pel programa al llarg de tota l'execució i quina aportació tenen el *heap* i el *stack* en aquest total. Com es pot observar, l'ús de la memòria local o *stacks* és mínim comparat amb l'ús del *heap* o memòria dinàmica.

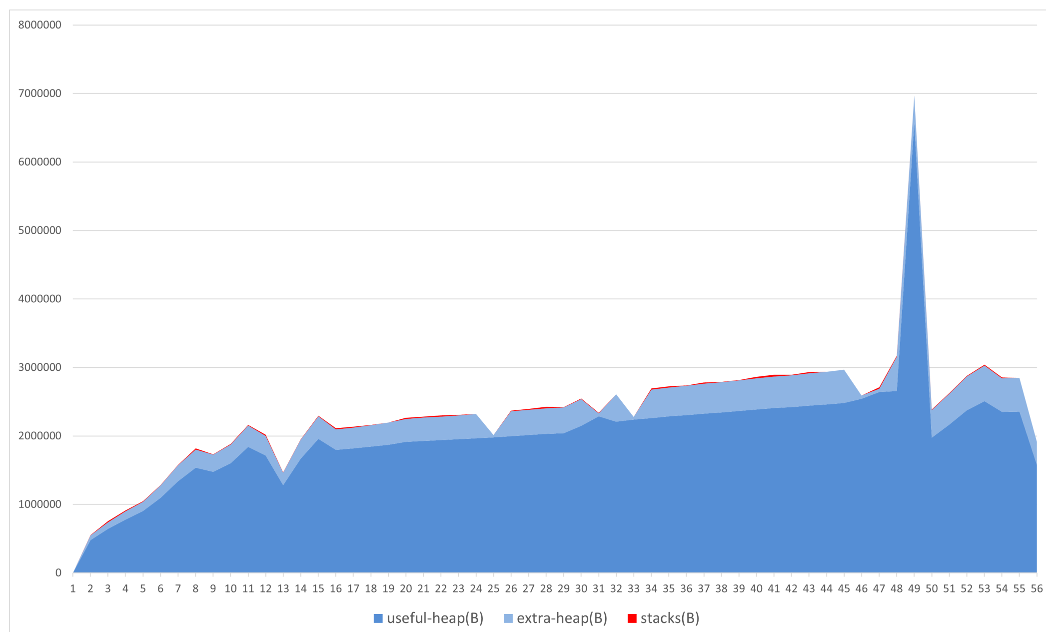

Figura 8.3: Evolució de la memòria total ocupada al llarg de l'execució del sistema GOS pel model nonogram.bup i la instància de nono_duck.json

Observant aquests resultats vaig plantejar-me la possibilitat que es fes un ús incorrecte de la memòria dinàmica. Per investigar aquest fet vaig utilitzar l'eina Valgrind (veure 7.2) per analitzar si el compilador GOS tenia *memory leaks* (veure 7.1.1). La comanda executada va ser:

```
1  valgrind --leak-check=full ./CSP2SAT ../input/nonogram.bup ../input/nono_duck
       .json -pf
```

El resultat obtingut un cop finalitzada l'execució va ser:

```
HEAP SUMMARY:
    in use at exit: 1,030,934 bytes in 21,130 blocks
  total heap usage: 482,490 allocs, 461,360 frees, 25,934,030 bytes allocated

LEAK SUMMARY:
   definitely lost: 368,392 bytes in 7,304 blocks
   indirectly lost: 662,262 bytes in 13,821 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 280 bytes in 5 blocks
        suppressed: 0 bytes in 0 blocks
```

Per tant, vaig determinar que sí hi havia força problemes de *memory leaks*. Mitjançant l'experimentació amb instàncies més grans i seguint la traça de l'ús de memòria del sistema durant les execucions vaig esbrinar que aquesta no era la causa principal del problema de rendiment perquè en cap moment s'exhauria la memòria.

Tot i que no es tractava de la raó principal del problema del temps de càlcul excessiu en determinades instàncies, sí que es tracta d'un problema greu de rendiment. Per això vaig decidir que, per minimitzar els *memory leaks*, caldria substituir els *raw pointers* per *smart pointers* (veure 7.1.1). Addicionalment, vaig observar que no estaven ben implementats els destructors virtuals dels objectes. Això provocava que no s'alliberés completament la memòria dinàmica reservada en cas d'invocar el destructor d'un objecte utilitzant un punter a una classe base. Per tant, caldria incorporar els destructors virtuals a les classes bases en les jerarquies de classes pertinents. Cal destacar que la millor opció hauria estat reescriure el compilador per tal de minimitzar l'ús del *heap* i resoldre tots els *memory leaks*, però això hauria requerit molt temps i quedava fora dels objectius del projecte.

Encara no havia trobat l'arrel del problema indicat als requeriments. Per aquesta raó vaig optar per utilitzar l'eina *perf* (veure 7.4) per veure en què s'invertia el temps durant l'execució i detectar algun possible error d'optimització del codi. La comanda executada va ser:

```
perf record -g ./CSP2SAT ../input/nonogram.bup ../input/nono_duck.json
```

Llavors, amb l'ajut de l'eina *hotspot* vaig representar les dades de manera gràfica per poder analitzar-les més fàcilment. Així vaig esbrinar que en cap part de l'execució es penjava el programa o es dedicava una quantitat de temps no raonable.

Finalment vaig decidir resseguir la traça d'execució amb l'eina *gdb* (veure 7.5). Mitjançant l'anàlisi manual de les crides i temps d'execució de cada funció vaig esbrinar que l'error estava a la implementació de l'API LIA, concretament en la implementació de l'API dels *solvers*. Per tant, caldria identificar l'arrel del problema i resoldre'l.

8.2.3 Predicats

La primera ampliació del sistema GOS consistia en ampliar les característiques del llenguatge BUP permetent que l'usuari definís predicats personalitzats i els pogués utilitzar per generalitzar parts de la lògica del seu model. Addicionalment, un requeriment relacionat amb aquesta ampliació era la possibilitat de poder incloure definicions de predicats fetes en altres fitxers.

Respecte a la ampliació de BUP per permetre predicats, l'usuari havia de poder definir predicats amb una sintaxi senzilla. A més, hauria de poder incloure fitxers amb definicions de predicats. La proposta va ser definir aquest predicats i *includes* dins un nou bloc opcional al fitxer del model, identificat per la paraula clau `predicates`, seguint el disseny de BUP organitzat en blocs. La sintaxi d'un predicat (esquerra) i d'una inclusió de fitxer extern (dreta) seria:

```
1  <ident>(<param>*) {
2      <decl_var_aux>*
3      <constraint>+
4  }
```

```
1  include "nom_fitxer.bup";
```

El bloc de predicats tindria la següent sintaxi:

```
1  predicates:
2      (<definicio_predicat> | <include>)*
```

Així doncs, un fitxer de model BUP passaria a tenir la següent estructura:

```
1  <bloc_entitats>?
2  <bloc_viewpoint>
3  <bloc_predicats>?
4  <bloc_constraints>
5  <bloc_output>?
```

Un predicat havia de permetre rebre cap, un o més d'un paràmetre de qualsevol tipus. Aquest paràmetres s'expressarien seguint una sintaxi molt semblant a la declaració de variables i paràmetres del bloc de *viewpoint*. També havia de permetre fer declaracions de variables locals auxiliars. El cos del predicat havia de contenir un o més *constraints* amb la mateixa sintaxi que s'utilitza al bloc de *constraints*. Aquestes decisions de disseny es van fer procurant que la nova notació fos molt semblant a la sintaxi de BUP existent, per qüestions de facilitat d'ús.

La crida a un predicat hauria de fer-se indicant l'identificador del mateix i, entre parèntesis, els paràmetres pertinents amb el tipus esperat. Com que s'havia de suportar la sobrecàrrega, els predicats es diferenciarien segons la seva signatura, formada per el nom i el tipus dels paràmetres.

Els predicats definits s'havien de poder cridar des de qualsevol lloc on es permetés expressar un *constraint*. En altres paraules, una crida a un predicat podia trobar-se al bloc de *constraints*, però també en el cos d'altres predicats i, fins i tot, en el cos del propi predicat. Això volia dir que s'havia de suportar la recursivitat i les referències creuades. La primera característica s'implementaria amb àmbits d'execució locals per cada crida, on es definirien els paràmetres de la crida i les variables locals. El problema de les referències creuades s'hauria pogut resoldre exigint a l'usuari que primer declarés la signatura de tots els predicats i llavors proporcionés la implementació per cadascun però, com la notació es va idear per fer els dos passos a la vegada, aquesta opció va quedar descartada. Per aquesta raó, es podrien donar situacions en el procés de compilació en què es troba una cria a un predica abans d'haver-se'n trobar la declaració. Un exemple seria:

```
1  A(var bool x, param bool y){
2      if(y){
3          !x;
4          B(x, y);
5      }
6      else{
7          x;
8      };
9  }
10
11 B(var bool x, param bool y){
12     A(x, not y);
13 }
```

Per tant, seria necessari fer la compilació dels predicats en, mínim, dues passes: primer fer la declaració de totes les signatures i llavors compilar els cossos dels predicats.

La crida als predicats comportava un repte addicional. Això és perquè una crida hauria de generar les clàusules pertinents segons els valors dels paràmetres i les restriccions indicades al seu cos per tal d'afegir-les a la fórmula del model, de la mateixa manera que un *constraint* qualsevol. El problema és que el valor dels paràmetres d'un predicat depèn dels valors passats a cada crida. Per tant, caldria compilar el cos de cada predicat per cada crida, ja que el paràmetres passats podien ser diferents i, per tant, les clàusules generades també.

Pel que fa al requeriment de poder incloure fitxers, l'usuari hauria de poder cridar qualsevol predicat definit en un fitxer després d'incloure'l en el codi. La sentència d'inclusió es podria trobar en qualsevol lloc dins el bloc de definició de predicats, permetent cridar els predicats continguts en el fitxer a continuació de la sentència. A més, un fitxer inclòs podria, a la vegada, incloure altres fitxers, pel que calia resoldre tota la cadena d'inclusions. Això podia ocasionar algunes situacions com la de la figura 8.4, on un mateix fitxer (*globals.bup*) pot haver

estat inclòs dues vegades (tant en *a.bup* com en *b.bup*). En aquests casos s'hauria d'evitar compilar el mateix fitxer dues vegades. La noció d'incloure un fitxer seria com "copiar i enganxar" el codi contingut al fitxer, pel que s'aplicarien els mateixos controls i verificacions que als predicats definits al fitxer del model.

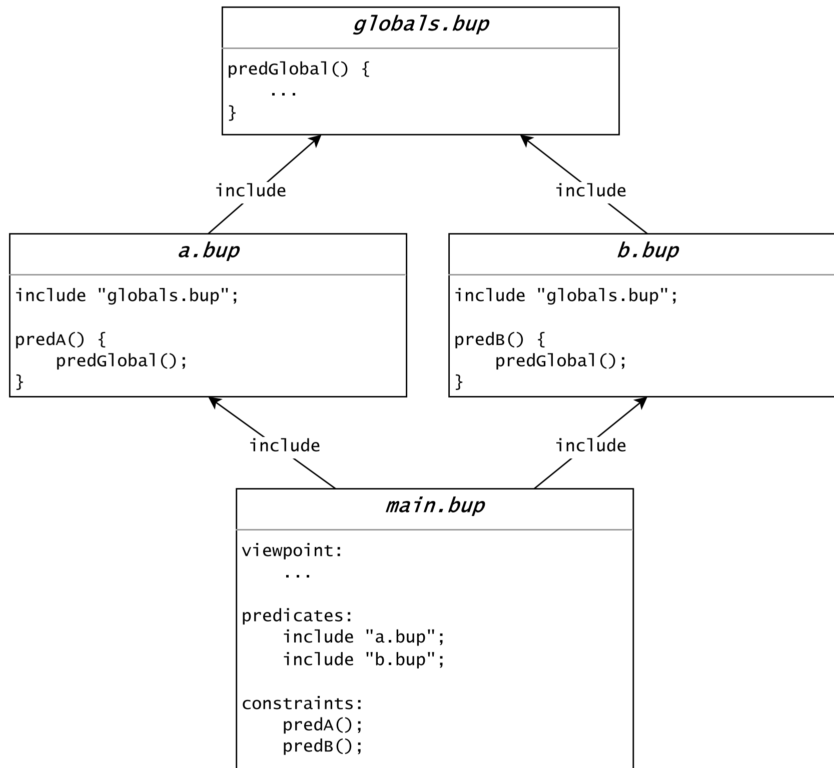

Figura 8.4: Exemple de cadena d'inclusions

Així doncs, per implementar aquesta ampliació caldria modificar la gramàtica de BUP amb noves regles lèxiques i sintàctiques per suportar la nova notació, a més de modificar la definició d'un *constraint* per permetre la crida dels predicats. Pel que fa a l'anàlisi sintàctic, s'hauria de crear un nou *visitor* que realitzés la definició de predicats i control d'errors. La generació de codi s'implementaria al *visitor* de *constraints*. La compilació dels predicats es faria en tres passes: primer es declararien totes les signatures, llavors es comprovarien els errors d'ús del llenguatge en el cos dels predicats i, per cada crida a un predicat, es compilaria el seu cos amb els paràmetres pertinents per generar les clàusules corresponents.

8.2.4 Soft constraints

El GOS o *Girona Optimization System* en un principi estava pensat per ser un sistema d'optimització però, en la seva primera versió, només permetia trobar

solucions satisfactibles, no pas òptimes. Per tant, un dels requeriments principals era implementar aquesta funcionalitat per tal de completar GOS.

Per resoldre problemes CSP amb GOS, primer s'havien de modelar en llenguatge BUP. Amb el model s'havia de proporcionar una instància en format JSON. Llavors el sistema compilava el model i cercava una assignació satisfactible a la fórmula compilada que satisfés totes les restriccions indicades. Per poder trobar la millor de les solucions possibles, calia afegir a BUP un mecanisme d'optimització.

El mètode escollit va ser implementar la possibilitat d'expressar problemes MaxSAT. Per fer-ho, BUP hauria de permetre indicar els pesos de cada clàusula (veure 5.5.2). Així doncs, BUP hauria de permetre dos tipus de *constraints*: *hard constraints* i *soft constraints*. Els primers haurien de satisfer-se tots per tal que una solució fos vàlida. En canvi, els *soft constraints* es podrien no complir, penalitzant la puntuació de la solució segons el pes assignat a la restricció.

Per implementar aquesta nova funcionalitat caldria afegir a BUP la sintaxi necessària per expressar *soft constraints*. La notació seria igual a la dels *hard constraints* però amb un indicador del pes. La sintaxi seria:

```
1  <constraint> @pes;
```

Cal destacar que no tots a tots els tipus de restriccions se'ls podria assignar un pes. Només es podria a aquells que es poden traduir en una sola clàusula. Aquesta decisió es va prendre conjuntament amb el grup de recerca LIA i consistia en una de les limitacions contemplades als requeriments. La justificació és perquè, per poder suportar assignar pesos a més d'una clàusula, s'haurien d'haver dut a terme processos interns a GOS per generar la fórmula correctament. Aquest procés podria haver estat la reificació de les clàusules amb una variable booleana. Això es va descartar perquè es volia que BUP fos un llenguatge transparent, és a dir, que l'usuari sabés en tot moment quines clàusules estava codificant. Tot i això, l'usuari podria dur a terme la reificació en cas que ho desitgés de manera manual ajudant-se, fins i tot, de la nova funcionalitat dels predicats (veure 8.2.3).

Aleshores, el compilador hauria de generar les clàusules corresponents als *hard constraints* i *soft constraints* trobats en el codi i afegir-les a una fórmula de L'API LIA, la qual ja suportava la creació de fórmules amb pesos. Llavors la fórmula s'hauria de resoldre mitjançant un *solver* que suportés MaxSat. Per aquesta raó la millora dels *solvers* (8.2.5) seria indispensable.

8.2.5 *Solvers*

El sistema GOS inicial permetia resoldre la fórmula generada pel compilador només amb el *solver* MiniSat. Aquest fet li restava en gran mesura flexibilitat al sistema i suposava un inconvenient per l'usuari perquè en cas de tenir models i instàncies que es resoldrien més ràpidament amb altres *solvers* no podria ferhi res. També, com es pretenia que GOS servís com a eina docent, seria útil poder experimentar amb la combinació de diferents *solvers* per estudiar-ne el comportament donats models i instàncies determinades, segons la tècnica de *solving* que implementés cadascun. Addicionalment, aquesta millora serviria per poder implementar al complet l'ampliació dels *soft constraints* (veure 8.2.4), on es requeria d'un *solver* que suportés MaxSat.

Fins al moment GOS suportava dos *solvers* (MiniSat i Glucose) mitjançant l'API LIA. El sistema codificava la fórmula pertinent i llavors la resolia cridant l'API específica de cada *solver*. Aquesta implementació provocava que s'hagués de codificar un adaptador per poder comunicar-se amb cada *solver* que es volgués afegir. Això resultava costós d'implementar i de mantenir. Per aquesta raó vaig decidir desacoblar l'etapa de *solving* del procés de resolució de GOS. Tot i això, es mantindria la possibilitat de cridar l'API dels *solvers* ja implementats per raons de flexibilitat. A la figura 8.5 s'il·lustra amb un diagrama de flux simplificat el funcionament del procés de *solving* després de la millora.

Per fer-ho, modificaria l'API LIA perquè pogués interactuar amb un *solver* executant-se en un procés extern mitjançant *pipes* del sistema operatiu. La transferència de la fórmula al *solver* es faria mitjançant un fitxer en format DIMACS (veure 7.9). L'elecció d'aquest format es basava en què es tracta d'un format popular i força estandarditzat, utilitzat en la majoria de competicions. La recuperació dels resultats es faria interpretant la sortida en text del *solver* en format DIMACS simplificat per les mateixes raons.

Per tal que l'usuari tingués de la possibilitat d'utilitzar *solvers* amb *pipes* sense haver de realitzar cap acció addicional, els executables dels *solvers* MiniSat i Glucose es compilarien juntament amb el sistema GOS per tal que l'usuari pogués escollir entre resoldre la fórmula amb API o sense. A més, com que la millora dels *soft constraints* (veure 8.2.4) requeria d'un *solver* que permetés MaxSat, s'afegiria també l'executable de OpenWBO. Addicionalment s'afegiria una opció per tal que l'usuari pogués utilitzar un *solver* de la seva elecció proporcionant la comanda necessària per executar-lo.

Així doncs, GOS permetria resoldre les fórmules generades cridant l'API dels *solvers* Minisat i Glucose o passant-la a un procés extern on s'estigués executant Glucose, OpenWBO o un *solver* personalitzat. Aquesta elecció es faria mitjançant paràmetres d'execució del sistema GOS.

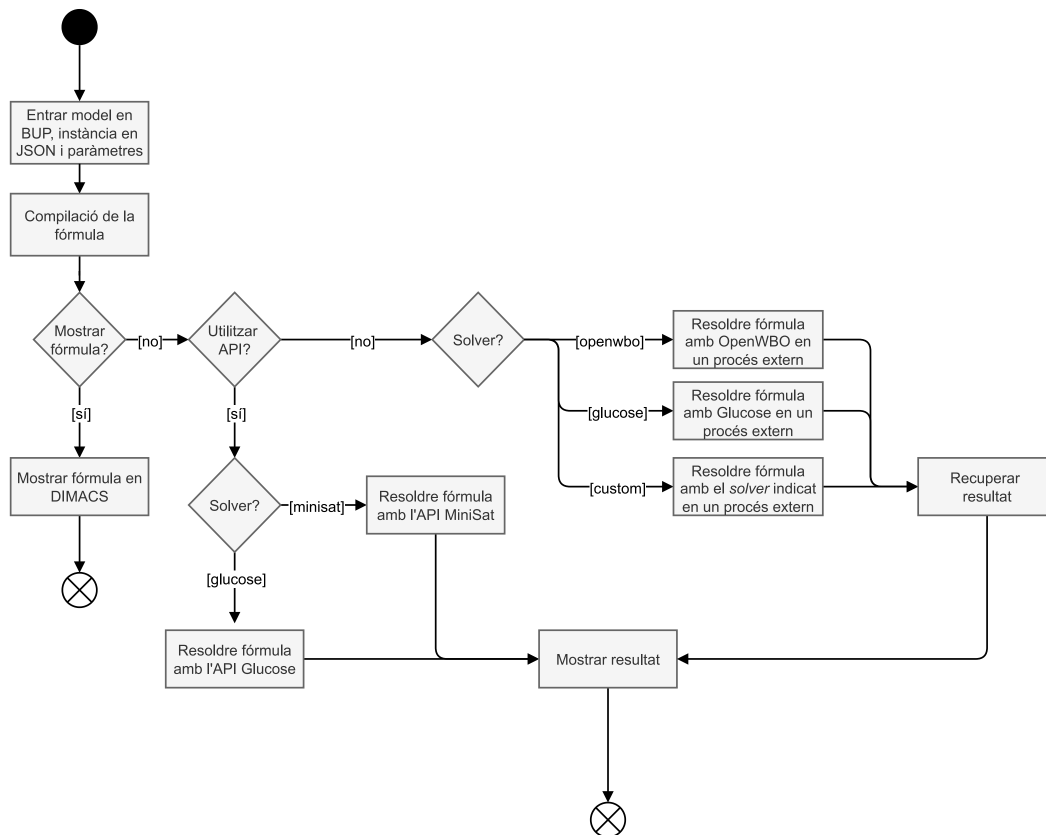

Figura 8.5: Diagrama de flux simplificat del funcionament de GOS un cop desacoblat el procés de *solving*

8.2.6 *Debug*

En els requeriments establerts pel grup de recerca LIA hi constava que GOS hauria de ser utiltzable per la docència. Per aquesta raó es demanava que l'usuari pogués visualitzar informació rellevant de *debug* en el fitxer resultant de la codificació a DIMACS.

Per tant, caldria implementar un sistema d'anotacions per poder escriure comentaris directament al fitxer DIMACS segons l'ordre d'aparició en el codi BUP. Arribant a un consens amb els membres del grup, es va determinar que la millor notació era utilitzant un comentari especial de la forma `//c <missatge>`. Aquest comentari tenia la mateixa sintaxi que un comentari ordinari de BUP, però en comptes d'ignorar-se, s'hauria de plasmar en el fitxer DIMACS. Aquesta funcionalitat permetria a l'usuari identificar les clàusules generades en determinades parts del codi.

Addicionalment, seria molt convenient poder identificar les variables representades en DIMACS. Per tant, es mostraria el valor de tots els paràmetres i els identificadors de les variables declarades al llarg del model BUP.

Aquesta informació generada era per qüestions de *debug*. Per tant, podria no ser rellevant per alguns usuaris. Per aquesta raó s'afegiria una opció per activar o desactivar la generació d'aqueta informació.

CAPÍTOL 9

Implementació i proves

9.1 Escalabilitat

Pel que fa a l'ús incorrecte dels *namespaces*, vaig incloure totes les definicions d'identificadors relatius a GOS dins un nou *namespace* anomenat GOS. Addicionalment vaig crear dos *namespaces* niuats per incloure totes les funcions d'ús general en el projecte o *utils* que s'anomenarien *VisitorsUtils* i *Utils* i substituirien les classes amb mètodes estàtics que hi havia prèviament. La substitució de les classes amb mètodes estàtics públics d'*utils* per *namespaces* es va dur a terme perquè l'ús de la orientació a objectes per contenir només funcions d'ús general no està massa justificat, sobretot quan existeix el mecanisme de *namespaces* que permet la funcionalitat desitjada, que és aïllar els identificadors.

Per tant, les funcions i accions d'*utils* que prèviament estaven dins la classe `Helpers` patirien les següents modificacions:

```
1  class Helpers {
2  public:
3  namespace GOS {
4  namespace Utils {
5      static bool check_number(const string & str) { ... }
6      static vector<string> splitVarAccessNested(string a) { ... }
7      static string toRawString(string const& in) { ... }
8  }
9  }
```

El mateix procediment vaig seguir per les funcions i accions d'*utils* relatives als *visitors* que prèviament estaven dins la classe `Utils` .

En quant a l'ús de la sentència `using namespace` en fitxers capçalera o *headers*, vaig eliminar-los tots per evitar conflictes d'identificadors. També vaig modificar tots els identificadors que pertanyien a aquests *namespaces* per la seva versió completa o *fully qualified name*. Per exemple, en els fitxers on hi havia la sentència `using namespace std;` per incloure tots els identificadors de la llibreria estàndard de C++ a l'espai de noms actual, vaig substituir tots els identificadors de tipus `string` per `std::string.`

Pel que fa a la correctesa dels *include*, vaig incloure en cada fitxer les capçaleres corresponents a tots els identificadors utilitzats. Un exemple és l'ús de la

classe *std::string*, que requereix la inclusió de la seva capçalera amb la sentència `#include <string>`.

A continuació es mostra un exemple d'implementació amb la classe `Symbol`, on hi ha totes les millores estructurals aplicades. S'han ressaltat amb color verd els elements de codi afegits i en vermell aquells que s'han esborrat.

```
1  #include <string>
2  #include <utility> // move semantics
3
4  namespace GOS {
5
6  using namespace std;
7
8  class Scope;
9  class Type;
10
11 class Symbol {
12 public:
13     Symbol(std::string name) : name(std::move(name)) {}
14     Symbol(std::string name, Type *type) : name(std::move(name)), type(type)
15     std::string getName() { ... }
16     virtual Type* getType() { ... }
17     virtual bool isAssignable() { ... }
18     virtual bool isScoped() { ... }
19
20 protected:
21     std::string name;
22     Type* type = nullptr;
23 };
24
25 }
```

Alguns detalls a destacar són la correcció de l'accés als atributs de la classe, que en comptes de ser *public*, van passar a ser *protected*. Això ho vaig fer per no trencar l'encapsulament de l'orientació a objectes i per permetre que només les classes derivades tinguessin accés als atributs de les superclasses.

9.2 Rendiment

9.2.1 Gestió de memòria

El sistema GOS tenia diversos problemes de gestió de memòria dinàmica causats per l'ús excessiu i a vegades incorrecte del *heap*. Per tal de minimitzar els *memory leaks* sense haver de reescriure el compilador, vaig decidir substituir els *raw pointers* per *smart pointers* i implementar correctament els destructors virtuals.

Per fer la refacció[1] dels punters vaig inspirar-me en el patró *factory*, és a dir, vaig controlar la creació d'instàncies evitant que es poguessin cridar els cons-

[1] Alteració o reescriptura del codi per millorar-ne la funcionalitat, llegibilitat o estructura sense alterar-ne el comportament

tructors de les classes i implementant un mètode que s'encarregués de generar i retornar els objectes. En conseqüència no es podrien crear objectes en memòria dinàmica de manera manual i seria pràcticament impossible generar un *memory leak* per una mala gestió del temps de vida dels objectes.

Així doncs, vaig canviar l'accés dels constructors a *protected* perquè des de fora la classe no es poguessin cridar, però permetent a la vegada la creació de classes derivades. Llavors vaig implementar un mètode `Create()` per cada constructor diferent. Aquest s'encarregaria de generar la instància corresponent i retornaria un *shared_ptr* que gestionés el temps de vida de l'objecte creat.

Per qüestions de conveniència vaig definir un tipus nou per cada classe en què vaig aplicar aquests canvis. Aquest tipus permetria fer referència al tipus de `shared_ptr<T>` d'una manera més senzilla, on `T` seria l'identificador de la classe en concret. La sintaxi seguida per aquests nous tipus seria `TRef`. Un exemple aplicat a la classe `Symbol` seria:

```
1   typedef std::shared_ptr<Symbol> SymbolRef;
```

Addicionalment, vaig implementar els destructors virtuals a les classes base perquè es realitzés la destrucció completa dels objectes derivats.

A continuació es mostra un exemple d'implementació a la classe `Symbol`, on s'il·lustren totes les modificacions d'aquesta millora. En color verd s'indica el codi afegit i en vermell l'esborrat.

```
1   #include <string>
2   #include <utility> // move semantics
3   #include <memory> // shared_ptr
4
5   namespace GOS {
6
7   class Scope;
8   typedef std::shared_ptr<Scope> ScopeRef;
9   class Type;
10  typedef std::shared_ptr<Type> TypeRef;
11
12  class Symbol;
13  typedef std::shared_ptr<Symbol> SymbolRef;
14  class Symbol {
15  public:
16      Symbol(std::string name) : name(std::move(name)) {}
17      Symbol(std::string name, TypeRef type) : name(std::move(name)), type(type
            ) {}
18      static SymbolRef Create(const std::string &name) {
19          return SymbolRef(new Symbol(name));
20      }
21      static SymbolRef Create(const std::string &name, TypeRef type) {
22          return SymbolRef(new Symbol(name, type));
23      }
24      virtual ~Symbol() {}
25      std::string getName() const { ... }
26      virtual Type* getType() { ... }
```

```
    virtual TypeRef getType() { ... }
    virtual bool isAssignable() { ... }
    virtual bool isScoped() { ... }

protected:
    Symbol(std::string name) : name(std::move(name)) {}
    Symbol(std::string name, TypeRef type) : name(std::move(name)), type(type) {}

    std::string name;
    Type* type = nullptr;
    TypeRef type = nullptr;
};

}
```

Alguns aspectes addicionals a ressaltar del codi són la inclusió del *header memory* per poder utilitzar la classe *shared_ptr*, la substitució dels punters per la seva versió *smart* i la necessitat de fer *forward declaration*[2] de la classe `Symbol` abans de declarar el tipus `SymbolRef`. Aquest darrer detall és perquè en el cos de la classe `Symbol` s'utilitza aquest tipus i ha d'haver estat definit abans.

Un cop implementades les millores de gestió de memòria vaig executar l'eina Valgrind (veure 7.2) per comprovar si havien disminuït els *memory leaks*. La comanda executada va ser la mateixa que a l'apartat 8.2.2. Els resultats obtinguts es mostren a continuació, comparats amb els valors d'abans de la millora:

```
HEAP SUMMARY:
    in use at exit: 1,030,934 bytes in 21,130 blocks
    in use at exit: 166,650 bytes in 2,343 blocks
  total heap usage: 482,490 allocs, 461,360 frees, 25,934,030 bytes allocated
  total heap usage: 495,572 allocs, 493,229 frees, 27,373,018 bytes allocated

LEAK SUMMARY:
   definitely lost: 368,392 bytes in 7,304 blocks
   definitely lost: 96 bytes in 1 blocks
   indirectly lost: 662,262 bytes in 13,821 blocks
   indirectly lost: 166,114 bytes in 2,332 blocks
     possibly lost: 0 bytes in 0 blocks</add>
   still reachable: 280 bytes in 5 blocks
   still reachable: 440 bytes in 10 blocks
        suppressed: 0 bytes in 0 blocks
```

En la sortida de Valgrind anterior es mostra en vermell els resultats obtinguts abans de la millora de gestió de memòria, en verd els de després i en negre els que no han variat. Com es pot veure, els *memory leaks* s'han reduït dràsticament. Per acabar-los d'eliminar hauria estat necessària una reescriptura més a fons del compilador, cosa que quedava fora dels objectius del projecte.

[2] Declaració d'un identificador abans de donar una definició. Si el llenguatge ho suporta, també es podrà utilitzar l'identificador abans de la seva definició.

9.2.2 Temps de càlcul

Per solucionar el problema del temps de càlcul vaig haver de resseguir l'execució d'instàncies problemàtiques de manera manual. Així vaig determinar que l'error estava en la càrrega de la fórmula compilada al *solver* mitjançant l'API. En concret, el tall de codi problemàtic era on s'afegien totes les clàusules d'una `SMTFormula` resultant del procés de compilació a un *solver* determinat. A continuació es mostra el codi en qüestió, ressaltant en vermell el codi esborrat i en verd l'afegit:

```
1  bool consistent = true;
2  for(int i = lastClause+1; i < workingFormula.f->getNClauses(); i++){
3      const clause & c = workingFormula.f->getClauses()[i];
4      vec<Lit> cv;
5      for(const literal & l : c.v)
6          cv.push(getLiteral(l,vars));
7      if(!s->addClause(cv)){
8          consistent = false;
9          break;
10     }
11     consistent = s->simplify();
12 }
13 consistent &= s->simplify();
```

En aquest codi es mostra com s'afegeixen al solver `s` les clàusules compilades contingudes a `workingFormula.f`. Abans de la modificació es realitzava una simplificació de la fórmula per cada clàusula afegida. El problema és que el procés de simplificació és molt costós i, en cas d'haver-hi moltes clàusules, el temps de càlcul es tornava impracticable. Això explicava perquè les instàncies que presentaven problemes segons els requeriments no poguessin solucionar-se en un temps raonable, i és perquè es traduïen en milions de clàusules. Per exemple, el problema de `nonogram.bup` amb una instància petita com `nono_duck.json` generava 3132 clàusules i trigava 351ms en solucionar-se però, en canvi, la instància `nono_gladiator.json` generava 1042422 clàusules i en una hora de càlcul no acabava l'execució. Després de la modificació el temps d'execució de `nono_duck.json` va passar a ser 266ms i el de `nono_gladiator.json` a 93.759s.

9.2.3 Etapes de compilació

El compilador de BUP inicial generava un arbre de *parsing* de tot el fitxer amb la codificació del model per cada bloc del programa. Llavors, es feia l'anàlisi semàntic i generació de codi individualment, mitjançant un *visitor* específic. Aquesta aproximació, tot i no ser incorrecta, sí era molt ineficient, perquè era innecessari haver de generar el mateix arbre de nou per cada bloc.

Per tant, vaig modificar la implementació perquè es generés un sol arbre de *parsing*, el qual s'exploraria múltiples vegades pels diferents *visitors*. A la figura 9.1 es mostra la modificació del flux del programa, on s'indica amb vermell les parts eliminades i amb verd les afegides.

9.3 Predicats

9.3.1 Definició

9.3.1.1 Gramàtica

Per tal que l'usuari de GOS pogués definir predicats personalitzats, vaig haver de modificar la gramàtica de BUP per incorporar les regles lèxiques i sintàctiques corresponents. Pel que fa als nous *tokens*, calia identificar la paraula clau d'inici de bloc de definició de predicats i la d'inclusió de nous fitxers. Les regles corresponents implementades a la gramàtica van ser:

```
1  TK_PREDICATES: 'predicates';
2  TK_INCLUDE: 'include';
```

L'estructura general d'un programa BUP passava a tenir un nou bloc opcional per la definició dels predicats. Per tant, vaig haver de modificar la regla inicial de la gramàtica de la següent manera:

```
1  csp2sat: entityDefinitionBlock? viewpointBlock predDefBlock?
       constraintDefinitionBlock outputBlock?;
```

El bloc de definició de predicats `predDefBlock` començaria amb la paraula clau `predicates` seguit de dos punts i d'un cos del bloc:

```
1  predDefBlock: TK_PREDICATES TK_COLON predDefBlockBody;
```

El cos del bloc `predDefBlockBody` estaria format per cap o més definicions de predicats o inclusions de fitxers, especificades en qualsevol ordre i, fins i tot, intercalades:

```
1  predDefBlockBody: (predDef | predInclude)*;
```

Pel que fa a les sentències per incloure altres fitxers `predInclude`, estarien iniciades per la paràula clau `include`, seguit d'un *string* (cadena de caràcters

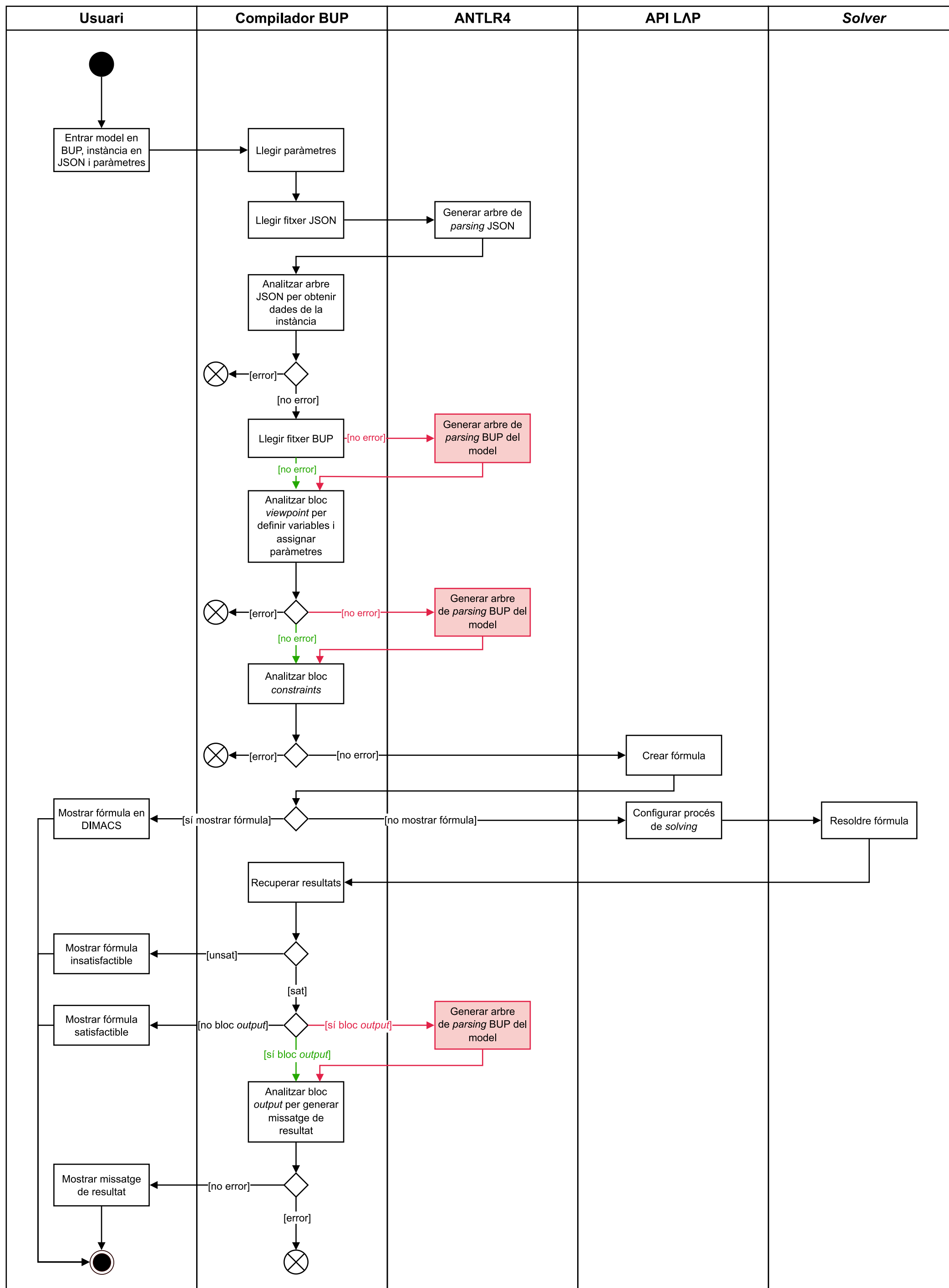

Figura 9.1: Diagrama de flux del funcionament del sistema GOS amb la modificació de la creació d'un sol arbre de *parsing*

entre cometes dobles) i acabades en punt i coma:

```
1 predInclude: TK_INCLUDE TK_STRING TK_SEMICOLON;
```

En quant a la definició d'un predicat `predDef` , la sintaxi seria molt semblant a la dels llenguatges de programació més populars (per exemple C++). És a dir, un predicat estaria format per un identificador, seguit dels paràmetres (opcionals) entre parèntesis i, a continuació, el cos del predicat entre claudàtors:

```
1 predDef: name=TK_IDENT TK_LPAREN predDefParams? TK_RPAREN TK_LBRACKET
      predDefBody TK_RBRACKET;
```

Els paràmetres de la signatura del predicat `predDefParams` tindrien la sintaxi pròpia de BUP, és a dir, es declararien de la mateixa manera que al bloc de *viewpoint*. Això també vol dir que es podrien utilitzar els mateixos tipus que els suportats pel llenguatge originalment (paràmetres enters i booleans, variables booleanes i vectors de qualsevol tipus). Només hi hauria una diferència en la sintaxi de la declaració dels vectors, i és que haurien de declarar-se les dimensions però sense especificar la mida de cadascuna. Això és perquè la mida dependria del vector que se li passés al predicat en el moment de la crida. Per exemple, una matriu de paràmetres enters es declararia com `param vec[][]` . La comprovació de quan s'havien d'indicar les mides es deixaria per l'anàlisi semàntic, on seria més senzill de comprovar. A continuació es mostra la sintaxi dels paràmetres, juntament amb les diferències en les regles corresponents a la definició de variables i predicats:

```
1  predDefParams: definition (TK_COMMA definition)*;
2
3  definition: varDefinition | paramDefinition;
4
5  varDefinition: TK_VAR type=TK_BASE_TYPE_BOOL? name=TK_IDENT arrayDefinition
       TK_SEMICOLON;
6
7  paramDefinition: (
8          TK_PARAM type=(TK_BASE_TYPE_BOOL | TK_BASE_TYPE_INT)
9          | type=TK_IDENT
10     ) name=TK_IDENT arrayDefinition TK_SEMICOLON;
11
12 arrayDefinition: (TK_LCLAUDATOR arraySize=expr? TK_RCLAUDATOR)*;
```

Pel que va al cos del predicat `predDefBody` , estaria format per un bloc de definició de variables locals i una o més definicions de *constraints*, on les variables haurien d'estar sempre abans del primer *constraint*:

```
1   predDefBody: predVarDefinitionBlock constraintDefinition+;
```

El bloc de variables locals `predVarDefinitionBlock` consistiria en una sèrie de definicions de variables booleanes (o cap):

```
1   predVarDefinitionBlock: (varDefinition TK_SEMICOLON)*;
```

Una definició de *constraint* `constraintDefinition` es faria de la mateixa manera que les restriccions expressades en el bloc `constraints` del model BUP.

9.3.1.2 *Visitor*

L'anàlisi semàntic i generació de codi el vaig implementar seguint l'esquema amb què va ser dissenyat inicialment el compilador BUP, és dir, vaig crear la classe `GOSPredVisitor` que s'encarregaria d'explorar el subarbre de *parsing* del bloc de definició de predicats. Per cada regla sintàctica descrita a l'apartat anterior (o node de l'arbre) implementaria un mètode en aquesta classe que sobreescrigués els mètodes virtuals de les classes pare. Aquestes classes eren, per ordre en la jerarquia, `GOSCustomBaseVisitor` i `BUPBaseVisitor`. La darrera proporcionava una implementació per defecte per totes les regles de la gramàtica BUP on simplement es feia un retorn agregat de tots els resultats de l'exploració dels nodes fills. En canvi, la classe `GOSCustomBaseVisitor`, implementava els mètodes d'exploració d'aquelles regles comunes entre tots els *visitors*, com l'avaluació d'expressions o la generació de llistes per comprensió. A la figura 9.2 es mostra el diagrama de classes de la jerarquia de *visitors*, mostrant-hi amb color verd el nou.

A continuació es descriu l'anàlisi semàntic i generació de codi que efectua cadascun dels mètodes de `GOSPredVisitor` per cada node de l'arbre de *parsing* o regla sintàctica:

- `predDefBlock` → És el node arrel del subarbre de *parsing* corresponent al bloc de definició de predicats. S'encarrega de visitar tots els nodes fills per declarar a la taula de símbols totes les definicions de predicats. Llavors visita de nou totes les definicions per comprovar la correctesa de les possibles crides a altres predicats amb el mètode auxiliar `checkPredCalls()`. Tot el codi de la funció està dins un bloc *try catch* per tal que si hi ha qualsevol error en el procés de visitar les definicions de predicats es mostri el missatge d'error corresponent però la compilació pugui seguir. Tot i que la compilació no acabarà de manera correcta perquè s'ha trobat un error,

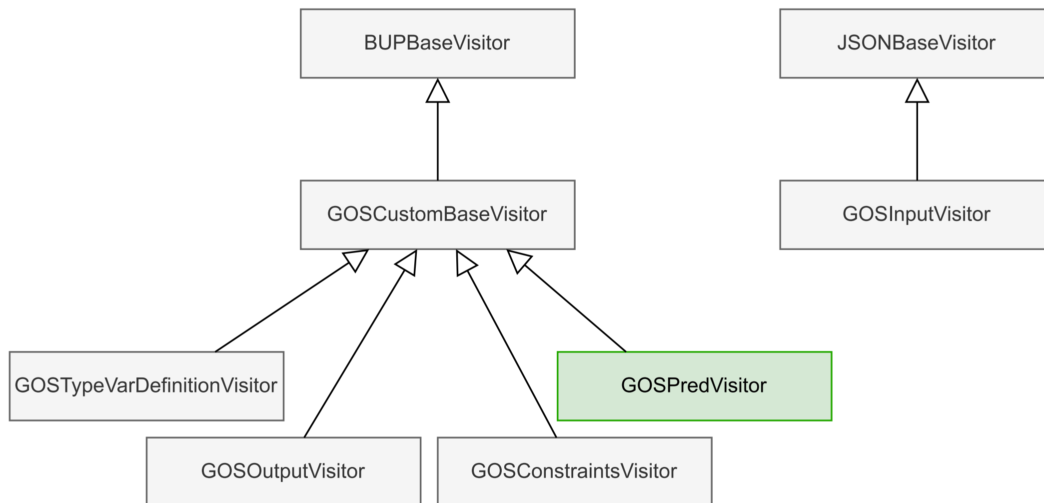

Figura 9.2: Diagrama de classes dels *Visitors* que exploren l'arbre de *parsing* compilat pel sistema GOS

això és desitjable perquè es pugui notificar a l'usuari de tots els errors en el fitxer BUP i no pas només d'un.

```
1  antlrcpp::Any
2  visitPredDefBlock(BUPParser::PredDefBlockContext *ctx) override
3  {
4      try {
5          BUPBaseVisitor::visitPredDefBlock(ctx);
6          checkPredCalls();
7      } catch (GOSException &e) {
8          std::cerr << e.getErrorMessage() << std::endl;
9      }
10     return nullptr;
11 }
```

La funció `checkPredCalls()` , s'encarrega d'iterar tots els símbols de tipus predicat de la taula de símbols per verificar la correctesa semàntica del cos dels predicats. Això ho fa obtenint el subarbre de *parsing* guardat a cada símbol de predicat i prepara un àmbit simulat abans d'explorar el cos del predicat. Aquest àmbit simulat consisteix en un nou nivell de la taula de símbols on s'hi declaren els paràmetres del predicat com a símbols falsos o sense valor real. Els símbols sense valor són suficients per fer el control d'errors en les crides a predicats, com la coherència de tipus o l'existència de predicats amb determinades signatures. Es segueix aquesta estratègia per poder reutilitzar al màxim el codi per visitar una crida a un predicat.

- `predDefBlockBody` → El cos del bloc de definicions de predicats simplement crida al mateix mètode de la classe pare, el qual visita per defecte

tots els nodes fills.

```
1  antlrcpp::Any
2  visitPredDefBlockBody(BUPParser::PredDefBlockBodyContext *ctx)
       override
3  {
4      return BUPBaseVisitor::visitPredDefBlockBody(ctx);
5  }
```

- `predInclude` → Per incloure un fitxer primer s'obté l'adreça del fitxer i, en cas de ser relativa, es calcula l'adreça canònica[3] respecte l'últim fitxer a la pila de fitxers inclosos `_includes`. Aquesta estructura de dades representa el fitxer actual en l'arbre d'*includes*, ja que BUP permet fer cadenes d'inclusions (incloure fitxers en fitxers inclosos). Així doncs, es permet incloure fitxers amb *path* absolut o relatiu al fitxer on s'escriu la sentència. Per exemple, si al fitxer */path/to/source/a.bup* hi ha la sentència `include "../../b.bup";`, l'adreça que s'interpretarà serà */path/b.bup*.

  A continuació es cerca al vector que guarda un registre de tots els fitxers inclosos (`st->parsedFiles`) si el fitxer que s'està avaluant ja ha estat inclòs abans. La condició per determinar-ho consisteix en comprovar si les adreces dels fitxers són equivalents. Això permet identificar fitxers iguals referenciats amb adreces diferents. Per exemple, es considerarient el mateix fitxer tant */path/a.bup* com */path/../path/a.bup*.En cas afirmatiu s'ignora i es mostra un missatge d'avís. Altrament es crea una instància de `BUPFile` (veure 9.3.5), s'afegeix al registre de fitxers inclosos i s'actualitza el nivell d'*include* empilant el fitxer a `_includes`. Finalment es visita l'arbre de *parsing* corresponent al fitxer inclòs. En cas d'error obrint el fitxer, es mostra un missatge a l'usuari.

```
1  antlrcpp::Any
2  visitPredInclude(BUPParser::PredIncludeContext *ctx) override
3  {
4      // Compute the path of the included file
5      const std::string name = ctx->TK_STRING()->getText();
6      std::filesystem::path filePath = name.substr(1, name.length()-2);
           // Trim double quotes
7      if (filePath.is_relative())
8          filePath = canonical(_includes.top().parent_path() / filePath);
9
10     // Discard those files already included
11     auto conditionFunc = [&] (const BUPFileRef& file) { return
           equivalent(file->getPath(),filePath); };
12     const bool isFileAlreadyIncluded = std::find_if(st->parsedFiles.
           begin(), st->parsedFiles.end(), conditionFunc) != st->
```

[3] Adreça o *path* absolut i únic. En sistemes operatius basats en UNIX, comença amb el directori arrel indicat amb una barra ('/'), seguida del camí en l'arbre de directoris, separant els salts entre nodes amb barres. Com és una adreça única, és comparable.

```
        parsedFiles.end();
13      if (!isFileAlreadyIncluded) {
14          try {
15              // Parse file and visit its subtree
16              BUPFileRef file = BUPFile::Create(filePath);
17              st->parsedFiles.emplace_back(file);
18              _includes.push(filePath);
19              visit(file->getParser()->predDefBlockBody());
20              _includes.pop();
21          } catch (std::ifstream::failure e) {
22              std::cerr << "Error reading file: " << filePath.filename()
                    << std::endl;
23          }
24      }
25      else {
26          std::cerr << "Warning: file " << filePath.filename() << "
                already included, parsing omitted" << std::endl;
27      }
28
29      return nullptr;
30  }
```

- `predDef` → El codi de la definició d'un predicat està envoltat per un bloc *try catch* per evitar que el compilador deixi sense avaluar els predicats definits a continuació però que es mostri a l'usuari el missatge d'error generat.

  En primer lloc es genera la signatura del predicat obtenint-ne el nom i els paràmetres. Aquesta signatura es representa mitjançant una sèrie de classes auxiliars que permeten descriure signatures de manera convenient (veure 9.3.4.1). En cas d'error, es mostra el missatge i es continua avaluant els paràmetres per tal de trobar tots els errors en la declaració de paràmetres.

  A continuació es verifica que no existeixi un altre predicat amb la mateixa signatura. En cas afirmatiu es mostra un missatge d'error. Altrament es defineix un nou `PredSymbol` a la taula de símbols, concretament al *scope* global, guardant-ne també la localització, és a dir, el fitxer i la posició on està definit el predicat. El fet de definir els predicats al *scope* global és perquè les definicions de predicats han de ser visibles des de totes les parts del codi i no es permeten declaracions niuades.

```
1   antlrcpp::Any
2   visitPredDef(BUPParser::PredDefContext *ctx) override
3   {
4       try {
5           std::string name = ctx->name->getText();
6           const size_t line = ctx->name->getLine();
7           const size_t col = ctx->name->getCharPositionInLine();
8
9           // Get whole predicate signature
10          PredSymbol::Signature signature;
11          signature.name = name;
```

```
12          if(ctx->predDefParams()) {
13              for (auto defCtx: ctx->predDefParams()->definition()) {
14                  try {
15                      PredSymbol::ParamRef param = visit(defCtx);
16                      signature.params.push_back(param);
17                  } catch (GOSException &e) {
18                      std::cerr << e.getErrorMessage() << std::endl;
19                  }
20              }
21          }
22
23          // Check if a predicate with same signature is already declared
24          if(this->currentScope->existsInScope(signature.toStringSymTable
                ())) {
25              ... // Throw CSP2SATALreadyExistsException
26          }
27
28          // Define predicate symbol as global
29          PredSymbol::Location loc = {_includes.top(), line, col};
30          PredSymbolRef pred = PredSymbol::Create(signature, loc, ctx);
31          st->gloabls->define(pred);
32
33      } catch (GOSException &e) {
34          std::cerr << e.getErrorMessage() << std::endl;
35      }
36
37      return nullptr;
38  }
```

- `predDefParams` → Per avaluar tots els paràmetres de la signatura del predicat es crida al mètode corresponent de la classe pare que, per defecte, visita tots els nodes fills.

```
1   antlrcpp::Any
2   visitPredDefParams(BUPParser::PredDefParamsContext *ctx) override
3   {
4       return BUPBaseVisitor::visitPredDefParams(ctx);
5   }
```

- `predDefBody` → La compilació del cos d'un predicat es porta a terme en el moment de la crida. Tot i això, en el bloc de definició de predicats és necessari que es visitin tots els *constraints* del cos d'un predicat per realitzar el control d'errors. Per aquesta raó en la regla semàntica corresponent a `predDefBody` només es crida al mètode de la classe pare, que s'encarrega de visitar tots els nodes fills.

```
1   antlrcpp::Any
2   visitPredDefBody(BUPParser::PredDefBodyContext *ctx) override
3   {
4       return BUPBaseVisitor::visitPredDefBody(ctx);
5   }
```

Llavors vaig haver d'implementar una regla semàntica per avaluar les cri-

des a predicats i fer-ne el control d'errors amb una passada extra de compilació. Aquesta s'anomena `visitPredCall` i té una implementació molt semblant a la feta en el *visitor* del bloc de *constraints*, tot i que només s'hi realitza el control d'errors. Això és gràcies a la construcció d'un àmbit d'execució simulat en la funció `checkPredCalls()`, comentada en la descripció del `predDefBlock`.

- `predVarDefinitionBlock` → El bloc de definició de variables locals d'un predicat no s'hauria d'avaluar en la definició de predicats, per aquesta raó hi ha un `assert` per detectar possibles errors de programació.

```
1  antlrcpp::Any visitPredVarDefinitionBlock(BUPParser::
       PredVarDefinitionBlockContext *ctx) override {
2      assert(false); // this block should not be evaluated in pred
           definition
3      return nullptr;
4  }
```

9.3.2 Crida

9.3.2.1 Gramàtica

Els predicats definits en BUP s'havien de poder cridar en el mateix lloc que es podia expressar un *constraint*. Això vol dir que es podien trobar crides tan el bloc de *constraints* i com en el cos d'altres predicats. Per tant, vaig modificar la regla gramatical que definia un *constraint* de la següent manera:

```
1  constraint:
2      constraint_expression
3      | predCall
4      | constraint_aggreggate_op;
```

Una crida `predCall` estaria formada per l'identificador del predicat seguit d'una llista opcional de paràmetres entre parèntesis:

```
1  predCall: name=TK_IDENT TK_LPAREN predCallParams? TK_RPAREN;
```

Els paràmetres de la crida `predCallParams` consistirien en un paràmetre seguit de cap o més paràmetres separats per comes:

```
1  predCallParams: predCallParam (TK_COMMA predCallParam)*;
```

Un paràmetre `predCallParam` podria ser un accés a una variable, una expressió o una llista:

```
1  predCallParam: varAccess | expr | list;
```

Cal destacar que en aquesta regla sintàctica hi ha una ambigüitat intencionada. Això és perquè una expressió (`expr`), per definició de la gramàtica, pot ser també un `varAccess` en el cas base de l'estratificació de les expressions. El problema és que una expressió sempre s'avalua en un valor i, en el cas del pas de paràmetres a predicats, pot interessar que això no passi en certs casos. Per solucionar-ho, vaig utilitzar una de les característiques d'ANTLR4, que és prioritzar les alternatives d'una regla segons l'ordre d'aparició. Per tant, primer consideraria un `varAccess` abans que `expr` . Es podria haver estratificat la regla per eliminar l'ambigüitat, però es va optar per l'alternativa descrita per raons de simplicitat.

Per exemple, suposem que existeix un vector definit al bloc de *viewpoint* com a `var myArray[n]` i un predicat amb signatura `pred(var[])` . Si la regla sintàctica fos `predCallParam: expr | list;` i es volgués passar al predicat tot el vector com `pred(myArray)` , el compilador detectaria `myArray` com a expressió i resultaria en un error peruquè l'expressió no es pot avaluar en un valor. En canvi, amb la regla ambigua, el compilador detectaria `myArray` com a `varAccess` i es podria passar tot el vector al predicat.

9.3.2.2 *Visitor*

Per realitzar l'anàlisi semàntic i generació de les clàusules de les crides a predicats vaig modificar el *visitor* de *constraints* perquè, tot i que les crides també es podrien trobar a les definicions dels predicats, només es visitaria el subarbre de *parsing* en el moment d'avaluar una crida en el bloc de *constraints*. En altres paraules, la compilació del cos dels predicats es faria sota demanda i només es compilarien aquells predicats que s'arribessin a utilitzar en algun lloc del bloc de *constraints*. A continuació s'explica el codi dels mètodes implementats segons les regles sintàctiques que corresponen:

- `constraint` → Com una crida de predicat podia estar en el mateix lloc que un constraint, vaig modificar la regla semàntica `visitConstraint` per afegir-hi aquest nou cas.

```
1  antlrcpp::Any
2  visitConstraint(BUPParser::ConstraintContext *ctx) override
3  {
```

```
4      if (ctx->constraint_expression()) {
5          formulaReturnRef result = visit(ctx->constraint_expression());
6          for (clause clause : result->clauses)
7              this->_f->addClause(clause);
8      } else if (ctx->predCall()) {
9          visit(ctx->predCall());
10     } else visit(ctx->constraint_aggreggate_op());
11     return nullptr;
12 }
```

- `predCall` → Per avaluar una crida a predicat en primer lloc s'obté la signatura que hauria de tenir el predicat segons l'identificador i els paràmetres que formen la crida. Per fer-ho, es visita cadascun dels nodes corresponents a un paràmetre de la crida per conèixer-ne el tipus i el valor. Aquesta visita, en cas de ser a una expressió, retorna el valor resultant d'avaluar-la i, en cas de ser un accés a variable, retorna el símbol corresponent. Un cop s'ha generat la signatura, en cas que no existeixi un predicat compatible a la taula de símbols, es construeix i es mostra un missatge d'error amb els possibles predicats candidats per ajudar a l'usuari a identificar l'error comès en el model BUP. Altrament s'obté el `PredSymbol` i es prepara un nou àmbit d'execució a partir de l'actual. En aquest s'hi defineixen els paràmetres passats al predicat en la crida. A continuació s'avalua el cos del predicat i, finalment s'esborra l'àmbit creat. Cal destacar que l'ús d'àmbits d'execució locals en forma de *stack* o pila permet avaluar correctament crides des del cos d'un predicat, suportant fins i tot la recursivitat.

```
1  antlrcpp::Any
2  visitPredCall(BUPParser::PredCallContext *ctx) override
3  {
4      // Get call parameters
5      std::vector<SymbolRef> paramsSymbols;
6      PredSymbol::Signature signature;
7      signature.name = ctx->name->getText();
8      if (ctx->predCallParams()) {
9          this->accessingNotLeafVariable = true;
10         // Evaluate all parameters from call
11         for (auto predCallParamCtx: ctx->predCallParams()->
               predCallParam()) {
12             SymbolRef sym;
13             antlrcpp::Any res = visit(predCallParamCtx);
14             if (res.is<ValueRef>()) { // Anonymous constant
15                 ValueRef val = res.as<ValueRef>();
16                 AssignableSymbolRef assignableSym;
17                 if(val->isBoolean())
18                     assignableSym = AssignableSymbol::Create("",
                           SymbolTable::_boolean);
19                 else assignableSym = AssignableSymbol::Create("",
                       SymbolTable::_integer);
20                 assignableSym->setValue(val);
21                 sym = assignableSym;
22             }
```

```
            else sym = res; // Defined symbol
            paramsSymbols.emplace_back(sym);

            // Construct signature to lookup the predicate in the symbol table
            const int type = sym->getType()->getTypeIndex();
            PredSymbol::ParamRef param;
            if (type == SymbolTable::tArray) {
                ArraySymbolRef arraySym = Utils::as<ArraySymbol>(sym);
                PredSymbol::ParamArrayRef paramArray(new PredSymbol::ParamArray);
                paramArray->elemType = arraySym->getElementsType()->getTypeIndex();
                paramArray->nDimensions = arraySym->getNDimensions();
                param = paramArray;
            }
            else param.reset(new PredSymbol::Param);
            param->name = sym->getName();
            param->type = type;
            signature.params.emplace_back(param);
        }
        this->accessingNotLeafVariable = false;
    }

    // Check if a predicate with same signature is defined
    SymbolRef predSym = this->currentScope->resolve(signature.toStringSymTable());
    if (predSym == nullptr) {
        ... // Throw CSP2SATPredNotExistsException with possible candidates
    }
    PredSymbolRef pred = Utils::as<PredSymbol>(predSym);

    // Setup scoped exec environment and compile predicate body
    this->currentScope = LocalScope::Create(this->currentScope);
    for (int i = 0; i < paramsSymbols.size(); i++) {
        SymbolRef sym = paramsSymbols[i];
        PredSymbol::ParamRef param = pred->getSignature().params[i];
        assert(param->type == sym->getType()->getTypeIndex());
        Utils::as<BaseScope>(this->currentScope)->define(param->name, sym);
    }
    visit(pred->getPredDefTree()->predDefBody());
    this->currentScope = this->currentScope->getEnclosingScope();

    return nullptr;
}
```

- `predCallParams` → El mètode corresponent als paràmetres d'una crida visita tots els nodes fills cridant el mètode de la classe pare.

```
antlrcpp::Any
visitPredCallParams(BUPParser::PredCallParamsContext *ctx) override
{
    return BUPBaseVisitor::visitPredCallParams(ctx);
}
```

- `predCallParam` → El mètode que correspon a aquesta regla sintàctica només invoca al mètode de la classe pare, el qual s'encarregarà de visitar

el node corresponent segons si el paràmetre s'obté d'un accés a variable, és el valor resultant d'avaluar una expressió o un vector.

```
1 antlrcpp::Any visitPredCallParam(BUPParser::PredCallParamContext *ctx)
       override
2 {
3     return BUPBaseVisitor::visitPredCallParam(ctx);
4 }
```

9.3.3 *Operador sizeof()*

9.3.3.1 Gramàtica

Un dels requeriments exigia que a un predicat se li pogués passar com a paràmetre qualsevol tipus de dades, incloent vectors de múltiples dimensions. El problema era que les mides de les dimensions dels vectors no es podien conèixer en el moment de compilar la definició del predicat, per aquesta raó es va permetre declarar-los com a paràmetre sense indicar-ne les mides, tal com s'explica a l'apartat 9.3.1.1. Per tal que l'usuari pogués seguir operant amb les mides dels vectors en, per exemple, el rang d'una sentència *forall*, vaig implementar l'operador `sizeof()`, el qual retornava la mida d'una de les dimensions d'un vector.

Per implementar-ho, els operadors sobre llistes van patir les següents modificacions en les regles lèxiques i sintàctiques:

```
1 TK_OP_AGG_SIZEOF: 'sizeof';
2 TK_OP_AGG_SUM: 'sum';
3 TK_OP_AGG_LENGTH: 'length';
4 TK_OP_AGG_MAX: 'max';
5 TK_OP_AGG_MIN: 'min';
6
7 opAggregateExpr: TK_OP_AGG_LENGTH | TK_OP_AGG_MAX | TK_OP_AGG_MIN |
      TK_OP_AGG_SUM | TK_OP_AGG_SIZEOF;
```

9.3.3.2 *Visitor*

Cal destacar que ja existia l'operador `length()` que retornava la mida d'un vector, però només en el cas que fos un vector d'enters. Per raons de retrocompatibilitat vaig decidir no modificar-lo i crear-ne un de nou, que funcionés per vectors de qualsevol tipus.

Per implementar l'anàlisi semàntic i generació de codi, vaig tractar-lo de manera molt similar als altres operadors, modificant la regla `visitExprListAgg` de la següent manera:

```
antlrcpp::Any
visitExprListAgg(BUPParser::ExprListAggContext *ctx) override
{
    ArraySymbolRef list = visit(ctx->list());
    ValueRef result = nullptr;

    if(ctx->opAggregateExpr()->getText() == "sizeof") {
        const int size = list->getSymbolVector().size();
        result = IntValue::Create(size);
    }
    else if(list->getElementsType()->getTypeIndex() == SymbolTable::tInt){
        std::vector<SymbolRef> elements = list->getSymbolVector();

        if(ctx->opAggregateExpr()->getText() == "sum"){
            ... // Retornar valor de la suma
        }
        else if(ctx->opAggregateExpr()->getText() == "max"){
            ... // Retornar el maxim de la llista
        }
        else if(ctx->opAggregateExpr()->getText() == "min"){
            ... // Retornar el minim de la llista
        }
        else { // Length
            result = IntValue::Create(elements.size());
        }
    }
    else{
        ... // Throw CSP2SATInvalidExpressionTypeException
    }

    return result;
}
```

9.3.4 Taula de símbols

9.3.4.1 PredSymbol

Per poder implementar la millora dels predicats, havia de guardar-me informació sobre les definicions per poder realitzar les comprovacions d'errors i la generació de codi. Per aquesta raó vaig implementar la classe `PredSymbol`, que seria un nou tipus de símbol, tal com mostra la jerarquia de classes de la figura 9.3. Aquesta nova classe guardaria la signatura del predicat, la localització d'on havia estat definit el predicat i el subarbre de *parsing* del cos del predicat per fer la compilació sota demanda:

```
class PredSymbol;
typedef std::shared_ptr<PredSymbol> PredSymbolRef;
class PredSymbol : public Symbol {
public:
    ... // Auxiliar Signature and Param classes
    static PredSymbolRef Create(Signature sig, Location loc, BUPParser::
         PredDefContext* predDefTree) {
        return PredSymbolRef(new PredSymbol(sig, loc, predDefTree));
    }
    ... // Getters and setters
```

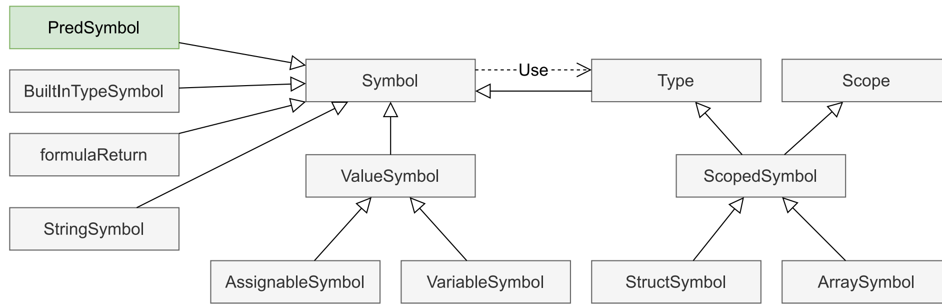

Figura 9.3: Diagrama de classes relacionades amb la classe *Symbol*

```
10
11 protected:
12     PredSymbol(Signature sig, Location loc, BUPParser::PredDefContext*
           predDefTree) :
13     Symbol(sig.toStringSymTable(), SymbolTable::_varbool),
14     _signature(sig), _predDefTree(predDefTree), _loc(loc)
15     { }
16
17 private:
18     Location _loc;
19     Signature _signature;
20     BUPParser::PredDefContext* _predDefTree;
21 }
```

Aquesta classe també contenia les definicions de les classes auxiliars per una gestió convenient de les signatures dels predicats. La classe principal era `PredSymbol::Signature` , la qual representava la signatura d'un predicant amb el nom i una llista de paràmetres:

```
1  struct Signature {
2      const std::string toStringSymTable() {
3          std::string res = name + "(";
4          for (ParamRef p : params) res += p->toStringSymTable();
5          res += ")";
6          return res;
7      }
8      const std::string toString() {
9          std::string res = name + "(";
10         if (params.size() > 0) {
11             for (int i = 0; i < params.size() - 1; i++)
12                 res += params[i]->toString() + ", ";
13             res += params[params.size() - 1]->toString();
14         }
15         res += ")";
16         return res;
17     }
18
19     std::string name;
20     std::vector<ParamRef> params;
21
22 };
23 typedef std::shared_ptr<Signature> SignatureRef;
```

Pel que fa als paràmetres de la signatura, n'hi havia de dos tipus. El primer, representat amb la classe `PredSymbol::Param`, representava un paràmetre qualsevol de la signatura amb el nom i el tipus:

```
1   struct Param {
2       Param() = default;
3       Param(const std::string& name, int type) : name(name), type(type) {}
4       virtual ~Param() {}
5       virtual std::string toStringSymTable() {
6           return std::to_string(type);
7       }
8       virtual std::string toString() {
9           return SymbolTable::typeToString(type);
10      }
11
12      std::string name;
13      int type;
14  };
15  typedef std::shared_ptr<Param> ParamRef;
```

El segon tipus de paràmetre `PredSymbol::ParamArray` consistia en una especialització de `PredSymbol::Param` per representar un paràmetre de tipus vector. Aquesta classe guardava, a més del nom i tipus del pare, el tipus dels elements i el nombre de dimensions.

```
1   struct ParamArray : public Param {
2       ParamArray() : Param() {}
3       ParamArray(const std::string& name, int type, int elemType, int
            nDimensions) :
4           Param(name,type), elemType(elemType), nDimensions(nDimensions) {}
5       std::string toStringSymTable() override {
6           std::string res = std::to_string(elemType);
7           for(int i = 0; i < nDimensions; i++) res += "[]";
8           return res;
9       }
10      std::string toString() override {
11          std::string res = SymbolTable::typeToString(elemType);
12          for(int i = 0; i < nDimensions; i++) res += "[]";
13          return res;
14      }
15
16      int elemType;
17      int nDimensions;
18  };
19  typedef std::shared_ptr<ParamArray> ParamArrayRef;
```

Aquests atributs addicionals eren necessaris per poder fer un control de tipus més acurat, permetent a l'usuari diferenciar predicats pel tipus dels vectors dels paràmetres o per les seves dimensions (per exemple, una llista d'una matriu).

També hi havia la classe auxiliar `PredSymbol::Location` que representava el lloc on s'havia definit un predicat amb el fitxer, línia i posició:

```
struct Location {
    std::filesystem::path file;
    size_t line;
    size_t col;
};
```

Aquesta informació era necessària per poder proporcionar missatges d'error convenients, tal com s'explica a `predErrors`.

Els predicats s'identificaven a la taula de símbols mitjançant la seva signatura codificada en un *string*. Per aquesta raó les classes signatura i els dos tipus de paràmetres tenien implementats els mètodes `toString()` i `toStringSymTable()`. El primer generava un *string* que representava la signatura de manera llegible per mostrar missatges d'error adequadament i el segon un *string* que consistia en la codificació de la signatura per identificar el predicat a la taula de símbols. Per exemple, la signatura del predicat `pred(var x[], param int y) { true; }` amb el primer mètode seria `pred(4[]2)` i, en canvi, amb el segon es generaria el *string* `pred(varBool[], int)`.

La codificació del *string* identificador de la signatura a la taula de símbols estava format pel nom del predicat, seguit pel codi enter del tipus dels paràmetres entre parèntesis. En cas que un paràmetre fos un vector, s'indicaria amb un parell de claudàtors de tipus `[]` per cada dimensió del vector, sempre a la dreta del tipus dels elements. Per exemple, el vector `var bool x[][]` s'identificaria amb `4[][]`.

9.3.5 BUPFile

La millora de permetre incloure fitxers externs amb definicions de predicats implicava haver de compilar fitxers addicionals. En el moment en què el compilador es trobava amb una sentència *include*, havia de passar a analitzar les definicions de predicats contingudes en el fitxer inclòs i després continuar amb la compilació del fitxer principal. Aquest procés es tradueix en haver de generar i explorar nous arbres sota demanda.

Conceptualment, un node de l'arbre de *parsing* corresponent a una sentència *include*, consisteix en un node terminal pel programa principal. Per incloure el nou fitxer, s'hauria de generar el nou arbre de *parsing* i afegir-lo a l'arbre principal, convertint el node *include* en l'arrel d'aquest nou arbre.

La implementació descrita no era possible perquè la interfície d'ANTLR4 en C++ per gestionar arbres de *parsing* no permetia afegir nous nodes a un arbre existent. Per tant, havia de guardarme individualment tots els arbres de cada fitxer inclòs. Per aquesta raó vaig crear la classe `BUPFile`:

```
1  class BUPFile;
2  typedef std::shared_ptr<BUPFile> BUPFileRef;
3  class BUPFile {
4  public:
5      static BUPFileRef Create(const std::filesystem::path& filePath) {
6          return BUPFileRef(new BUPFile(filePath));
7      }
8      BUPParser* getParser() { ... }
9      std::filesystem::path getPath() { ... }
10
11 private:
12     typedef std::shared_ptr<antlr4::CommonTokenStream> TokenStreamRef;
13     typedef std::shared_ptr<antlr4::ANTLRInputStream> InputStreamRef;
14     typedef std::shared_ptr<BUPLexer> BUPLexerRef;
15     typedef std::shared_ptr<BUPParser> BUPParserRef;
16
17     BUPFile(const std::filesystem::path& filePath) : _path(filePath) {
18         std::string fileContent;
19         try {
20             fileContent = Utils::readFile(absolute(filePath));
21         } catch(std::ifstream::failure e) {
22             std::cerr << "Error reading file: " << filePath.filename() << std
                   ::endl;
23             abort();
24         }
25         _inputStream = std::make_shared<antlr4::ANTLRInputStream>(fileContent)
               ;
26         _lexer = std::make_shared<BUPLexer>(_inputStream.get());
27         _tokenStream = std::make_shared<antlr4::CommonTokenStream>(_lexer.get
               ());
28         _parser = std::make_shared<BUPParser>(_tokenStream.get());
29     }
30
31     std::filesystem::path _path;
32     BUPParserRef _parser;
33     BUPLexerRef _lexer;
34     TokenStreamRef _tokenStream;
35     InputStreamRef _inputStream;
36 };
```

Aquesta classe s'encarrega de llegir un fitxer donat el seu *path* i generar-ne l'arbre de *parsing* corresponent. Un cop generat, guarda com atribut una referència a l'arbre de *parsing* (contingut a `_parser`). Els nodes d'aquest arbre generat carreguen la seva informació necessària sota demanda (per exemple, els *tokens* corresponents a la regla sintàctica). El problema és que, tot i fer servir dades d'estructures externes, el *parser* no es guarda cap referència a les estructures. Per tant, la responsabilitat de mantenir en memòria els objectes necessaris recau a l'usuari de la llibreria. Per aquesta raó `BUPFile` té els atributs `_lexer`, `_tokenStream` i `_inputStream`.

El cos dels predicats es compila sota demanda, és a dir, en el moment de fer la crida. Per poder fer-ho, el símbol es guarda el subarbre corresponent al cos del predicat. Perquè això sigui possible amb definicions fetes en fitxers externs, s'ha de mantenir en memòria els arbres de *parsing* dels fitxers inclosos. Així doncs, la taula de símbols guarda una llista de tots els `BUPFiles` creats. Aquesta es-

tructura serveix també per gestionar les cadenes d'inclusions i controlar errors, com compilar el mateix fitxer dues vegades.

```
std::vector<BUPFileRef> parsedFiles;
```

9.3.6 Errors

El sistema GOS era capaç de notificar a l'usuari amb missatges d'error convenients els possibles errors comesos en en el codi BUP. Amb la incorporació de la funcionalitat d'incloure definicions de predicats d'altres fitxers, aquests errors perdien el context i, per tant, la seva utilitat. Això és perquè, en cas de cometre un error de programació, no sabria a quin fitxer l'hauria comès, provocant que hagués de comprovar manualment tots els fitxers. Per aquesta raó vaig millorar els missatges d'error perquè informessin del fitxer corresponent. Per fer-ho, vaig modificar la classe `GOSException`:

```
class GOSException : public std::exception {
private:
    int line;
    int column;
    std::string message;
protected:
    std::optional<ExceptionLocation> _location;
    std::string _message;

public:
    GOSException(int line, int pos, const std::string &message) : line(line),
         column(pos), message(message) {
    GOSException(ExceptionLocation location, const std::string &message) :
        _location(location), _message(message) {
        SymbolTable::errors = true;
    }

    GOSException(const std::string &message) : _message(message) {
        SymbolTable::errors = true;
    }

    void setLocation(ExceptionLocation location) {
        _location = location;
    }

    std::string getErrorMessage(){
        std::string error = std::string("ERROR on line ") + std::to_string(
            line) + ":" + std::to_string(column) + "\n\t" + message;
        return error;
    }
    std::string getErrorMessage() {
        std::string error;
        if(_location.has_value())
            error += _location.value().toString() + " ";
        error += "ERROR: " + _message;
        return error;
    }
};
```

Aquesta classe era la base per tots els diferents tipus d'excepcions concretes. En el moment de crear-se una instància, es generava el missatge a mostrar a partir d'una localització i un missatge. Per qüestions de flexibilitat l'atribut `_location` era opcional i, per tant, es podrien crear errors sense localització. Per guardar aquesta informació vaig crear la següent classe auxiliar:

```
1  struct ExceptionLocation {
2      std::filesystem::path file;
3      size_t line;
4      size_t pos;
5
6      std::string toString() const {
7          return "In file \"" + file.string() + "\" (" + std::to_string(line) +
               ":" + std::to_string(pos) + "):";
8      }
9  };
```

Pel que fa a les excepcions concretes, vaig crear-ne una de nova per gestionar els errors relatius als predicats:

```
1  class CSP2SATPredNotExistsException : public GOSException {
2  public:
3      CSP2SATPredNotExistsException(ExceptionLocation location, const std::
           string& signature, const std::vector<std::pair<std::string,
           ExceptionLocation>>& candidates) :
4          GOSException(location, errorMessage(signature, candidates)) {}
5
6  private:
7      static std::string errorMessage(const std::string& signature, const std::
           vector<std::pair<std::string, ExceptionLocation>>& candidates)
8      {
9          std::string message = "Predicate with signature \"" + signature + "\"
               is undefined. ";
10         message += (candidates.size()>1 ? "Candidates are" : "Candidate is") ;
11         message += ":\n";
12         for (auto c : candidates) {
13             std::string modifiedLocStr = c.second.toString().erase(0,1);
14             modifiedLocStr.pop_back();
15             message += "\t" + c.first + "\t" + "Defined i" + modifiedLocStr +
                   "\n";
16         }
17         return message;
18     }
19 };
```

Aquesta classe generava un missatge d'error per informar a l'usuari que la crida a un predicat no corresponia a cap predicat definit. Per tal de facilitar la comprensió de l'error i ajudar a l'usuari a resoldre'l, amb el missatge d'error proporcionava una llista dels predicats candidats als que possiblement es referia en la crida errònia. En concret, es llistaven tots els predicats amb el mateix nom, però amb diferent signatura.

9.4 *Soft constraints*

9.4.1 Gramàtica

La millora dels *soft constraints* li aportaria al *Girona Optimization System* la capacitat d'optimitzar, és a dir, de trobar la millor solució a un determinat problema i instància. Per implementar la nova notació que permetria anotar una clàusula amb el pes corresponent vaig modificar la gramàtica del llenguatge BUP. En concret, vaig afegir el *token* `@` que indicaria l'assignació d'un pes a una clàusula:

```
1   TK_WEIGHT : '@';
```

També vaig modificar la regla sintàctica que definia els *constraints*, en concret les expressions, que passarien a tenir de manera opcional un pes a la part dreta, just abans del punt i coma. Una anotació de pes estaria formada per el *token* `@` seguida d'una expressió;

```
1   weight: TK_WEIGHT expr;
2   constraint:
3       constraint_expression weight?
4       | predCall
5       | constraint_aggregate_op;
```

Cal destacar que el valor del pes d'una clàusula era el resultat d'avaluar una expressió. Per definició a la gramàtica, una expressió podia avaluar-se en diferents tipus (enter i booleà), però un pes només podria consistir en un enter. Aquest control es realitzaria a l'anàlisi semàntic per no complicar la gramàtica.

9.4.2 *Visitor*

Pel que fa a l'anàlisi semàntic i generació de codi, hauria de modificar el *visitor* del bloc de *constraints* per gestionar el nou tipus de *constraint*. El mètode corresponent a la regla sintàctica `constraint`, en concret en el cas de tractar-se d'una expressió (`constraint_expression`) podria tenir opcionalment un pes anotat al costat. En cas de tenir un pes, es comprovaria que l'expressió no hagués generat més d'una clàusula, ja que la reificació no estava suportada. Llavors, en cas de tenir una sola clàusula, es visitaria el node corresponent a l'anotació de pes per obtenir-ne el valor corresponent i s'afegiria la clàusula a la fórmula com a *soft clause*. En cas de no tenir pes, es gestionaria de la mateixa manera com fins llavors, és a dir, afegint totes les clàusules a la fórmula com a *hard clause*.

```
antlrcpp::Any visitConstraint(BUPParser::ConstraintContext *ctx) override {
    if (ctx->constraint_expression()) {
        formulaReturnRef result = visit(ctx->constraint_expression());
        for (clause clause : result->clauses)
            this->_f->addClause(clause);
        if(ctx->weight()) {
            if(result->clauses.size() != 1) {
                ... // Throw CSP2SATInvalidFormulaException, soft constraints
                    can only have one clause since reification is not
                    supported
            }
            IntValueRef weight = visit(ctx->weight());
            this->_f->addSoftClause(result->clauses.front(), weight->
                getRealValue());
        }
        else {
            for (clause clause : result->clauses)
                this->_f->addClause(clause);
        }
    } else if (ctx->predCall()) {
        visit(ctx->predCall());
    } else visit(ctx->constraint_aggreggate_op());
    return nullptr;
}
```

L'obtenció del valor del pes es feia mitjançant l'exploració del node `weight`. Primer es visitava l'expressió per tal d'avaluar-la i obtenir-ne el resultat. Llavors es comprovava que aquest resultat fos un valor enter i major que zero. En cas afirmatiu es retornava un valor enter. Altrament es mostraven els missatges d'error corresponents.

```
antlrcpp::Any visitWeight(BUPParser::WeightContext *ctx) override {
    ValueRef weight = visit(ctx->expr());

    if(weight->isBoolean()) {
        ... // Throw CSP2SATInvalidExpressionTypeException, only integer
            weights allowed
    }
    else if(weight->getRealValue() < 1) {
        ... // Throw CSP2SATInvalidFormulaException, weights must be >= 1
    }

    return Utils::as<IntValue>(weight);
}
```

9.4.3 *Encoding*

Fins aleshores el sistema GOS no permetia optimitzar, només era capaç de trobar solucions satisfactibles. Això ho feia amb l'ajut de l'API del grup LIA, configurada per utilitzar l'optimitzador `SingleCheck`. Per poder optimitzar fórmules vaig haver de configurar-ne un que permetés optimització. L'únic que assolia aquest requeriment era el `NativeOptimizer`. Per tant, vaig configurar el compilador de BUP perquè un cop generada la fórmula corresponent al model, en cas de

contenir *soft constraints* configurés l'API per utilitzar el `NativeOptimizer` i, altrament, el `CheckOptimizer` :

```
if(_f->getNSoftClauses() > 1)
    sargs->setOption(OPTIMIZER, (std::string)"native");
else
    sargs->setOption(OPTIMIZER, (std::string)"check");
```

El `NativeOptimizer` , un cop finalitzat el procés de resolució i en cas que la fórmula resultés satisfactible, mostrava per pantalla la solució i, addicionalment, el cost de la mateixa. Per fer-ho, utilitzava el següent codi:

```
if(issat){
    int obj = e->getObjective();
    if(obj==INT_MIN){
        smtapierrors::fatalError(
            "The encoding must implement getObjective() to retrieve optimal
                solutions from native optimization"
            ,SOLVING_ERROR
        );
    }
    if(onNewBoundsProved) onNewBoundsProved(obj,obj);
    if(onSATSolutionFound) onSATSolutionFound(lb,ub,obj);
    if(onProvedOptimum) onProvedOptimum(obj);

    return obj;
}
```

L'*encoding* utilitzat havia d'implementar el mètode `getObjective()` per poder obtenir el cost de la solució obtinguda. Per aquesta raó vaig haver de modificar el `GOSEncoding` per tal que complís aquest requeriment de la següent manera:

```
class GOSEncoding : public Encoding {
private:
    SMTFormula *f;
    SymbolTable *st;
    bool sat = false;
    std::vector<bool> model;

    void fillModelValuesResult(ScopeRef currentScope, const EncodedFormula
        formula, const std::vector<bool> & bmodel) { ... }
    void printModelSolution(ScopeRef currentScope, std::ostream &os, std::
        string prefix = "") const { ... }

public:
    GOSEncoding(SMTFormula *formula, SymbolTable *st) { ... }
    SMTFormula *encode(int LB = 0, int UB = 0) override { ... }
    bool printModelSolution(std::ostream &os) const { ... }
    bool printSolution(std::ostream &os) const override { ... }

    void setModel(const EncodedFormula &ef, int lb, int ub, const std::vector
        <bool> &bmodel, const std::vector<int> &imodel) override {
```

```
18            sat = true;
19            model = bmodel;
20            fillModelValuesResult(this->st->gloabls, ef, bmodel);
21        }
22
23        int getObjective() const override {
24            const std::vector<int>& weights = f->getWeights();
25            const std::vector<clause>& softclauses = f->getSoftClauses();
26
27            int objective = 0;
28            for (int i = 0; i < softclauses.size(); i++) {
29                clause c = softclauses[i];
30                bool isSat = false;
31                for (auto lit : c.v) {
32                    bool val = lit.sign ? model[lit.v.id] : !model[lit.v.id];
33                    isSat = isSat || val;
34                }
35                if(!isSat)
36                    objective += weights[i];
37            }
38
39            return objective;
40        }
41
42        bool isSat(){ ... }
43    };
```

Vaig afegir l'atribut `model`, que es guardaria l'assignació de variables corresponent a la solució. Llavors, en el mètode `getObjective()`, recorreria totes les *soft clauses* de la fórmula per determinar si s'havien satisfet. Per comprovar-ho, aplicaria l'operador *or* lògic entre tots els literals de la clàusula, posant-li al literal el signe corresponent abans. En cas de no haver estat satisfeta, consistiria en una clàusula violada i caldria assumir el seu pes com a cost de la solució. Un cop sumats tots els costos, es retornaria el valor corresponent al cost de la solució òptima.

Un cop acabada aquesta ampliació del sistema GOS, l'usuari podia definir *soft constraints* en el seu model utilitzant el llenguatge BUP i aquest model es podria compilar en una fórmula amb pesos. També existia la possibilitat de mostrar la fórmula en pantalla en format DIMACS. Tot i això, no es podia resoldre dins el sistema GOS de la mateixa manera que sí es podia en el cas de problemes de satisfactibilitat. Per aquesta raó, en la implementació de la millora del desacoblament dels *solvers* (veure 9.5), s'afegiria el OpenWBO a la col·lecció de *solvers* oferits per defecte a GOS.

9.5 *Solvers*

9.5.1 Compilació

La millora de desacoblar el procés de *solving* de GOS tenia com a finalitat oferir més flexibilitat a l'usuari per configurar el sistema al seu gust, podent escollir el *solver* a utilitzar per resoldre la fórmula compilada o, fins i tot, indicar-ne un de personalitzat. Addicionalment, aquesta millora serviria per implementar la resolució de fórmules en el cas de problemes d'optimització.

L'objectiu era que GOS es comuniqués amb un *solver* que s'estigués executant en un procés extern en comptes de cridar l'API específica de cadascun. Per tant, el sistema hauria de disposar dels binaris dels *solvers* per poder-los executar en un procés independent.

Una opció hauria estat incloure els binaris amb el codi de GOS. Aquesta aproximació resultava difícil de mantenir, ja que, cada vegada que un dels *solvers*, s'actualitzés, s'hauria de compilar el codi actualitzat manualment i canviar-lo en el repositori de GOS.

Una altra hauria estat incloure una còpia del codi dels *solvers* amb el codi de GOS i compilar els executables juntament amb el sistema. Aquesta opció presentava els mateixos problemes que l'anterior.

Per tant, vaig optar per utilitzar submòduls de Git i per tenir els projectes dels *solvers* com a subprojectes dins GOS. un submòdul consisteix en una referència a un determinat *commit* o versió d'un codi desat en un repositori extern. Per tant, quan un *solver* s'actualitzés, els canvis tindrien efecte en GOS de manera implícita, només caldria actualitzar el submòdul a la versió desitjada.

Així doncs, vaig crear el fitxer *.gitmodules* i hi vaig incorporar com a submòduls els projectes dels *solvers* MiniSat, Glucose i OpenWBO:

```
1  [submodule "src/api/solvers/minisat"]
2     path = src/api/solvers/minisat-repo
3     url = https://github.com/niklasso/minisat
4  [submodule "src/api/solvers/glucose-repo"]
5     path = src/api/solvers/glucose-repo
6     url = https://github.com/mi-ki/glucose-syrup
7  [submodule "src/api/solvers/openwbo"]
8     path = src/api/solvers/openwbo-repo
9     url = https://github.com/sat-group/open-wbo
```

Llavors vaig editar el fitxer `CMakeLists.txt` per afegir-hi directives de compilació per cada submòdul:

```
1  # build solvers
2  set(SOLVERS_PARALLEL true)
3
4  # build openwbo
```

```
5   set(SOLVERS_SOURCES_DIR ${PROJECT_SOURCE_DIR}/src/api/solvers)
6   set(SOLVERS_BINARY_DIR ${CMAKE_BINARY_DIR}/solvers)
7   add_custom_target(openwbo ALL
8       WORKING_DIRECTORY ${SOLVERS_SOURCES_DIR}/openwbo-repo
9       COMMAND $(MAKE) rs && mkdir -p ${SOLVERS_BINARY_DIR} && mv -vn open-
            wbo_static ${SOLVERS_BINARY_DIR}/openwbo
10  )
11
12  # build glucose
13  if (${SOLVERS_PARALLEL})
14      add_custom_target(glucose ALL
15          WORKING_DIRECTORY ${SOLVERS_SOURCES_DIR}/glucose-repo/parallel
16          COMMAND $(MAKE) rs && mkdir -p ${SOLVERS_BINARY_DIR} && mv -vn glucose
                -syrup_static ${SOLVERS_BINARY_DIR}/glucose
17      )
18  else ()
19      add_custom_target(glucose ALL
20          WORKING_DIRECTORY ${SOLVERS_SOURCES_DIR}/glucose-repo/simp
21          COMMAND $(MAKE) rs && mkdir -p ${SOLVERS_BINARY_DIR} && mv -vn glucose
                -syrup_static ${SOLVERS_BINARY_DIR}/glucose
22      )
23  endif ()
24
25  # build minisat
26  set(MINISAT_REPO_DIR ${SOLVERS_SOURCES_DIR}/minisat-repo)
27  add_custom_target(minisat ALL
28      WORKING_DIRECTORY ${SOLVERS_SOURCES_DIR}/minisat-repo
29      COMMAND $(MAKE) CXXFLAGS="-fpermissive" && mkdir -p ${SOLVERS_BINARY_DIR}
            && mv -vn build/release/bin/minisat ${SOLVERS_BINARY_DIR}/minisat
30  )
```

Tal com mostra el codi, vaig afegir tres nous *targets* de compilació al fitxer. Això permetria a l'usuari compilar cada *solver* per separat especificant el nom del *target* en el moment de la compilació. En cas de no especificar-ne cap, es construiria, per defecte, el sistema GOS juntament amb tots els *solvers*. Cal destacar l'existència del *flag* `SOLVERS_PARALLEL`, el qual permetia a l'usuari compilar la versió concurrent d'aquells *solvers* que ho permetessin que, en aquest cas, només aplicava per Glucose.

La generació dels executables de cadascun dels *solvers* seguia un esquema molt semblant. En primer lloc es compilava l'executable seguint les indicacions de cada projecte i després es copiava al directori *./solvers/*, relatiu al directori de compilació de GOS.

9.5.2 Procés de *solving*

En l'etapa de *solving* del procés de resolució d'un model, el sistema GOS hauria de crear un procés extern amb l'executable del *solver* pertinent, comunicar-li la fórmula compilada i recuperar els resultats un cop finalitzada la tasca del *solver*.

Per fer-ho, vaig modificar l'API LIA perquè creés un nou procés de *shell*, hi executés una comanda i obtingués el resultat de l'execució:

```
1  std::shared_ptr<FILE> pipe(popen(getCall().c_str(),"r"), pclose);
2  if (!pipe) throw std::runtime_error("popen() failed!");
3  while (!feof(pipe.get())) {
4      if (fgets(buffer.data(), 512, pipe.get()) != nullptr)
5          result += buffer.data();
6  }
```

La comanda a executar en el nou procés es generava mitjançant la funció `getCall()`. Aquesta retornava la comanda necessària per invocar al *solver* amb el fitxer temporal DIMACS i recuperar aquelles línies rellevants del resultat:

```
1  std::string DimacsFileEncoder::getCall() const{
2      if(produceModels()){
3          if(solver == "openwbo")
4              return solverpath + " " + getTMPFileName() + " | grep -E '(^s )|(^
                   v )'";
5          else if(solver == "glucose")
6              return solverpath + " -model " + getTMPFileName() + " | grep -E
                   '(^s )|(^v )'";
7          else
8              return solverpath + " " + getTMPFileName() + " | grep -E '(^S )|(^
                   s )|(^v )'";
9      }
10     else {
11         if(solver == "glucose")
12             return solverpath + " " + getTMPFileName() + " | grep -E '(^s )|(^
                   c CPU time)' | cut -d ':' -f 2 | sed -e 's/s//g'";
13         else
14             return solverpath + " " + getTMPFileName() + " | grep -E '(^S )|(^
                   s )|(^v )'";
15     }
16 }
```

Finalment, amb el resultat obtingut del procés on s'havia executat el *solver*, es podria obtenir el model de la fórmula en cas d'existir solució.

9.5.3 Selecció de *solver*

Per tal que l'usuari pogués escollir el mètode de *solving* i poder complir amb la funcionalitat descrita a la figura 8.5, vaig afegir la possibilitat d'afegir arguments en el moment de la crida a l'executable de GOS.

Per fer-ho vaig modificar la classe `SolvingArguments` de L'API LIA, la qual s'encarregava de gestionar els arguments passats en la crida de l'aplicació per la configuració del procés de *solving*. En concret, vaig modificar l'argument per escollir el *solver* a utilitzar de la següent manera:

```
1  arguments::sop("s","solver",SOLVER,"glucose",
2      {"openwbo","glucose","minisat","custom"},
3      "Solver to use. API only available with minisat and glucose. Without API,
           only available glucose, openwbo and custom. For MaxSAT: openwbo. For
           SAT: glucose, minisat. Default: glucose."
```

```
4 )
```

Així l'usuari podria escollir el *solver* per MaxSAT implementat (OpenWBO) i també indicar-ne un de personalitzat amb l'opció `custom`. En aquest darrer cas, seria necessari indicar la comanda amb què s'hauria d'invocar el *solver* amb qüestió. Per això vaig afegir un nou argument:

```
1  arguments::sop("c","solver-command",SOLVERCOMMAND,"",
2     "Command to execute the custom solver. It must print the whole solver
          standard output (including model). Only when -s=custom."
3  )
```

Amb aquest argument l'usuari hauria d'indicar la comanda perquè el *solver* mostrés el resultat en format DIMACS (veure 7.9).

Per facilitar la detecció d'errors en la crida també vaig afegir uns missatges d'errors que indicarien arguments incompatibles:

```
1  if(sargs->getStringOption(SOLVER) == "custom" && sargs->getBoolOption(USE_API
       )) {
2     std::cerr << "Incompatible options found: Custom solver (-s=custom)
          cannot use api (-a=1) " << std::endl << std::endl;
3     std::cerr << "Run \"" << argv[0] << " -h\" for help" << std::endl;
4     exit(BADARGUMENTS_ERROR);
5  }
6
7  if(sargs->getStringOption(SOLVER) == "custom" && sargs->getStringOption(
       SOLVERCOMMAND) == "") {
8     std::cerr << "Custom solver set (-s=custom) but no custom command found
          (-c unset)" << std::endl << std::endl;
9     std::cerr << "Run \"" << argv[0] << " -h\" for help" << std::endl;
10    exit(BADARGUMENTS_ERROR);
11 }
12 else if(sargs->getStringOption(SOLVER) != "custom" && sargs->getStringOption(
       SOLVERCOMMAND) != "") {
13    std::cerr << "Warning: No custom solver (-s != custom) set but custom
          command found (-c set). Custom command will be ignored." << std::endl
           << std::endl;
14 }
```

9.6 *Debug*

Un dels requeriments era que GOS fos capaç de mostrar informació de *debug* per poder analitzar la creació de la fórmula i utilitzar el sistema en l'àmbit docent. Per aquesta raó vaig afegir la possibilitat de fer anotacions en forma de comentaris en el fitxer DIMACS generat i, també, que es mostressin el valor de totes les variables i paràmetres generats.

Per les anotacions, vaig definir un nou tipus de comentari en la gramàtica de BUP de la següent manera:

```
DIMACS_LINE_COMMENT : '//c ' ~[\r\n]*;
LINE_COMMENT : '//' ~[\r\n]* -> skip;
BLOCK_COMMENT : '/*' .*? '*/' -> skip;
```

El comentari `DIMACS_LINE_COMMENT` tenia una sintaxi molt semblant al comentari de línia ordinari, però en comptes d'iniciar-se amb `//`, el comentari per DIMACS començaria amb `//c`. Addicionalment, no tenia l'anotació `-> skip`, ja que es desitjava que el *token* no s'ignorés en l'anàlisi lèxic i es comuniqués al sintàctic. Per la gestió d'ambigüitats en les regles lèxiques que realitza ANTLR4 (veure 7.7), vaig haver de definir el `DIMACS_LINE_COMMENT` abans que el `LINE_COMMENT` perquè es detectés correctament.

Llavors vaig modificar la regla sintàctica corresponent a la definició d'un constraint perquè també pogués ser un comentari DIMACS:

```
constraintDefinition:
    ( forall | ifThenElse | constraint ) TK_SEMICOLON
    | DIMACS_LINE_COMMENT;
```

Aquesta decisió va ser per facilitar l'anàlisi semàntic i generació de codi. Això és perquè, en realitat, la generació d'un fitxer DIMACS a partir d'una fórmula consisteix en la codificació d'una clàusula per línia (veure 7.9). Com que els comentaris havien d'estar ubicats al mateix lloc on s'havien escrit en el model BUP, vaig optar per implementar clàusules de tipus "comentari" i tractarles de la mateixa manera que les clàusules ordinàries. Així doncs, s'inseririen en l'ordre correcte dins la llista de clàusules i també apareixerien en el lloc adient en el fitxer DIMACS. Les modificacions realitzades a la classe `clause` van ser:

```
struct clause {
    std::vector<literal> v;
    std::string comment;
    clause() {}
    clause(const std::string& c) : comment(c) {}
    clause(const clause & c) { ... }
    clause(const literal &lit) { ... }
    clause(const boolvar &b) { ... }
    clause(const std::vector<literal> & vec) { ... }
    clause& operator|=(const clause &c) { ... }
};
```

Llavors, en la generació del fitxer DIMACS, vaig haver de diferenciar els tipus de clàusules, tant per fitxers SAT com MaxSAT. A continuació es mostren les modificacions a la generació per MaxSAT:

```
int whard = f->getHardWeight();
os << "p wcnf " << f->getNBoolVars() << " " << f->getNClauses() + f->getNSoftClauses() << " " << whard << std::endl;
for(const clause & c : f->getClauses()){
    if (debug && c.comment != "") {
        add>os << "c " << c.comment << std::endl;</add>
    }
    else if (!c.v.empty()){
        os << whard << " ";
        for (const literal &l: c.v) {
            if (l.arith) {
                std::cerr << "Error: attempted to add arithmetic literal to SAT encoding" << std::endl;
                exit(BADCODIFICATION_ERROR);
            }
            if (l.v.id <= 0 || l.v.id > f->getNBoolVars()) {
                std::cerr << "Error: asserted undefined Boolean variable" << std::endl;
                exit(UNDEFINEDVARIABLE_ERROR);
            }
            os << (l.sign ? l.v.id : -l.v.id) << " ";
        }
        os << "0" << std::endl;
    }
}
```

Per mostrar el valor de les variables i paràmetres declarats al llarg del codi, vaig afegir una clàusula de tipus comentari després de cada declaració. A continuació es mostra, per ordre, la clàusula afegida a la fórmula `_f` en cas de ser una variable, un paràmetre o una variable local declarada dins un predicat:

```
_f->addClause(clause("var " + name + "-> " + newVar->toString()));
_f->addClause(clause("param " + name + "-> " + newConst->toString()));
_f->addClause(clause("local var " + name + "-> " + newVar->toString()));
```

Per obtenir un *string* que representés el contingut de les variables i paràmetres de manera convenient, vaig implementar a la classe `Symbol` el mètode `toString()` :

```
virtual std::string toString() const {
    return name;
}
```

Aquest mètode era virtual, però amb implementació per defecte. Aquesta implementació la vaig haver de sobreescriure per certs tipus de símbols per tal que el missatge generat fos fàcilment interpretable. La primera classe modificada va ser `AssignableSymbol` , on vaig haver de diferenciar el tipus de valor segons si era booleà o enter:

```
std::string toString() const override {
    if (val->isBoolean())
        return val->getRealValue() == 0 ? "false" : "true";
    else
        return std::to_string(val->getRealValue());
}
```

A la classe `VariableSymbol` vaig haver de convertir a *string* l'identificador de la variable:

```
std::string toString() const override {
    return std::to_string(var.v.id);
}
```

Finalment, a la classe `ArraySymbol`, vaig haver de mostrar els elements de totes les dimensions del vector:

```
std::string toString() const override {
    std::string res;
    res += "[";
    for (int i = 0; i < elements.size(); i++) {
        res += elements[i]->toString();
        if (i < elements.size()-1)
            res += ",";
    }
    res += "]";
    return res;
}
```

Aquesta informació de *debug* havia de poder-se activar i desactivar. Per tant, afegiria un nou argument a la classe `SolvingArgs` per poder configurar amb arguments en la crida al sistema GOS si mostrar o no aquesta informació. Per fer-ho vaig afegir el següent argument:

```
 arguments::bop("d","debug-dimacs",DEBUG_DIMACS,0,
    "Append debug information to DIMACS files (available info: variables id,
        parameters value, literal comments written after \"\\\\c \"). Default
        : 0."
)
```

9.7 Altres millores i correccions

Durant el desenvolupament del projecte van sorgir alguns problemes i es van detectar errors no coneguts abans de l'inici del projecte. Tot i no haver estat planificats ni contemplats a l'etapa de disseny, vaig decidir arreglar-los. Als següents apartats s'indiquen els més rellevants.

9.7.1 Warnings

Durant la compilació d'un model escrit en BUP, és possible que hi hagi situacions on es detecten errors en l'ús del llenguatge dels quals el compilador pugui recuperar-se. Per gestionar aquests casos de manera més convenient vaig crear la classe `GOSWarning`. Per implementar-la, vaig modificar la jerarquia de classes de `GOSException` per generalitzar el comportament comú d'ambdues classes en una comuna anomenada `GOSMessage`. En la figura 9.4 es mostra la jerarquia de classes resultant.

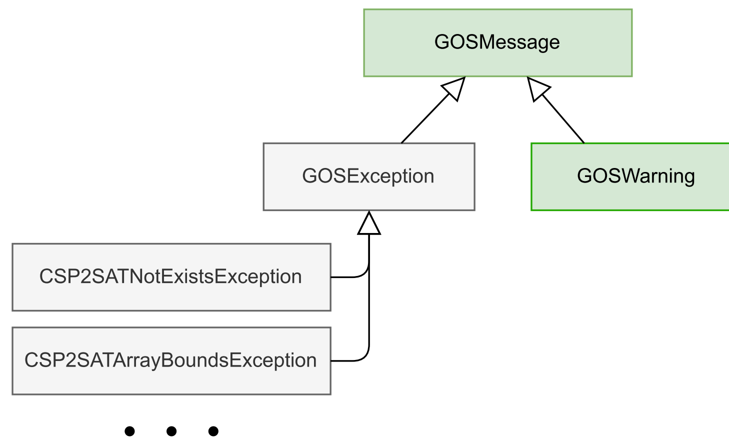

Figura 9.4: Jerarquia de classes de *GOSMessage*

Un dels casos on vaig utilitzar aquesta classe va ser en el cas de definir rangs descendents de l'estil `0..-1`. Fins aleshores, el sistema GOS mostrava un error i s'acabava l'execució quan, en realitat, en cas d'haver un rang descendent es pot tractar com un rang buit i simplement no realitzar el bucle. A continuació es mostra el codi modificat

```
1       if (minValue <= maxValue) {
2               ... // Fer bucle
3        } else {
4           throw GOSException(
5               {
6                   st->parsedFiles.front()->getPath(),
7                   ctx->start->getLine(),
8                   ctx->start->getCharPositionInLine()
9               },
10              "Range must be ascendant"
11          )
12          GOSWarning warning = GOSWarning(
13              {
14                  st->parsedFiles.front()->getPath(),
15                  ctx->min->start->getLine(),
16                  ctx->min->start->getCharPositionInLine(),
17              },
```

```
18            "Descendant range \"" + ctx->getText() + "\" detected (" + std::
                  to_string(minValue) + ".." + std::to_string(maxValue) + "),
                  loop omitted"
19          );
20          std::cerr << warning.getErrorMessage() << std::endl;
21      }
```

9.7.2 Prioritat dels operands

L'estratificació de prioritats per la correcta avaluació de les expressions tenia un error. El problema era que la prioritat dels operadors lògics *and* i *or* estaven invertides. Per aquesta raó vaig modificar les regles sintàctiques següents:

```
1 exprAnd: exprOr (TK_OP_LOGIC_AND exprOr)*;
2 exprOr: exprEq (TK_OP_LOGIC_OR exprEq)*;
3 exprOr: exprAnd (TK_OP_LOGIC_OR exprAnd)*;
4 exprAnd: exprEq (TK_OP_LOGIC_AND exprEq)*;
```

9.7.3 Operador *and* i *or* sobre llistes

Al llenguatge BUP, abans de l'inici d'aquest projecte, existien els operadors `||` i `&&` que aplicaven, respectivament, l'operació *or* i *and* a tots els elements d'una llista. Aquesta expressió s'avaluava en clàusules booleanes i es resolia el seu valor en temps d'execució en l'etapa de *solving*.

El problema és que no existien aquestes operacions pels casos en què es treballés amb paràmetres booleans. És a dir, no existia l'equivalent resoluble en temps de compilació. Per aquesta raó vaig implementar els operadors `lor` i `land`. Les modificacions fetes a la gramàtica van ser:

```
1 TK_OP_AGG_LENGTH: 'length';
2 TK_OP_AGG_MAX: 'max';
3 TK_OP_AGG_MIN: 'min';
4 TK_OP_AGG_OR: 'lor';
5 TK_OP_AGG_AND: 'land';
6
7 opAggregateExpr:
8     TK_OP_AGG_LENGTH | TK_OP_AGG_MAX | TK_OP_AGG_MIN | TK_OP_AGG_SUM |
          TK_OP_AGG_SIZEOF | TK_OP_AGG_OR | TK_OP_AGG_AND;
```

Pel que fa a la regla semàntica, vaig modificar el mètode `visitExprListAgg` perquè aquests operadors retornessin el resultat d'aplicar la operació *or* o *and* a tots els elements d'una llista amb elements booleans:

```
1 antlrcpp::Any
2 visitExprListAgg(BUPParser::ExprListAggContext *ctx) override
3 {
```

```
4         ArraySymbolRef list = visit(ctx->list());
5         ValueRef result = nullptr;
6 
7         if(ctx->opAggregateExpr()->getText() == "sizeof") {
8             ... // Sizeof eval
9         }
10        else if (list->getElementsType()->getTypeIndex() == SymbolTable::tBool
              ) {
11            std::vector<SymbolRef> elements = list->getSymbolVector();
12            if(ctx->opAggregateExpr()->getText() == "land") {
13                bool res = true;
14                for(auto & element : elements) {
15                    const int val = Utils::as<AssignableSymbol>(element)->
                          getValue()->getRealValue();
16                    res &= (val == 0 ? false : true);
17                }
18                result = BoolValue::Create(res);
19            }
20            else if (ctx->opAggregateExpr()->getText() == "lor") {
21                bool res = false;
22                for(auto & element : elements) {
23                    const int val = Utils::as<AssignableSymbol>(element)->
                          getValue()->getRealValue();
24                    res |= (val == 0 ? false : true);
25                }
26                result = BoolValue::Create(res);
27            }
28            else throw std::invalid_argument("Aggregate operator not supported
                  ");
29        }
30        else if(list->getElementsType()->getTypeIndex() == SymbolTable::tInt){
31            ... // Other list aggregate operators eval
32        }
33        else {
34            ... // Throw CSP2SATInvalidExpressionTypeException
35        }
36 
37        return result;
38    }
```

9.8 Proves

9.8.1 Predicats

9.8.1.1 Pas de paràmetres

En aquesta prova es demostra que es detecten correctament tots els tipus de paràmetres.

```
1 viewpoint:
2     var v;
3     var vArray[3];
4     var vMatrix[3][3];
5     param int i;
6     param int iArray[3];
7     param int iMatrix[3][3];
8     param bool b;
```

```
 9      param bool bArray[3];
10      param bool bMatrix[3][3];
11
12  predicates:
13      predVar(var vp, var vArrayp[], var vMatrixp[][]) {
14          true;
15      }
16
17      predInt(param int ip, param int iArrayp[], param int iMatrixp[][]) {
18          true;
19      }
20
21      predParamBool(param bool bp, param bool bArrayp[], param bool bMatrixp
            [][]) {
22          true;
23      }
24
25  constraints:
26      predVar(v, vArray, vMatrix);
27      predInt(i, iArray, iMatrix);
28      predParamBool(b, bArray, bMatrix);
```

```
s SATISFIABLE
```

9.8.1.2 Recursivitat

El següent codi utilitza com a restricció la crida a un predicat que conté una crida recursiva. El predicat crea les restriccions necessàries per posar a *true* les 3 primeres posicions del vector `vArray`

```
 1  viewpoint:
 2      var vArray[10];
 3
 4  predicates:
 5      predRecursive(var vArrayp[], param int j) {
 6          vArrayp[j-1];
 7          if (j-1 > 0) {
 8              predRecursive(vArrayp, j-1);
 9          };
10      }
11
12  constraints:
13      predRecursive(vArray, 3);
14
15  output:
16      [ vArray[j] ++ j < 9 ? ", " : "" | j in 0..9];
```

```
s SATISFIABLE
true, true, true, false, false, false, false, false, false, false
```

9.8.1.3 Referència creuada

En aquesta prova es té com a *constraint* la crida al predicat `crossRefA`. El predicat conté una crida a un altre predicat `crossRefB`. A la vegada, aquest darrer conté una crida al predicat `crossRefA`. Així doncs, consisteix en un cas de referència creuada. El propòsit del programa és el mateix que a l'apartat anterior, és a dir, posar les 3 primeres posicions del vector de variables `vArray` amb valor *true*;

```
1  viewpoint:
2      var vArray[10];
3
4  predicates:
5      crossRefA(var vArrayp[], param int j) {
6          vArrayp[j-1];
7          if (j-1 > 0) {
8              crossRefB(vArrayp, j-1);
9          };
10     }
11
12     crossRefB(var vArrayp[], param int j) {
13         vArrayp[j-1];
14         if (j-1 > 0) {
15             crossRefA(vArrayp, j-1);
16         };
17     }
18
19 constraints:
20     crossRefA(vArray, 6);
21
22 output:
23     [ vArray[j] ++ j < 9 ? ", " : "" | j in 0..9];
```

```
s SATISFIABLE
true, true, true, true, true, true, false, false, false, false
```

9.8.1.4 Sizeof

L'operador `sizeof` servia per obtenir les mides d'una de les dimensions d'un vector. En aquesta prova es verifica que funciona per vectors d'una i més dimensions. A més, es comprova que funciona en el cas d'ús principal de l'operador, és a dir, dins els predicats. El predicat `predSizeof` genera els *constraints* necessaris perquè les `j` posicions del vector de variables `vArrayp` tinguin el valor *true*. Per tant, com es crida amb els paràmetres `predSizeof(vArray, 4)`, les darreres 4 posicions seran *true*.

```
1  viewpoint:
2      var vArray[10];
3      param int iArray[4];
```

```
4       param int iMatrix[3][4];
5       param bool bArray[3];
6
7   predicates:
8       predSizeof(var vArrayp[], param int j) {
9           forall(k in sizeof(vArrayp)-j..sizeof(vArrayp)-1) {
10              vArrayp[k];
11          };
12      }
13
14  constraints:
15      predSizeof(vArray, 4);
16
17  output:
18      [ vArray[j] ++ j < 9 ? ", " : "" | j in 0..9];
19      "sizeof(vArray) = " ++ sizeof(vArray);
20      "sizeof(iArray) = " ++ sizeof(iArray);
21      "sizeof(bArray) = " ++ sizeof(bArray);
22      "sizeof(iMatrix[0]) = " ++ sizeof(iMatrix[0]);
```

```
s SATISFIABLE
false, false, false, false, false, false, true, true, true, true
sizeof(vArray) = 10
sizeof(iArray) = 4
sizeof(bArray) = 3
sizeof(iMatrix[0]) = 4
```

9.8.1.5 Sobrecàrrega

Per provar que la sobrecàrrega de predicats funciona, hi ha definits tres predicats amb identificadors `predOverload`, però amb signatures diferenciades en el tipus dels operands. En cas de cridar al predicat amb tres paràmetres, el model resulta insatisfactible.

```
1   viewpoint:
2       var v;
3       var vArray[10];
4       param int i;
5
6   predicates:
7       predOverload(var x, param int y) {
8           true;
9       }
10
11      predOverload(var x, param int y, var bool z[]) {
12          false;
13      }
14
15  constraints:
16      predOverload(v, i, vArray);
```

```
s UNSATISFIABLE
```

En cas de cridar al segon, el model resulta satisfactible.

```
1  ...
2  constraints:
3     predOverload(v, i);
```

```
s SATISFIABLE
```

En cas de cridar de manera errònia al predicat, es mostren tots els possibles predicats sobrecarregats.

```
1  ...
2  constraints:
3     predOverload(true);
```

```
In file "../input/test_predicates.bup" (69:4): ERROR: Predicate with signature "
    predOverload(bool)" is undefined. Candidates are:
       predOverload(varBool, int)   Defined in file "/home/david/tfg/GOS/input/
           test_predicates.bup" (52:4)
       predOverload(varBool, int, varBool[]) Defined in file "/home/david/tfg/GOS/
           input/test_predicates.bup" (56:4)

Execution stopped due to errors in constraint definition
```

9.8.1.6 *Includes*

Per verificar la inclusió de fitxers, les cadenes d'inclusions i veure que no es compilen fitxers inclosos més d'una vegada, aquesta prova replica l'escenari descrit a la figura 8.4. Seguint aquest esquema, suposem els fitxers *include.bup*, *include2.bup* i *globals.bup*, mostrats per ordre a continuació:

```
1  include "./globals.bup";
2
3  predInclude(var bool x) {
4     predGlobal(x);
5  }
```

```
1  include "./globals.bup";
2
3  predInclude2(var bool x) {
4     predGlobal(x);
5  }
```

```
1  predGlobal(var bool x) {
2     true;
3  }
```

Llavors, si executem el següent model ens dóna que és satisfactible i es pot

comprovar que les inclusions funcionen correctament. També es mostra el corresponent missatge d'avís de inclusió del mateix fitxer més d'una vegada.

```
1   viewpoint:
2       var v;
3
4   predicates:
5       include "./include.bup";
6       include "./include2.bup";
7
8   constraints:
9       predInclude(v);
10      predInclude2(v);
```

```
In file "../input/test_predicates.bup" (1:0): WARNING: file "globals.bup" already
    included, parsing omitted
s SATISFIABLE
```

9.8.2 *Soft constraints*

En la següent prova es verifica que els pesos assignats a clàusules s'incorporen correctament a la fórmula i que efectivament els pesos poden ser el resultat de qualsevol expressió, fins i tot l'accés a un vector d'enters. A continuació es mostra el model utilitzat:

```
1   viewpoint:
2       var vArray[10];
3       param int iArray[4]; // [1,2,3,4]
4
5   constraints:
6       forall(j in 0..sizeof(vArray)-1) {
7           vArray[j] @iArray[j % 4];
8       };
```

Per executar aquesta prova, es va optar per mostrar la fórmula en format DIMACS per corroborar que efectivament s'anotaven bé els pesos:

```
p wcnf 10 12 25
1 1 0
2 2 0
3 3 0
4 4 0
1 5 0
2 6 0
3 7 0
4 8 0
1 9 0
2 10 0
```

9.8.3 *Solvers*

Donat que a la resta de proves s'utilitzen tot els tipus de *solvers*, en aquesta es comprovarà que l'aplicació s'executa correctament amb un *solver* extern, és a dir, indicant una comanda personalitzada.

El model utilitzat és el mateix que a l'apartat 10.1.3.1, per demostrar que funciona amb un exemple que conté totes les noves funcionalitats implementades. El *solver* que es fa servir és un executable de Pacose compilat manualment, independentment de GOS. En concret, la versió del *solver* és la del 2019, obtinguda de la MaxSAT Evaluation [Bacchus et al., nd]. El resultat de l'execució és el següent:

```
$ ./CSP2SAT ../input/amk3.bup ../input/amk3.json -a=0 -s=custom -c="./pacose"
   s OPTIMUM FOUND
   o 6
   PB formula = 1*1 + 2*1 + 3*1 + 4*1 + 5*0 + 6*0 + 7*0 + 8*0 + 9*0 + 10*0
   Result = 10
```

9.8.4 *Debug*

Per provar que la informació de debug es mostrava correctament, es va implementar un model de prova on es declaressin tot tipus de variables i paràmetres i s'utilitzessin *constraints*. A més, es provaria que la funcionalitat serveix per predicats també. A continuació es mostra el model:

```
1  viewpoint:
2      var v;
3      var vArray[10];
4      var vMatrix[10][10];
5      param int i;
6      param int iArray[4];
7      param int iMatrix[3][4];
8      param bool b;
9      param bool bArray[3];
10     param bool bMatrix[3][3];
11
12 predicates:
13     pred() {
14         var bool predArray[10];
15         //c Pred comment
16         forall(i in 0..sizeof(predArray)-1) {
17             predArray[i];
18         };
19     }
20
21 constraints:
22     //c Pred call
23     pred();
24
25     //c Constraint block clause
26     !v;
```

Per tal que es mostrés la informació de *debug*, es va executar el sistema GOS perquè mostrés per pantalla la codificació DIMACS amb les annotacions:

```
$ ./CSP2SAT ../input/test_debug.bup ../input/test_debug.json -pf=1 -d=1
    p cnf 121 24
    c var v-> 1
    c var vArray-> [2,3,4,5,6,7,8,9,10,11]
    c var vMatrix-> [[12,13,14,15,16,17,18,19,20,21],
         [22,23,24,25,26,27,28,29,30,31], [32,33,34,35,36,37,38,39,40,41],
         [42,43,44,45,46,47,48,49,50,51], [52,53,54,55,56,57,58,59,60,61],
         [62,63,64,65,66,67,68,69,70,71], [72,73,74,75,76,77,78,79,80,81],
         [82,83,84,85,86,87,88,89,90,91], [92,93,94,95,96,97,98,99,100,101],
         [102,103,104,105,106,107,108,109,110,111]]
    c param i-> 3
    c param iArray-> [1,2,3,4]
    c param iMatrix-> [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
    c param b-> true
    c param bArray-> [true,false,true]
    c param bMatrix-> [[true,false,false],[false,true,false],[false,false,true]]
    c Pred call
    c local var predArray-> [112,113,114,115,116,117,118,119,120,121]
    c Pred comment
    112 0
    113 0
    114 0
    115 0
    116 0
    117 0
    118 0
    119 0
    120 0
    121 0
    c Constraint block clause
    -1 0
```

La sortida anterior mostra el valor de tots els paràmetres i variables del bloc de *viewpoint* i també del bloc de declaració de variables locals del predicat. Addicionalment, mostra els comentaris amb línies iniciades amb `c`.

CAPÍTOL 10

Implantació i resultats

10.1 *Sequential Weight Counter*

En aquest apartat es codifica amb BUP els *constraints* pseudo-booleans utilitzant la codificació*Sequential Weight Counter* (SWC), descrita a [Bofill et al., 2020].

Un *constraint* pseudo-booleà consisteix en una funció booleana amb forma de $\sum_{i=1}^{n} q_i x_i \# K$, on K i q_i són constants enteres. Les constants q_i són els coeficients o pesos de les variables booleanes x_i i # és un operador relacional. En altres paraules, que el sumatori de les variables x_i que siguin certes multiplicades pel seu coeficient q_i sigui # que K.

Un SWC és una codificació per *constraints* pseudo-booleans que suma seqüencialment, d'esquerra a dreta, els coeficients o pesos d'aquelles variables que siguin certes.

Un *constraint At-Most-K* (AMK) és una funció booleana de la forma $\sum_{i=1}^{n} x_i \leq K$, on x_i són variables booleanes i K és una constant entera. Aquest representa la restricció que com a molt K variables del conjunt $\{x_1, \ldots, x_n\}$ siguin certes. Per tant, un AMK no és més que un pseudo-booleà on tots els coeficients q_i són igual a 1.

La codificació en llenguatge BUP expressa les següents restriccions, basades en la codificació de circuit d'un SWC, com el que es mostra en la figura 10.1:

$$\overline{s_{i-1,j}} \vee s_{i,j} \qquad 2 \leq i < n, i \leq j \leq K \qquad (10.1)$$

$$\overline{x_i} \vee s_{i,j} \qquad 1 \leq i < n, i \leq j \leq q_i \qquad (10.2)$$

$$\overline{s_{i-1,j}} \vee \overline{x_i} \vee s_{i,j+q_i} \qquad 2 \leq i < n, i \leq j \leq K - q_i \qquad (10.3)$$

$$\overline{s_{i-1,K+1-q_i}} \vee \overline{x_i} \qquad 2 \leq i < n \qquad (10.4)$$

10.1.1 SAT

En primer lloc, es codifica el SWC com a problema de tipus SAT. Com a *viewpoint* hi ha el conjunt de variables $\{x_1, \ldots, x_n\}$ representat amb el vector de variables `x`. Els paràmetres declarat son el nombre de variables i coeficients (`nVars`), els coeficients de cada variable (vector `q`) i la constant entera K (`k`). Els

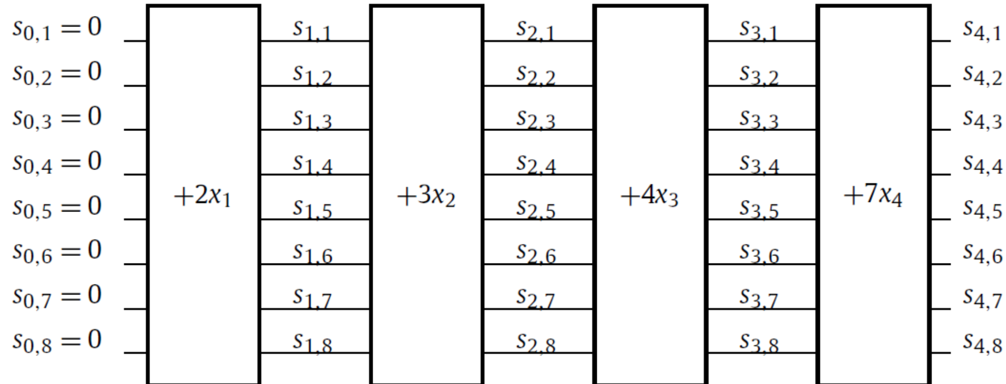

Figura 10.1: Representació en circuit d'un SWC [Bofill et al., 2020]

valors que adopten aquests paràmetres estan determinats per les dades del fitxer `.json`. El seu contingut és el següent:

```
{
  "k" : 10,
  "q" : [1,2,3,4,5,6,7,8,9,10]
}
```

Els *constraints* pseudo-booleans s'han codificat dins el predicat `swc_pb`, que rep un vector de variables, un vector amb els coeficients i la constant K. En seu cos hi ha codificat el circuit de la figura 10.1 amb l'ajut de la matriu de variables auxiliars `s`, que codifica les entrades i sortides de cada bloc del circuit. Els *constraints* del cos del predicat expressen les restriccions expressades al principi d'aquest apartat.

Per aquest problema existeixen nombroses solucions satisfactibles. Per evitar que el resultat sigui 0 (la primera assignació satisfactible és totes les variables amb valor *false*) s'ha afegit la restricció `ALK(x,2)` per obligar que com a mínim dues variables siguin certes. A continuació es mostra el fitxer del model:

```
viewpoint:
    param int nVars; // 10
    var bool x[nVars];
    param int q[nVars]; // = [1,2,3,4,5,6,7,8,9,10]
    param int k; // = 10

predicates:
swc_pb(var bool x[], param int q[], param int k) {
    var bool s[sizeof(x)+1][k];

    //c init
    forall(j in 0..k-1) {
        !s[0][j];
    };
```

```
15
16      //c clausules 1r constraint
17      forall(i in 1..sizeof(s)-1, j in 0..k-1) {
18          s[i-1][j] -> s[i][j];
19      };
20
21      //c clausules 2n constraint
22      forall(i in 1..sizeof(s)-1, j in 0..q[i-1]-1) {
23          x[i-1] -> s[i][j];
24      };
25
26      //c clausules 3r constraint
27      forall(i in 1..sizeof(s)-1, j in 0..k-1-q[i-1]) {
28          (s[i-1][j] & x[i-1]) -> s[i][j+q[i-1]];
29      };
30
31      //c clausules 4t constraint
32      forall(i in 1..sizeof(x)) {
33          s[i-1][k-q[i-1]] -> !x[i-1];
34      };
35  }
36
37  constraints:
38      swc_pb(x, q, k);
39      ALK(x,2);
40
41  output:
42      "PB formula = " ++ [ q[i] ++ "*" ++ (x[i] ? "1" : "0") ++ (i < nVars-1 ?
            " + " : "") | i in 0..nVars-1];
43      "Result = " ++ sum([ q[i] * (x[i] ? 1 : 0) | i in 0..nVars-1]);
```

Com es tracta d'un problema SAT, s'utilitza el *solver* Glucose. El resultat de l'execució és:

```
$ ./CSP2SAT ../input/amk.bup ../input/amk.json -a=0 -s=glucose

    s SATISFIABLE
    PB formula = 1*0 + 2*0 + 3*0 + 4*1 + 5*1 + 6*0 + 7*0 + 8*0 + 9*0 + 10*0
    Result = 9
```

Com a mínim dues variables són certes (les d'índex 3 i 4) i la suma és com a molt 10 (suma igual a 9).

10.1.2 MaxSAT

En aquest exemple es modifica el model de l'apartat anterior per convertir-lo en un problema d'optimització MaxSAT. El que es vol és obtenir el mateix resultat però maximitzant el pes de les variables assignades a *true*. Per fer-ho, s'assignen pesos diferents al fet que una variable sigui certa segons la seva posició al vector de variables. Per tant, assumint que els coeficients de les variables estan ordenats de menor a major valor, quan major índex tingui la variable, més valor aporta al sumatori. A continuació es mostra el bloc de *constraints* del model descrit:

```
1  constraints:
2      swc_pb(x, q, k);
3      forall(i in 0..sizeof(x)-1) {
4          x[i] @i+1;
5      };
```

Com es tracta d'un problema MaxSAT, s'ha utilitzat el *solver* OpenWBO per resoldre'l:

```
$ ./CSP2SAT ../input/amk.bup ../input/amk.json -a=0 -s=openwbo
    s OPTIMUM FOUND
    o 45
    PB formula = 1*0 + 2*0 + 3*0 + 4*0 + 5*0 + 6*0 + 7*0 + 8*0 + 9*0 + 10*1
    Result = 10
```

El resultat com a màxim és igual a $K = 10$ i s'ha minimitzat el cost de les variables amb valor *false* assignat (45).

10.1.3 MaxSAT *Evaluations*

En aquest apartat es modifica el model descrit a l'apartat 10.1.1 perquè sigui equivalent a unes instàncies de la *MaxSAT Evaluations* [Bacchus et al., nd].

10.1.3.1 *Unweighted o Weigthed Partial* MaxSAT

Per convertir el problema a *Weigthed Partial* MaxSAT, es decideixen afegir *soft clauses* per tal que es maximitzi el nombre de variables amb valor *true*. Aquestes clàusules assignaran un pes igual a 1 al fet que una variable sigui certa. Per tant, cada variable amb valor *false*, aportarà 1 de cost a la solució. Així doncs, el programa cercarà l'assignació de variables que maximitza la suma de la fórmula pseudo-booleana però sempre essent, com a molt, K. A continuació es mostra el bloc de *constraints* amb les noves clàusules afegides:

```
1      // Instancia competicio maxsat unweighted
2      swc_pb(x, q, k);
3      forall(i in 0..sizeof(x)-1) {
4          x[i] @1;
5      };
```

En tractar-se d'un problema d'optimització de tipus MaxSAT, s'utilitza el *solver* OpenWBO per resoldre'l. El resultat és el següent:

```
$ ./CSP2SAT ../input/amk2.bup ../input/amk2.json -a=0 -s=openwbo

    s OPTIMUM FOUND
    o 6
```

```
PB formula = 1*1 + 2*1 + 3*1 + 4*1 + 5*0 + 6*0 + 7*0 + 8*0 + 9*0 + 10*0
Result = 10
```

El cost de la solució correspon a les 6 variables amb valor *false*. La solució és l'òptima perquè tracta d'assignar el major nombre de variables a *true* i, la manera d'aconseguir-ho, és utilitzant variables amb coeficients petits.

10.1.3.2 *Weighted* MaxSAT

Per aquesta instància, es segueix una estratègia molt semblant a l'apartat anterior però també convertint totes les *hard clauses* a *soft clauses*. És a dir, s'afegeixen les clàusules *soft* que maximitzen el nombre de variables a *cert* i es crea un nou predicat `swc_pb_w`. Aquest consisteix en el `swc_pb` però amb pesos a totes les clàusules. El pes de cada clàusula del predicat té un valor molt alt per tal que s'hagin de complir obligatòriament. A continuació es mostra el bloc de predicats i de *constraints*:

```
1  predicates:
2  swc_pb_w(var bool x[], param int q[], param int k) {
3      var bool s[sizeof(x)+1][k];
4
5      // init
6      forall(j in 0..k-1) {
7          !s[0][j] @sizeof(x)+1;
8      };
9
10     forall(i in 1..sizeof(s)-1, j in 0..k-1) {
11         s[i-1][j] -> s[i][j] @sizeof(x)+1;
12     };
13
14     forall(i in 1..sizeof(s)-1, j in 0..q[i-1]-1) {
15         x[i-1] -> s[i][j] @sizeof(x)+1;
16     };
17
18     forall(i in 1..sizeof(s)-1, j in 0..k-1-q[i-1]) {
19         (s[i-1][j] & x[i-1]) -> s[i][j+q[i-1]] @sizeof(x)+1;
20     };
21
22     forall(i in 1..sizeof(x)) {
23         s[i-1][k-q[i-1]] -> !x[i-1] @sizeof(x)+1;
24     };
25 }
26
27 constraints:
28     // Instancia competicio maxsat weighted
29     swc_pb_w(x, q, k);
30     forall(i in 0..sizeof(x)-1) {
31         x[i] @1;
32     };
```

En tractar-se d'un problema d'optimització s'utilitza el *solver* OpenWBO per resoldre'l. El resultat és el següent:

```
$ ./CSP2SAT ../input/amk3.bup ../input/amk3.json -a=0 -s=openwbo

    s OPTIMUM FOUND
    o 6
    PB formula = 1*1 + 2*1 + 3*1 + 4*1 + 5*0 + 6*0 + 7*0 + 8*0 + 9*0 + 10*0
    Result = 10
```

El resultat és i hauria de ser com el de l'apartat anterior.

10.2 *Combinatorial Auctions*

Les subhastes són un mètode de venda o assignació de recursos a clients o agents que serveixen perquè el venedor obtingui el màxim benefici possible. Existeixen diverses variants, però la més típica consisteix en què els compradors fan ofertes per un recurs o producte que ofereix el venedor i només es produeix la venda amb el client que hagi ofert la millor oferta.

En aquesta prova es codifica un cas bàsic de subhasta combinatòria, basant-se en l'exemple extret de [Bofill et al., 2013]. Suposem que existeix un conjunt amb n ítems $X = \{x_1, \ldots, x_n\}$. Els clients poden fer ofertes per un o més ítems amb un determinat valor. Per tant, una oferta té forma de tupla $o = (S, k)$, on S és un subconjunt d'ítems de X als que es dirigeix la oferta i k és el valor de la oferta. Així doncs, existeix un conjunt d'ofertes $O = \{o_1, \ldots, o_j\}$. Els diferents subconjunts que formen les ofertes poden no ser disjunts.

Per exemple, suposem el conjunt d'ítems $\{x_1, x_2, x_3\}$. Algunes ofertes d'aquesta subhasta podrien ser $o_1 = (\{x_1, x_3\}, 10)$, $o_2 = (\{x_2\}, 20)$, $o_3 = (\{x_2, x_3\}, 5)$.

Les restriccions que s'imposen a la subhasta són:

- **Ofertes incompatibles** → Per cada parella d'ofertes o_i i o_j on $i \neq j$, tals que els subconjunt d'ítems pels qual apliquen no siguin disjunts, és a dir, $S_i \cap S_j \neq \emptyset$, es restringeix que només una de les dues ofertes pugui ser guanyadora.

$$\neg o_i \vee \neg o_j$$

- **Màxim benefici** → Cada oferta o_i, en cas de no ser guanyadora, el venedor perd l'import que oferia k_i. Per tant, per maximitzar els beneficis, s'indica un cost per cada oferta no guanyada igual al seu valor. Per indicar això, s'utilitzen *soft constraints* unitaris de la següent manera:

$$(o_i, k_i)$$

El model equivalent escrit en llenguatge BUP es mostra a continuació:

```
viewpoint:
    param int nBids;
    param int nItems;
    param bool bidsItems[nBids][nItems];
    param int bidsValues[nBids];
    var bidsAccepted[nBids];
    var x;

constraints:
    //c Ofertes incompatibles
    forall(b1 in 0..nBids-1, b2 in b1+1..nBids-1) {
        if(lor([bidsItems[b1][i] and bidsItems[b2][i] | i in 0..nItems-1])) {
            !bidsAccepted[b1] | !bidsAccepted[b2];
        };
    };

    //c Maxim benefici
    forall(b in 0..nBids-1) {
        bidsAccepted[b] @bidsValues[b];
    };

output:
    "Bids:";
    ["#" ++ i ++ ": ({" ++ [ (bidsItems[i][j] ? (j ++ (j < nItems-1 ? ", " :
         "")) : "") | j in 0..nItems-1] ++ "}, " ++ bidsValues[i] ++ ")" ++ (i
          < nBids-1 ? "\n" : "") | i in 0..nBids-1];
    "Sold items = " ++ [bidsAccepted[b] and bidsItems[b][i] ? i ++ " " : "" |
          b in 0..nBids-1, i in 0..nItems-1];
    "Winner bids = " ++ [ "#" ++ b ++ " " | b in 0..nBids-1 where
         bidsAccepted[b]];
    "Seller benefit = " ++ sum([ bidsAccepted[i] ? bidsValues[i] : 0 | i in
         0..nBids-1]);
```

Si li passem la instància següent:

```
{
  "nBids" : 3,
  "nItems" : 7,
  "bidsItems" : [
    [1,1,1,0,0,0,0],
    [0,1,0,0,1,1,1],
    [0,0,0,0,1,0,1]
  ],
  "bidsValues": [25,40,10]
}
```

I l'executem amb el *solver* OpenWBO:

```
$ ./CSP2SAT ../input/auctions.bup ../input/auctions.json -a=0 -s=openwbo
    s OPTIMUM FOUND
    o 35
    Bids:
    #0: ({0, 1, 2, }, 25)
    #1: ({1, 4, 5, 6}, 40)
    #2: ({4, 6}, 10)
    Sold items = 1 4 5 6
    Winner bids = #1
    Seller benefit = 40
```

CAPÍTOL 11

Conclusions

L'objectiu d'aquest projecte era crear una eina que permetés modelar i resoldre CSPs d'una manera convenient. Aquest objectiu es va assolir mitjançant la correcció, millora i ampliació del *Girona Optimization System*, un projecte en desenvolupament al grup de recerca LIA del departament IMAE que consistia en el resultat d'un anterior TFG.

El procés de desenvolupament va requerir una part important d'estudi, lectura i comprensió del sistema inicial. Això és perquè GOS tenia un disseny complex, és a dir, feia ús de molts conceptes teòrics avançats i estava construït amb llenguatges i llibreries que, per fer-ne un ús correcte, requerien dominar-les notablement. Aquest estudi es va fer sobretot al principi, per poder entendre el disseny i estructura de l'aplicació, però també va estar present al llarg de tot el desenvolupament. Això va ser imprescindible per poder implementar les noves millores i ampliacions de manera correcta i eficient.

Del plantejament inicial, es van descartar les millores opcionals per poder implementar els requeriments que van sorgir durant el desenvolupament, que resultaven prioritaris pel grup de recerca LIA. Aquestes eren la codificació a SMT, la millora de la plataforma web i la generació de documentació en LaTeX. La primera i la darrera implicaven un temps de desenvolupament massa gran i la segona va resultar impossible per no tenir accés al codi font del compilador *online*.

Gràcies a la dedicació per entendre el projecte, vaig poder implementar tots els requeriments obligatoris que es plantejaven per aquest treball i, fins i tot, d'altres opcionals que van sorgir durant el desenvolupament. Per exemple, va ser imprescindible l'estudi del codi per poder trobar el problema que feia que GOS trigués un temps no raonable en certes instàncies.

Aquest projecte deixava com a resultat un GOS completament funcional que incorporava característiques noves i millores en el rendiment i estructura. Això el convertia en un sistema més flexible, eficient i aplicable tant en entorns reals com en l'àmbit de la docència i la recerca. Cal destacar que GOS es requeria per ser utilitzat com a eina docent en assignatures com PDA i, gràcies a les noves millores, això seria possible. Algunes de les característiques noves més rellevants eren la possibilitat de definir predicats personalitzats, inclusió de fitxers externs, obtenció d'informació de *debug*, flexibilitat alhora d'escollir el *solver* o possibilitat de modelar problemes MaxSAT.

Tot i això, el sistema encara tenia marge de millora. Per això es proposen algunes característiques que s'hi podrien implementar com a treball futur. Addicionalment, es va deixar el projecte millor estructurat i ben comentat en aquesta memòria per facilitar-ne la col·laboració.

CAPÍTOL 12
Treball futur

12.1 Noves estructures de dades

Actualment BUP suporta vectors de múltiples dimensions que contingui qualsevol dels tipus suportats. Tot i que és suficient per implementar la majoria de models, seria molt convenient poder disposar d'estructures de dades més sofisticades que facilitarien el modelatge d'alguns problemes complexos. Algunes propostes són conjunts o enumeracions.

12.2 Adaptació a nous formats DIMACS

La versió actual del sistema GOS suporta el format DIMACS descrit a la secció 7.9. Els *solvers* més actuals implementen diverses variacions d'aquest format. En concret, s'utilitzen les regles d'entrada i sortida de la MaxSAT *Evaluation*, la qual ha canviat les regles en els darrers anys. Per aquesta raó, seria convenient que GOS pogués adaptar-se als nous formats DIMACS per comunicar-se amb els *solvers*. Una altra alternativa seria implementar un nou mètode de comunicació que no necessités un fitxer DIMACS intermedi.

12.3 Millora de l'eficiència

Tot i que en aquest projecte s'ha millorat notablement el rendiment i l'eficiència del sistema GOS, encara hi ha alguns detalls que poden millorar-se. Per exemple, podria fer-se una reescriptura més a fons del compilador per utilitzar estructures de dades més eficients i prescindir de l'ús del *heap* el màxim possible, donat que la gestió de la memòria dinàmica és molt costosa.

12.4 Adaptació per SAT-IT

Donat que GOS es plantejava també com a eina docent, es proposa l'adaptació o la incorporació a l'aplicació SAT-IT per formar un sistema d'ajut a la docència complet. Aquesta eina permet a l'usuari resoldre pas a pas, mitjançant diferents

algorismes de resolució, fórmules codificades en DIMACS. Per tant, seria molt interessant que la sortida de GOS s'enllacés com a entrada de l'aplicació SAT-IT per estudiar el procés complet de modelat i resolució de problemes CSP.

12.5 Creació de llibreria de predicats

Amb la incorporació a GOS de la possibilitat de definir predicats personalitzats, s'ha dotat al sistema d'una gran flexibilitat i s'ha tornat potencialment utilitzable en aplicacions més exigents. Perquè GOS fos un sistema més convenient per l'usuari, seria adient implementar llibreries de predicats que facilitessin l'escriptura de codi BUP. Llavors, per utilitzar aquestes llibreries dins el codi, només caldria incloure-les.

12.6 Generació de documentació LaTeX

Es proposa la generació de documentació LaTeXdel model expressat amb BUP. Això permetria a l'usuari expressar el model codificat formalment amb documentació matemàtica d'una manera convenient. Per fer-ho, es proposa seguir l'esquema de compilació implementat a GOS i substituir les etapes d'anàlisi semàntic i generació de codi perquè, en comptes de compilar una fórmula, generés documentació.

12.7 Aplicació multiplataforma

Per tal que GOS fos un sistema accessible a tots els usuaris, seria convenient que es pogués compilar i executar en altres plataformes diferents a Linux. En aquest projecte s'ha fet un primer pas per la compilació en Windows, aportant directives de construcció CMake per la conversió de *paths*. Tot i això, seria necessari un estudi i reescriptura del codi per adaptar-lo a les necessitats de cada sistema. Una proposta seria utilitzant les directives condicionals del preprocessador de C++ per poder fer una compilació amb el codi necessari en cada sistema.

Annex

Gramàtica BUP

```
grammar BUP;

WS
   : [ \t\n\r] + -> skip
   ;

DIMACS_LINE_COMMENT : '//c ' ~[\r\n]*;
LINE_COMMENT : '//' ~[\r\n]* -> skip;
BLOCK_COMMENT : '/*' .*? '*/' -> skip;

// basic structure
TK_ENTITIES: 'entities';
TK_VIEWPOINT: 'viewpoint';
TK_PREDICATES: 'predicates';
TK_CONSTRAINTS: 'constraints';
TK_OUTPUT: 'output';

TK_COLON: ':';
TK_SEMICOLON: ';';

TK_UNDERSCORE: '_';

TK_ASSIGN: ':=';

TK_PARAM: 'param';
TK_VAR: 'var';
TK_AUX: 'aux';

TK_INT_VALUE: ('1'..'9')('0'..'9')* | '0';
TK_BOOLEAN_VALUE: 'true' | 'false';

TK_BASE_TYPE_INT : 'int';
TK_BASE_TYPE_BOOL : 'bool';

TK_IN: 'in';
TK_RANGE_DOTS: '..';

TK_IF: 'if';
TK_ELSEIF: 'else if';
TK_ELSE: 'else';

TK_LPAREN: '(';
TK_RPAREN: ')';

TK_LCLAUDATOR: '[';
TK_RCLAUDATOR: ']';

TK_LBRACKET: '{';
TK_RBRACKET: '}';

TK_COMMA: ',';
TK_DOT: '.';
```

```
53
54  TK_WHERE: 'where';
55
56  TK_FORALL: 'forall';
57
58  TK_INCLUDE: 'include';
59
60  //EXPRESSIONS
61  TK_OP_AGG_SIZEOF: 'sizeof';
62  TK_OP_AGG_SUM: 'sum';
63  TK_OP_AGG_LENGTH: 'length';
64  TK_OP_AGG_MAX: 'max';
65  TK_OP_AGG_MIN: 'min';
66  TK_OP_AGG_OR: 'lor';
67  TK_OP_AGG_AND: 'land';
68
69  TK_OP_LOGIC_NOT: 'not';
70  TK_OP_LOGIC_AND: 'and';
71  TK_OP_LOGIC_OR: 'or';
72
73  TK_OP_ARIT_SUM: '+';
74  TK_OP_ARIT_DIFF: '-';
75  TK_OP_ARIT_MULT: '*';
76  TK_OP_ARIT_DIV: '/';
77  TK_OP_ARIT_MOD: '%';
78
79  TK_OP_REL_LT: '<';
80  TK_OP_REL_GT: '>';
81  TK_OP_REL_GE: '>=';
82  TK_OP_REL_LE: '<=';
83  TK_OP_REL_EQ: '==';
84  TK_OP_REL_NEQ: '!=';
85
86  TK_OP_IMPLIC_R: '->';
87  TK_OP_IMPLIC_L: '<-';
88  TK_OP_DOUBLE_IMPLIC: '<->';
89
90  TK_INTERROGANT: '?';
91
92  TK_CONSTRAINT_OR_PIPE: '|';
93  TK_CONSTRAINT_AND: '&';
94  TK_CONSTRAINT_NOT: '!';
95
96  TK_CONSTRAINT_AGG_EK : 'EK';
97  TK_CONSTRAINT_AGG_EO : 'EO';
98  TK_CONSTRAINT_AGG_ALK : 'ALK';
99  TK_CONSTRAINT_AGG_ALO : 'ALO';
100 TK_CONSTRAINT_AGG_AMK : 'AMK';
101 TK_CONSTRAINT_AGG_AMO : 'AMO';
102
103 TK_WEIGHT : '@';
104
105 TK_IDENT: ( ('a'..'z' | 'A'..'Z' | '_')('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
        );
106
107
108 //OUTPUT
109 fragment ESCAPED_QUOTE : '\\"';
110 TK_STRING :  '"' ( ESCAPED_QUOTE | ~('"') )*? '"';
111
112 TK_STRING_AGG_OP: '++';
113
114 // SINTCTIC
115
```

```
116  csp2sat: entityDefinitionBlock? viewpointBlock predDefBlock?
         constraintDefinitionBlock outputBlock?;
117
118  entityDefinitionBlock: TK_ENTITIES TK_COLON entityDefinition* ;
119  entityDefinition: name=TK_IDENT TK_LBRACKET (definition TK_SEMICOLON)*
         TK_RBRACKET TK_SEMICOLON;
120
121  viewpointBlock: TK_VIEWPOINT TK_COLON (definition TK_SEMICOLON)*;
122
123  predDefBlock: TK_PREDICATES TK_COLON predDefBlockBody;
124  predDefBlockBody: (predDef | predInclude)*;
125  predDef: name=TK_IDENT TK_LPAREN predDefParams? TK_RPAREN TK_LBRACKET
         predDefBody TK_RBRACKET;
126  predDefParams: definition (TK_COMMA definition)*;
127  predDefBody: predVarDefinitionBlock constraintDefinition+;
128  predCall: name=TK_IDENT TK_LPAREN predCallParams? TK_RPAREN;
129  predCallParams: predCallParam (TK_COMMA predCallParam)*;
130  predCallParam:
131      varAccess
132      | expr
133      | list;
134  predVarDefinitionBlock: (varDefinition TK_SEMICOLON)*;
135  predInclude: TK_INCLUDE TK_STRING TK_SEMICOLON;
136
137  constraintDefinitionBlock: TK_CONSTRAINTS TK_COLON constraintDefinition*;
138
139  outputBlock: TK_OUTPUT TK_COLON (string TK_SEMICOLON)*;
140
141  definition: varDefinition | paramDefinition;
142  varDefinition: TK_VAR type=TK_BASE_TYPE_BOOL? name=TK_IDENT arrayDefinition;
143  paramDefinition: (
144          TK_PARAM type=(TK_BASE_TYPE_BOOL | TK_BASE_TYPE_INT)
145          | type=TK_IDENT
146      ) name=TK_IDENT arrayDefinition;
147  arrayDefinition: (TK_LCLAUDATOR arraySize=expr? TK_RCLAUDATOR)*;
148
149  // EXPRESSIONS
150
151  expr: condition=exprOr (TK_INTERROGANT op1=expr TK_COLON op2=expr)?; //
         Ternary
152
153  opAggregateExpr:
154      TK_OP_AGG_LENGTH | TK_OP_AGG_MAX | TK_OP_AGG_MIN | TK_OP_AGG_SUM
155      | TK_OP_AGG_SIZEOF | TK_OP_AGG_OR | TK_OP_AGG_AND;
156  exprListAgg: opAggregateExpr TK_LPAREN list TK_RPAREN;
157
158  exprOr: exprAnd (TK_OP_LOGIC_OR exprAnd)*;
159  exprAnd: exprEq (TK_OP_LOGIC_AND exprEq)*;
160
161  opEquality: TK_OP_REL_EQ | TK_OP_REL_NEQ;
162  exprEq: exprRel (opEquality exprRel)*;
163
164  opRelational: TK_OP_REL_LT | TK_OP_REL_GT | TK_OP_REL_GE | TK_OP_REL_LE;
165  exprRel: exprSumDiff (opRelational exprSumDiff)*;
166
167  opSumDiff : TK_OP_ARIT_SUM | TK_OP_ARIT_DIFF;
168  exprSumDiff: exprMulDivMod (opSumDiff exprMulDivMod)*;
169
170  opMulDivMod: TK_OP_ARIT_MULT | TK_OP_ARIT_DIV | TK_OP_ARIT_MOD;
171  exprMulDivMod: exprNot (opMulDivMod exprNot)*;
172
173  exprNot: op=TK_OP_LOGIC_NOT? expr_base;
174
```

```
175 expr_base: valueBaseType | TK_LPAREN expr TK_RPAREN | varAccess | exprListAgg
        ;
176
177 varAccess: id=TK_IDENT varAccessObjectOrArray*;
178
179 varAccessObjectOrArray:
180     TK_DOT attr=TK_IDENT
181     | TK_LCLAUDATOR index=expr TK_RCLAUDATOR
182     | TK_LCLAUDATOR underscore=TK_UNDERSCORE TK_RCLAUDATOR;
183
184 valueBaseType: integer=TK_INT_VALUE | boolean=TK_BOOLEAN_VALUE;
185
186
187 // CONSTRAINTS
188
189 constraintDefinition:
190     ( forall | ifThenElse | constraint ) TK_SEMICOLON
191     | DIMACS_LINE_COMMENT;
192
193 auxiliarListAssignation: name=TK_IDENT TK_IN list;
194
195 localConstraintDefinitionBlock: constraintDefinition*;
196
197 forall: TK_FORALL TK_LPAREN auxiliarListAssignation (TK_COMMA
        auxiliarListAssignation)* TK_RPAREN TK_LBRACKET
        localConstraintDefinitionBlock TK_RBRACKET;
198
199 ifThenElse:
200     TK_IF TK_LPAREN expr TK_RPAREN TK_LBRACKET localConstraintDefinitionBlock
            TK_RBRACKET
201     (TK_ELSEIF TK_LPAREN expr TK_RPAREN TK_LBRACKET
            localConstraintDefinitionBlock TK_RBRACKET)*
202     (TK_ELSE TK_LBRACKET localConstraintDefinitionBlock TK_RBRACKET)?;
203
204 list: min=expr TK_RANGE_DOTS max=expr #rangList
205     | TK_LCLAUDATOR listResultExpr TK_CONSTRAINT_OR_PIPE
            auxiliarListAssignation (TK_COMMA auxiliarListAssignation)* (TK_WHERE
             condExpr=expr)? TK_RCLAUDATOR #comprehensionList
206     | TK_LCLAUDATOR listResultExpr (TK_COMMA listResultExpr)* TK_RCLAUDATOR #
            explicitList
207     | varAccess #varAccessList;
208
209
210 listResultExpr:
211     varAcc=varAccess
212     | resExpr=expr
213     | constraint_expression
214     | string;
215
216 weight: TK_WEIGHT expr;
217 constraint:
218     constraint_expression weight?
219     | predCall
220     | constraint_aggreggate_op;
221
222 constraint_expression: constraint_double_implication;
223
224 constraint_double_implication: constraint_implication (TK_OP_DOUBLE_IMPLIC
        constraint_implication)*;
225
226
227 implication_operator: (TK_OP_IMPLIC_L | TK_OP_IMPLIC_R);
228 constraint_implication: constraint_or (implication_operator constraint_or)*;
229
```

```
230
231 constraint_or: constraint_or_2 (TK_CONSTRAINT_OR_PIPE constraint_or_2)* #
        cOrExpression;
232
233 constraint_or_2:
234     TK_CONSTRAINT_OR_PIPE TK_CONSTRAINT_OR_PIPE TK_LPAREN list TK_RPAREN #
            cOrList
235     | constraint_and #cAnd;
236
237
238 constraint_and: constraint_and_2 (TK_CONSTRAINT_AND constraint_and_2)* #
        cAndExpression;
239
240 constraint_and_2:
241     TK_CONSTRAINT_AND TK_CONSTRAINT_AND TK_LPAREN list TK_RPAREN #cAndList
242     | constraint_literal #cLit;
243
244 constraint_literal: TK_CONSTRAINT_NOT? constraint_base;
245
246 constraint_base:
247     varAccess
248     | TK_BOOLEAN_VALUE
249     | TK_LPAREN constraint_expression TK_RPAREN;
250
251 aggregate_op:
252     TK_CONSTRAINT_AGG_EK
253     | TK_CONSTRAINT_AGG_EO
254     | TK_CONSTRAINT_AGG_AMK
255     | TK_CONSTRAINT_AGG_AMO
256     | TK_CONSTRAINT_AGG_ALK
257     | TK_CONSTRAINT_AGG_ALO;
258
259 constraint_aggreggate_op: aggregate_op TK_LPAREN list (TK_COMMA param=expr)?
        TK_RPAREN;
260
261
262 //OUTPUT
263
264 string:
265     string concatString
266     | TK_LPAREN string TK_RPAREN
267     | stringTernary
268     | varAccess
269     | expr
270     | list
271     | TK_STRING;
272
273 stringTernary:
274     condition=exprAnd TK_INTERROGANT op1=string TK_COLON op2=string;
275
276 concatString:
277     TK_STRING_AGG_OP string concatString?;
```

Manual d’usuari

Instal·lació

Instal·lació ràpida

Per una instal·lació ràpida del sistema GOS, amb els paràmetres recomanats cal descarregar una còpia del codi des del repositori de GitHub. A contiuació cal situar-se en el directori arrel del projecte executar les comandes següents:

```
mkdir build
cd build
cmake ..
make -j<#threads>
```

En el directori actual apareixerà l’executable *CSP2SAT* i un directori *solvers* amb els binaris dels *solvers* Glucose, MiniSAT i OpenWBO a dins.

Instal·lació detallada

Per poder utilitzar el sistema GOS cal descarregar-se una còpia del codi font des del repositori de GitHub. Un cop obtinguda, cal compilar el projecte. Per fer-ho, és necessari situar-se primer en el directori arrel del projecte. Llavors es recomana crear un directori `build` per posar-hi tots els arxius compilats. Per fer-ho, s’han d’executar les següents comandes:

```
mkdir build
cd build
```

A continuació cal configurar CMake amb els paràmetres desitjats. El projecte GOS es pot configurar perquè es construeixi amb els *solvers* externs en versió concurrent o d’un sol fil d’execució mitjançant el *flag* `-DSOLVERS_PARALLEL=true`. També es pot definir que es compili GOS en versió *debug* o *release* amb el *flag* `-DCMAKE_BUILD_TYPE=Debug` o `-DCMAKE_BUILD_TYPE=Release`.Per defecte, es configura per la construcció amb concurrència activada i en mode *release*. La configuració de CMake es realitza amb la comanda següent:

```
cmake .. [flags]
```

Un cop finalitzada la configuració, es pot construir el projecte amb l'eina Make. Es recomana habilitar la construcció concurrent amb el *flag* `-j<#threads>`, on `<#threads>` són els fils d'execució que Make pot utilitzar. Es poden especificar diferents *targets* o objectius de construcció. Si no s'indica *target*, per defecte es construeix tot el sistema GOS i els *solvers* externs Glucose, OpenWBO i MiniSAT. Tot i això, és possible compilar només un *solver* específic indicant el seu nom en minúscules com a *target*. Per construir el projecte, cal executar:

```
1  make [target] [flags]
```

Un cop finalitzada la construcció, s'haurà generat un fitxer executable anomenat *CSP2SAT*, que és el fitxer binari del sistema GOS. Addicionalment, s'haurà generat un directori *./solvers*, on s'hi trobaran els executables dels *solvers*.

Utilització

Per utilitzar el sistema GOS, cal invocar l'executable des de la línia de comandes, indicant també un fitxer model codificat en llenguatge BUP i un fitxer d'instància amb les dades en format JSON. La sintaxi bàsica de la crida al sistema és:

```
1  CSP2SAT <fitxer_model> <fitxer_instancia> [flags]
```

Els *flags* bàsics són:

- `-h` → Mostra un missatge d'ajuda.
- `-s=<string>` → Especifica el *solver* a utilitzar. Disponibles amb API: glucose, minisat. Disponibles sense API: glucose, openwbo, custom. Per defecte: glucose.
- `-a=<bool>` → Indica si GOS ha d'utilitzar l'API dels *solvers* per resoldre la fórmula compilada. Valors possibles: 1, 0: Per defecte: 1.
- `-c=<string>` → Especifica la comanda personalitzada entre cometes dobles per executar un *solver* extern a GOS. El resultat de la comanda ha de ser en format DIMACS. Només s'utilitza quan el *solver* és *custom*.
- `-d=<bool>` → Activa o desactiva la generació d'informació de *debug* en la codificació de la fórmula en DIMACS. Informació disponible: *id* de les variables, valor dels paràmetres i comentaris escrits al model amb "//c ". Per defecte: 1.

- `-pf=<bool>` → Indica si GOS ha de resoldre la fórmula compilada o mostrar-la en format DIMACS. Per defecte: 0.

A més d'aquests *flags*, n'hi ha d'altres de secundaris que permeten configurar detalls interns del procés de *solving*. Aquests es mostren en el missatge d'ajuda de l'aplicació.

Bibliografia

[Albatross, nd] Albatross (n.d.). History of C++. https://cplusplus.com/info/history/. (Accedit: 12/08/2022).

[Bacchus et al., nd] Bacchus, F., Järvisalo, M., and Martins, R. (n.d.). MaxSAT Evaluations. https://maxsat-evaluations.github.io. (Accedit: 10/08/2022).

[Banbic, 1993] Banbic, D. (1993). Satisfiability Suggested Format. http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf/. (Accedit: 26/08/2022).

[Barták, nd] Barták, R. (n.d.). Constraint propagation and backtracking-based search. http://kti.ms.mff.cuni.cz/~bartak/downloads/CPschool05notes.pdf. (Accedit: 15/08/2022).

[Biere et al., 2009] Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors (2009). *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

[Bofill, 2020] Bofill, M. (2020). Apunts de l'assignatura de Fonaments de Computació.

[Bofill et al., 2013] Bofill, M., Busquets, D., Muñoz, V., and Villaret, M. (2013). Reformulation based maxsat robustness. *Constraints An Int. J.*, 18(2):202–235.

[Bofill et al., 2020] Bofill, M., Coll, J., Nightingale, P., Suy, J., Ulrich-Oltean, F., and Villaret, M. (2020). SAT encodings for Pseudo-Boolean constraints together with at-most-one constraints. page 7.

[CMake, nd] CMake (n.d.). About CMake. https://cmake.org/overview/. (Accedit: 15/08/2022).

[D-Wave Ocean, nd] D-Wave Ocean (n.d.). Map Coloring . https://docs.ocean.dwavesys.com/en/stable/examples/map_coloring.html. (Accedit: 16/08/2022).

[Drumond, nd] Drumond, C. (n.d.). What is Scrum? https://www.atlassian.com/agile/scrum. (Accedit: 17/08/2022).

[GDB Developers, nd] GDB Developers (n.d.). What is GDB? https://www.sourceware.org/gdb/. (Accedit: 08/08/2022).

[Generoso, 2020] Generoso, R. (2020). GOS. A new declarative tool for modelling and solving CSPs to SAT. Treball final de grau, Universitat de Girona.

[IBM, 2021] IBM (2021). Preprocessor directives. https://www.ibm.com/docs/en/zos/2.4.0?topic=reference-preprocessor-directives. (Accedit: 15/08/2022).

[Li et al., 2022] Li, C.-M., Xu, Z., Coll, J., Manyà, F., Habet, D., and He, K. (2022). Combining clause learning and branch and bound for maxsat (extended abstract). In Raedt, L. D., editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 5299–5303. International Joint Conferences on Artificial Intelligence Organization. Sister Conferences Best Papers.

[MaxSAT Evaluations, 2021] MaxSAT Evaluations (2021). Output format. https://maxsat-evaluations.github.io/2021/rules.html#output. (Accedit: 20/08/2022).

[Muchnick, 1997] Muchnick, S. (1997). *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers.

[Parr, nd] Parr, T. (n.d.). About the antlr parser generator. https://www.antlr.org/about.html. (Accedit: 15/08/2022).

[Perf Wiki, 2022] Perf Wiki (2022). perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. (Accedit: 07/08/2022).

[Suy, 2022] Suy, J. (2022). Apunts de l'assignatura de Compiladors.

[Tomassetti, Gabriele, nd] Tomassetti, Gabriele (n.d.). Listeners And Visitors. https://tomassetti.me/listeners-and-visitors/. (Accedit: 04/08/2022).

[Valgrind, nda] Valgrind (n.d.a). Massif: a heap profiler. https://valgrind.org/docs/manual/ms-manual.html. (Accedit: 05/08/2022).

[Valgrind, ndb] Valgrind (n.d.b). Using and understanding the Valgrind core. https://valgrind.org/docs/manual/manual-core.html#manual-core.whatdoes. (Accedit: 05/08/2022).