Alma Mater Studiorum - University of Bologna

Computer Science and Engineering - DISI

Artificial Intelligence

Extending the 2P-Kt ecosystem: CLP and Labelled LP

Master degree thesis

Supervisor

Prof.Roberta Calegari

Co-supervisor

Prof. Giovanni Ciatto

Candidate

Giuseppe Boezio

Academic Year 2021-2022- Third session

Abstract

Content of the abstract

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis organization

First chapter introduces the general content about thesis and gives a short presentation of the topic;

Second chapter a deepening about how CLP has been realized using 2P-Kt framework;

Third chapter presents the extension of Labelled Variables for every kind of term;

Fourth chapter presents how to adapt Constraint Logic Programming (CLP) with Labelled Prolog

Sixth chapter discusses about the results and possible future developments.

Chapter 2

Background notions

2.1 The Prolog Language

2.1.1 Brief History

Prolog stands for *PROgrammation en LOGique* and it emerged during 1970s as a way to use logic as a programming language. The early developers of this language were Robert Kowalski, Maarten van Emden and Alain Colmelauer. The programming language, Prolog, was born of a project aimed not at producing a programming language but at processing natural languages; in this case, French [**10.1145/155360.155362**]. The project gave rise to a preliminary version of Prolog at the end of 1971 and a more definitive version at the end of 1972. Prolog gained a lot of attraction from the computing society as it was the very first logic programming language. The language still holds considerable importance and popularity among the logic programming languages and comes with a range of commercial as well as free implementations.
Prolog is used for different kind of tasks such as:

- theorem proving [**coelho1986automated**]
- expert systems [**merritt2012building**]
- knowledge representation [**gelfond2002logic**]
- automated planning [**pinna2015resolving**]
- natural language processing [**lally2011natural**]

2.1.2 Concepts

Syntax and semantics of Prolog are described in ISO standard ISO/IEC 13211. Prolog is a logic programming language; this means that it is used to describe known facts and relationships about a problem and less about prescribing the sequence of steps taken by a computer to solve the problem. When a computer is programmed in Prolog, the actual way the computer carries out the computation is specified partially by the logic declarative semantics of Prolog, partly by what new facts can be inferred from the given ones, and only partly by explicit control information supplied by the programmer.

Prolog is used to solve problems which involve objects and relations among them. The main features of the programming language are:

- specifyng some *facts* about some objects and their relationships
- defining some *rules* about objects and their relationships
- asking *questions* about objects and their relationships

Prolog programs are built from terms. A term is either a constant, a variable or a structure.

2.1.2.1 Constants

A constant is a sequence of characters which denotes a specific object or relationship. A constant can be an atom or a number. All constants begin with a lower case letter.

Example

a is an atom
12 is a number

2.1.2.2 Variables

A variable looks like an atom except it has a name beginning with capital letter or underline signed. A variable should be thought of as standing for some objects we are unable or unwilling to name at the time we write the program.

Example

X and *Answer* are valid names for variables

2.1.2.3 Structures

A structure is a collection of other objects called *components*. Structures help to organize the data in a program because they permit a group of related information to be treated as a single object instead of separate entities. A structure is written in Prolog by specifying its *functor* and its *components*. The coponents are enclosed in round brackets and separated by commas. The functor is written just before the opening round brackets.

Example

owns(john,book) is a structure having owns as functor and, john and book as components

2.1.2.4 Facts

A fact is a relation among objects which are all *ground*. This means that a fact is a *structure* which does not contain any variables among its components. A fact is written as a structure followed by a dot (.). The names of the objects that are enclosed within the round brackets in each fact are called *arguments*. The name of the relationship which comes just before the round brackets is called *predicate*.

Example

king(john,france). is a fact

2.1.2.5 Rules

A rule is a disjunction of predicates, where at most one is not negated, written in the following way:

$$Head :- Body.$$

where Head is a predicate, Body is a conjunction of predicates and the symbol ":-" means that the body implies the head. This kind of structure is called Horn clause. A Prolog program can be seen as a list (because order matters) of Horn clauses called *theory*. Facts could be seen as Rules having Body equals to true. Rules are

used to describe some complex relations among objects of the domain of discourse and differently from facts can contain variables.

Example

motherOf(X,Y) :- parentOf(X,Y),female(X).

The aforementioned description of Prolog language has been adapted from **[Clocksin1987Programmin**

2.1.2.6 Unification

A substitution is a function which associates a variable to a given term. The most general unifier (m.g.u.) is the substitution which allows to transform two terms making them equals such that all other substitutions can be obtained through a composition with this one. The unification is a process whereby two structures are made equals via substitution and it is used several times during the Prolog resolution process.

2.1.2.7 Resolution

The resolution in Prolog happens in the following way:the interpreter tries to verify whether a conjunction of predicates (the goal) provided by the user can be derived from the current program or not and in the case it could, it provides a computed answer substitution (c.a.s.) which is a set of substitutions which allow to make true the user's goal.
Prolog resolution process is called SLD (Selective Linear Definite clause resolution) and works as follows:
SLD resolution implicitly defines a search tree of alternative computations, in which the initial goal clause is associated with the root of the tree. For every node in the tree and for every definite clause in the program whose positive literal unifies with the selected literal in the goal clause associated with the node, there is a child node associated with the goal clause obtained by SLD resolution. A leaf node, which has no children, is a success node if its associated goal clause is the empty clause. It is a failure node if its associated goal clause is non-empty but its selected literal unifies with no positive literal of definite clauses in the program. SLD resolution is non-deterministic in the sense that it does not determine the search strategy for exploring the search tree. Prolog searches the tree depth-first, one branch at a time, using backtracking when it encounters a failure node. Depth-first search is very efficient in

its use of computing resources, but is incomplete if the search space contains infinite branches and the search strategy searches these in preference to finite branches: the computation does not terminate. The SLD resolution search space is an or-tree, in which different branches represent alternative computations.[**Gallier1985LogicFC**]

2.1.3 2p-Kt

2p-Kt is a general, extensible, and interoperable ecosystem for logic programming and symbolic AI written in Kotlin which supports the Prolog ISO standard.
2p-kt is the evolution of another project called tuProlog [**CIATTO2021100817**].

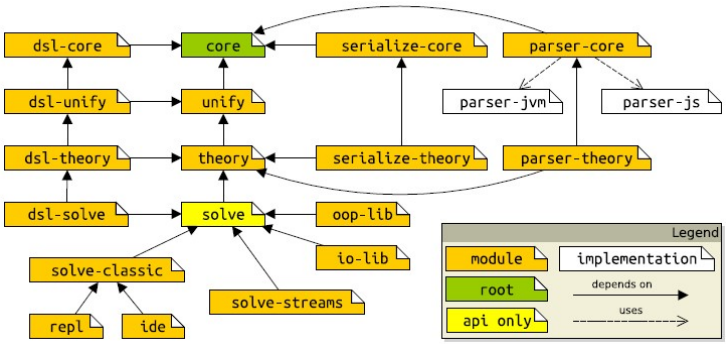

Figure 2.1: 2P-Kt project map. LP functionalities are partitioned into some loosely-coupled and incrementally-dependent modules.

To support reusability 2p-Kt is divided into several modules described as follows:

- **:core**: exposes data structures for knowledge representation via terms and clauses, other than methods supporting their manipulation
- **:unify**: used to compare and manipulate logic terms through logic unification
- **:theory**: in-memory storage of clauses into ordered (e.g. queues) or unordered (e.g. multisets) data structures, and their efficient retrieval via pattern-matching
- **:serialize-* and :parser-***: used to perform ancillary operations such as serialization and parsing
- **:solve**: aspects which are orthogonal w.r.t. any particular resolution strategy —e.g. errors management, extensibility via libraries, I/O, etc
- **:solve-***: modules which implement a specific resolution strategy

- **:repl and :ide**: provide CLI and GUI

The structure of the project can be seen in figure **??**.
2p-Kt provides a well-grounded technological basis for implementing/experimenting/extending the many solutions proposed in the literature—e.g., abductive inference, rule induction, probabilistic reasoning and labelled LP.

2.2 Constraint Programming

Constraint Programming is a paradigm for solvig combinatorial problems where different constraints are imposed on feasible solutions for different decision variables, each having its own domain. Constraints are relations among variables which limit the values decision variables can assume in feasible solutions **[10.5555/2843512]**

2.2.1 Brief History

In artificial intelligence interest in constraint satisfaction developed in two streams. In some sense a common ancestor of both streams is Ivan Sutherland' s groundbreaking 1963 MIT Ph.D. thesis, "Sketchpad: A man-machine graphical communication system" . In one stream, the versatility of constraints led to applications in a variety of domains, and associated programming languages and systems. This stream we can call the language stream. In 1964 Wilkes proposed that algebraic equations be allowed as constraint statements in procedural Algol-like programming languages, with relaxation used to satisfy the constraints. Around 1967, Elcock developed a declarative language, Absys, based on the manipulation of equational constraints.

2.2.2 Concepts

There are mainly two types of constraint programming problems: CSP (Constraint Satisfaction Problem) and COP (Constraint Optimization Problem).

A *constraint satisfaction problem* (CSP) involves finding solutions to a constraint network, that is, assignments of values to its variables that satisfy all its constraints. Constraints specify combinations of values that given subsets of variables are allowed to take.
A constraint can be specified extensionally by the list of its satisfying tuples, or intensionally by a formula that is the characteristic function of the constraint.

A *constraint optimization problem* (COP) is basically the same as a CSP but in addition to the aforementioned constraints there is another one which consists of finding a solution which minimizes or maximizes a certain function.

2.2.3 Constraint Logic Programming

Constraint Logic Programming (CLP) began as a natural merger of two declarative paradigms: constraint solving and logic programming [**JAFFAR1994503**]. Viewing the subject rather broadly, constraint logic programming can be said to involve the incorporation of constraints and constraint "solving" methods in a logic-based language. This characterization suggests the possibility of many interesting languages, based on different constraints and different logics. However, to this point, work on CLP has almost exclusively been devoted to languages based on Horn clauses.
Prolog can be said to be a CLP language where the constraints are equations over the algebra of terms (also called the algebra of finite trees, or the Herbrand domain). The equations are implicit in the use of unification.

2.2.4 SWI Prolog - CLP libraries

SWI-Prolog is an implementation of the Prolog language which is strong in education because it is free and portable, but also because of its compatibility with textbooks and its easy-to-use environment.
SWI-Prolog is used as an embedded language where it serves as a small rule subsystem in a large application. The syntax and set of built-in predicates is based on the ISO standard [**SWI-Prolog**].

2.2.4.1 CLP(X) libraries

CLP(X) stands for constraint logic programming over the domain X. Plain Prolog can be regarded as CLP(H), where H stands for Herbrand terms.
SWI Prolog supports:

- **CLP(FD)** for integers
- **CLP(B)** for boolean variables
- **CLP(Q)** for rational numbers

- **CLP(R)** for floating point numbers

All constraints contained in these libraries will be deeply explained in chapter **??**.

CLP(FD) has two main usages:

- declarative integer arithmetics
- solving combinatorial problems such as planning, scheduling and allocation tasks

The predicate of this library can be classified as:

- *arithmetic constraints*
- *membership constraints*
- *enumeration predicates*
- *combinatorial constraints*
- *reification predicates*
- *reflection predicates*

Practical usage of these constraints can be found in [**Triska12**].

CLP(Q) and CLP(R) share basically the same constraints except for *bb_inf* constraint which is used to find a minimum in the case of mixed integer programming.

CLP(B) can be used to model and solve combinatorial problems such as verification, allocation and covering tasks. Benchmarks and usage examples of this library are available from [**Triska2016**] and [**Triska2018**].

Chapter 3

CLP in 2p-Kt

3.1 Requirements

As described in section **??**, 2P-Kt is an exensible framework with different mechanisms which could be used to add to the standard Prolog other features; in this case we deal with Constraint Logic Programing (CLP).
There are existing libraries for Prolog interpreters which deal with Constraint Logic Programming. SWI Prolog uses different libraries called CLP(X) (where is X is the domain of the variables) which are very popular and used in the Prolog community. For this reason, different libraries have been developed which share the same predicates having the same functor/arity in such a way that for a new 2P-Kt user can be very easy and familiar to use them.
The main purpose of the following project is to implement CLP libraries in 2P-Kt having the following requirements:

- the interface of the predicates exposed by the libraries must be as close as possible to the one used by SWI Prolog for CLP
- there are not strict requirements about how to implement libraries but a good solution would exploit on existing components in 2P-kt
- the different libraries could contain different predicates wrt SWI Prolog only if it is not possible to find another solution which is compatible with the current framework

3.2 Design

3.2.1 Common aspects

Four different libraries have been realized:

- **clp-core** for basic functionality of the other libraries
- **clpfd** for integer variables
- **clpqr** for rational and real variables
- **clpb** for boolean variables

clpqr contains basically the predicates of **clpq** and **clpr** of SWI Prolog because there is any distinction between rational and reals in 2P-Kt.
Libraries have been developed in this way to keep separated predicates which affect variables with different domains. This is useful because each variabile type has own way to deal with constraints and searching for feasible solutions. Libraries will be described in the following sections highlighting common and/or different aspects with respect to the SWI Prolog counterpart.

3.2.2 Constraint Logic Programming over Finite Domains

For a better explaination predicates will be divided in groups as described in section **??**.

3.2.2.1 Arithmetic Constraints

All constraints supported by SWI Prolog are also supported in 2P-Kt; constraints are the followings:

Constraint	Explaination
Expr1 #= Expr2	Expr1 equals Expr2
Expr1 #\= Expr2	Expr1 is not equal to Expr2
Expr1 #>= Expr2	Expr1 is greater than or equal to Expr2
Expr1 #=< Expr2	Expr1 is less than or equal to Expr2
Expr1 #> Expr2	Expr1 is greater than Expr2
Expr1 #< Expr2	Expr1 is less than Expr2

Table 3.1: Arithmetic constraints in clpfd

Expression	Explaination
integer	Given value
variable	Unknown integer
-Expr	Unary minus
Expr + Expr	Addition
Expr * Expr	Multiplication
Expr - Expr	Subtraction
Expr Ê xpr	Exponentiation
min(Expr,Expr)	Minimum of two expressions
max(Expr,Expr)	Maximum of two expressions
Expr mod Expr	Modulo induced by floored division
abs(Expr)	Absolute value
Expr div Expr	Floored integer division

Table 3.2: Arithmetic expressions in clpfd

Expr1 and *Expr2* are **arithmetic expressions**. *Expr rem Expr* and *Expr // Expr* are not supported. *rem* is modulo induced by truncated division whereas // is truncated integer division.

3.2.2.2 Membership Constraints

These constraints are used to specify the admissible domains of variables. The predicates are:

- **Var in Domain**: Var is an element of Domain; Domain is either an integer or an interval (expressed as Lower..Upper)
- **Vars in Domain**: The variables in the list Vars are elements of Domain

It is not current supported union of domains as expression for building a domain.

3.2.2.3 Enumeration predicates

These predicates are used to customize the search to find a feasible assignment of all variables such that all constraints are satisfied.
The predicates are **labeling/2 and label/1**.

labeling(Options, Vars)

Assign a value to each variable in Vars; Options is a list of options that let exhibit some control over the search process. Several categories of options exist:

- **variable selection strategy**: it can be the order in which the variable occurs (*leftmost*, it is the default), the leftmost variable with smallest domain (*ff*), the variables with smallest domains, the leftmost one participating in most constraints (*ffc*), the leftmost variable whose lower bound is the lowest (*min*) or the leftmost variable whose upper bound is the highest (*max*)
- **value order**: elements of the chosen variable's domain in ascending order (*up*, it is the default) or domain elements in descending order (*down*)
- **branching strategy**: For each variable X, a choice is made between X = V and X $\#\bar{V}$, where V is determined by the value ordering options. This option is called *step*, it is the default and the only branching option supported

At most one option of each category can be specified, and an option must not occur repeatedly.
The order of solutions can be influenced with:

- min(Expr)
- max(Expr)

This generates solutions in ascending/descending order with respect to the evaluation of the arithmetic expression Expr.
The predicate **labeling/2** does not support as options the following branching strategies:

- *enum*: For each variable X, a choice is made between X = V_1, X = V_2 etc., for all values V_i of the domain of X. The order is determined by the value ordering options.
- *bisect*: For each variable X, a choice is made between X #=< M and X #> M, where M is the midpoint of the domain of X.

label(Vars)

Equivalent to labeling([], Vars).

3.2.2.4 Global constraints

A global constraint expresses a relation that involves many variables at once. The implemented constraints are the followings:

- **all_distinct(Vars)**: True iff Vars are pairwise distinct
- **sum(Vars, Rel, Expr)**: The sum of elements of the list Vars is in relation Rel to Expr. Rel is one of #=, #\, #<, #>, #=< or #>=
- **scalar_product(Cs, Vs, Rel, Expr)**: True iff the scalar product of Cs and Vs is in relation Rel to Expr. Cs is a list of integers, Vs is a list of variables and integers. Rel is #=, #\, #<, #>, #=< or #>=
- **lex_chain(Lists)**: Lists are lexicographically non-decreasing
- **tuples_in(Tuples, Relation)**: True iff all Tuples are elements of Relation. Each element of the list Tuples is a list of integers or finite domain variables. Relation is a list of lists of integers
- **serialized(Starts, Durations)**: Describes a set of non-overlapping tasks. Starts = $[S_1, ..., S_n]$, is a list of variables or integers, Durations = $[D_1, ..., D_n]$ is a list of non-negative integers. Constrains Starts and Durations to denote a set of non-overlapping tasks, i.e.: S_i + D_i =< S_j or S_j + D_j =< S_i for all 1 =< i < j =< n

- **element(N, Vs, V)**: The N-th element of the list of finite domain variables Vs is V
- **global_cardinality(Vs, Pairs)**: Global Cardinality constraint. Vs is a list of finite domain variables, Pairs is a list of Key-Num pairs, where Key is an integer and Num is a finite domain variable. The constraint holds iff each V in Vs is equal to some key, and for each Key-Num pair in Pairs, the number of occurrences of Key in Vs is Num
- **circuit(Vs)**: True iff the list Vs of finite domain variables induces a Hamiltonian circuit. The k-th element of Vs denotes the successor of node k
- **cumulative(Tasks, Options)**: Schedule with a limited resource. Tasks is a list of tasks, each of the form task(S_i, D_i, E_i, C_i, T_i). S_i denotes the start time, D_i the positive duration, E_i the end time, C_i the non-negative resource consumption, and T_i the task identifier. Each of these arguments must be a finite domain variable with bounded domain, or an integer. The constraint holds iff at each time slot during the start and end of each task, the total resource consumption of all tasks running at that time does not exceed the global resource limit. Options is a list of options. Currently, the only supported option is *limit(L)* which is the global resource limit
- **cumulative(Tasks)**: Like the previous one but with $L = 1$
- **disjoint2(Rectangles)**: True iff Rectangles are not overlapping. Rectangles is a list of terms of the form F(X_i, W_i, Y_i, H_i), where F is any functor, and the arguments are finite domain variables or integers that denote, respectively, the X coordinate, width, Y coordinate and height of each rectangle.
- **chain(Zs, Relation)**: Zs form a chain with respect to Relation. Zs is a list of finite domain variables that are a chain with respect to the partial order Relation, in the order they appear in the list. Relation must be #=, #=<, #>=, #< or #>

Notes

Wrt global constraints provided by CLP(FD) library of SWI Prolog the following aspects are different:

- the index of the predicate **circuit/1** starts from 1 and not from 0 for implementation issues

- the predicate **all_different/1** has not been supported because it has the same usage of **all_distinct** but it has a weaker propagation which cannot be simulated
- predicates **automaton/3** and **automaton/8** have not been implemented because of the fact that these predicates are rarely and difficult to use
- the predicate **global_cardinality/3** can be used but actually it throws an exception because the Option parameter cannot be supported

3.2.2.5 Reification Predicates

All relational constraints discussed in **??** can be reified. This means that their truth value is itself turned into a clpfd variable, so that it is possible to reason about whether a constraint holds or not. These predicates are reifiable themselves.

- # **Q**: Q does not hold
- **P** #<==> **Q**: P and Q are equivalent
- **P** #==> **Q**: P implies Q
- **P** #<== **Q**: Q implies P
- **P** #/ **Q**: P and Q hold
- **P** # **Q**: P or Q hold
- **P** # **Q**: Either P holds or Q holds, but not both
- **zcompare(Order, A, B)**: reify an arithmetic comparison of two integers

3.2.2.6 Reflection Predicates

Reflection predicates let obtain, in a well-defined way, information that is normally internal to this library. Predicates are:

- **fd_var(Var)**: True iff Var is a clpfd variable
- **fd_inf(Var, Inf)**: Inf is the infimum of the current domain of Var
- **fd_sup(Var, Sup)**: Sup is the supremum of the current domain of Var
- **fd_size(Var, Size)**: Reflect the current size of a domain. Size is the number of elements of the current domain of Var

- **fd_dom(Var, Dom)**: Dom is the current domain (see **??**) of Var
- **fd_degree(Var, Degree)**: Degree is the number of constraints currently attached to Var

3.2.3 Constraint Logic Programming over Rationals and Reals

This library is very different from the previous one because variable definitions and constraints can be stated in the same predicate. The main predicate is **{}(Constraints)** which allows to add the constraints given by Constraints to the constraint store.
Constraints can be defined using the following grammar:

<*Constraints*>	::=	<*Constraint*>	single constraint
	|	<*Constraint*> , <*Constraints*>	conjunction
	|	<*Constraint*> ; <*Constraints*>	disjunction
<*Constraint*>	::=	<*Expression*> < <*Expression*>	less than
	|	<*Expression*> > <*Expression*>	greater than
	|	<*Expression*> =< <*Expression*>	less or equal
	|	<=(<*Expression*>, <*Expression*>)	less or equal
	|	<*Expression*> >= <*Expression*>	greater or equal
	|	<*Expression*> =\= <*Expression*>	not equal
	|	<*Expression*> =:= <*Expression*>	equal
	|	<*Expression*> = <*Expression*>	equal
<*Expression*>	::=	<*Variable*>	Prolog variable
	|	<*Number*>	Prolog number
	|	+<*Expression*>	unary plus
	|	-<*Expression*>	unary minus
	|	<*Expression*> + <*Expression*>	addition
	|	<*Expression*> - <*Expression*>	substraction
	|	<*Expression*> * <*Expression*>	multiplication
	|	<*Expression*> / <*Expression*>	division
	|	abs(<*Expression*>)	absolute value
	|	sin(<*Expression*>)	sine
	|	cos(<*Expression*>)	cosine
	|	tan(<*Expression*>)	tangent
	|	exp(<*Expression*>)	exponent
	|	pow(<*Expression*>)	exponent
	|	<*Expression*> ^ <*Expression*>	exponent
	|	min(<*Expression*>, <*Expression*>)	minimum
	|	max(<*Expression*>, <*Expression*>)	maximum

Figure 3.1: clpqr constraints BNF

All constraints are supported except for <*Expression*> =
= <*Expression*> (not equal). This libraries contains also the following predicates:

- **satisfy(Vars)**: Provides for each variable in Vars a feasible assignment
- **entailed(Constraint)**: Succeeds if Constraint is necessarily true within the current constraint store. This means that adding the negation of the constraint to the store results in failure
- **inf(Expression, Inf)**: Computes the infimum of Expression within the current state of the constraint store and returns that infimum in Inf. This predicate does not change the constraint store
- **sup(Expression, Sup)**: Computes the supremum of Expression within the current state of the constraint store and returns that supremum in Sup. This predicate does not change the constraint store
- **minimize(Expression)**: Minimizes Expression within the current constraint store. This is the same as computing the infimum and equating the expression to that infimum
- **maximize(Expression)**: Maximizes Expression within the current constraint store. This is the same as computing the supremum and equating the expression to that supremum
- **bb_inf(Ints, Expression, Inf, Vertex, Eps)**: It computes the infimum of Expression within the current constraint store, with the additional constraint that in that infimum, all variables in Ints have integral values. Vertex will contain the values of Ints in the infimum. Eps denotes how much a value may differ from an integer to be considered an integer
- **bb_inf(Ints, Expression, Inf, Vertex)**: it behaves as the previous one but not use an error margin
- **bb_inf(Ints, Expression, Inf)**: as the previous one but without returning the values of the integers
- **dump(Target, Newvars, CodedAnswer)**: Returns the constraints on Target in the list CodedAnswer where all variables of Target have been replaced by NewVars. This operation does not change the constraint store

Notes

Eps of the predicate **bb_inf/5** cannot be fully supported, the only admissible value is 0.

Expression	Explaination
0	false
1	true
variable	unknown truth value
Expr	logical NOT
Expr + Expr	logical OR
Expr * Expr	logical AND
Expr # Expr	exclusive OR
Expr =:= Expr	equality
Expr =\= Expr	disequality (same as #)
Expr =< Expr	less or equal (implication)
Expr >= Expr	greater or equal
Expr < Expr	less than
Expr > Expr	greater than
+(Exprs)	n-fold disjunction
*(Exprs)	n-fold conjunction

Table 3.3: Admissible boolean expressions in clpb

3.2.4 Constraint Logic Programming over Boolean Variables

All predicates of this library are based on the concept of *boolean expression*. A *boolean expression* is one of:

Supported predicates are:

- **sat(Expr)**: True iff the Boolean expression Expr is satisfiable
- **taut(Expr, T)**: If Expr is a tautology with respect to the posted constraints, succeeds with T = 1. If Expr cannot be satisfied, succeeds with T = 0. Otherwise, it fails
- **labeling(Vs)**: Assigns truth values to the variables Vs such that all constraints are satisfied
- **sat_count(Expr, Count)**: Count the number of admissible assignments. Count is the number of different assignments of truth values to the variables in the Boolean expression Expr, such that Expr is true and all posted constraints are satisfiable

- **weighted_maximum(Weights, Vs, Maximum)**: Enumerate weighted optima over admissible assignments. Maximize a linear objective function over Boolean variables Vs with integer coefficients Weights. This predicate assigns 0 and 1 to the variables in Vs such that all stated constraints are satisfied, and Maximum is the maximum of $sum(Weight_i * V_i)$ over all admissible assignments

- **random_labeling(Seed, Vs)**: Select a single random solution. An admissible assignment of truth values to the Boolean variables in Vs is chosen in such a way that each admissible assignment is equally likely. Seed is an integer, used as the initial seed for the random number generator

3.3 Implementation

The aforementioned libraries have been implemented in 2P-Kt using the following interfaces and classes:

- **PrimitiveWrapper**: it is an abstract class which allows to define a predicate which is not unified as usual but it executes some code producing some side effects on the actual substitution. This class reifies the generator concept described in [**10.1007/978-3-030-75775-5_27**] where the solver can be seen as a stream consumer allowing it to get a stream of solutions interacting with a primitive mechanism (the generator) which can be seen by the solver as an ordinary build-in predicate denoted by its own *signature* and *arity*. The PrimitiveWrapper has been used to define most of predicates contained in the libraries.

- **RuleWrapper**: it is an abstract class which allows to define a Prolog *Rule*. This class is useful to avoid repeated code because it allows to define some predicates in terms of other existing ones.

- **Library**: it is an interface which has been implemented to group together different predicates implementations which can be either *PrimitiveWrapper* or *RuleWrapper*

- **durable CustomData**: it is a map from String to Any which allows to store data during the resolution process. It has been used to store the model containing all variables and constraints of the problem

- **DefaultTermVisitor**: this abstract class is used to realize the visitor pattern [**gamma1994design**] and it has been extensively used for different tasks such

as evaluation of expressions, generation of arithmetic expressions and all other tasks related to perform different operations wrt the type of the term

The actual implementation of all constraints have been realized exploiting on the Choco Library [**Prudhomme2022**]. This library has been chosen for different reasons:

- it is written in Java and can be easily used with Kotlin language
- wrt other similar libraries (e.g. OR Tools [**ortools**] or JaCoP [**Kuchcinski2013JaCoPJ**]) it simplifies the composition of expressions for the creation of new variables or constraints
- well documented and with an active community to get support in case of doubts or problems

Choco Library cannot be directly mapped with SWI Prolog CLP(X) libraries because these libraries belong to different paradigms. For this reason the mapping was not immediate but sometimes different approaches have been used to solve these issues.

Following, the main mapping choices grouped by Library

3.3.1 Constraint Logic Programming over Finite Domains

The **Model** class of Choco has been used to keep all information about variables and constraints.
in/2 and **ins/2** predicates have been mapped to the method **intVar(name,lower_bound,upper_bound)** which allows to store the different variables.

Each **relational constraint** can be added to the model using **decompose().post()** applied to the constraint which belongs to the class **ReExpression** whereas the different arguments of these constraints are **arithmetical expressions** belonging to the class **ArExpression Global constraints** have been mapped using different methods provided by the **Model** class, they are summarized in table **?? enumeration predicates** have been implemented using **Search.intVarSearch** which allows to customize the searching process.

SWI Prolog	Choco Library
all_distinct(Vars)	allDifferent(Vars)
sum(Vars,Op,Result)	sum(Vars,Op,Result)
scalar_product(Coeffs, Vars, Op, Result)	scalar(Vars, Coeffs, Op, Result)
lex_chain([List1, List2])	lexLessEq(List1, List2)
tuples_in([Tuple], Relation)	table(Tuple, Relation)
serialized(Starts, Durations)	diffN(Starts, Zeros, Durations, Ones)
element(N, Vs, V)	element(V, Vs, N)
global_cardinality(Vs, Pairs)	global_cardinality(Vs, Values, Occurrences)
circuit(Variables)	circuit(Variables)
cumulative(Tasks, Options)	cumulative(Tasks, Capacities, GlobalCapacity)
disjoint2(Rectangles)	diffN(XCoordinates, YCoordinates, Lengths, Heights)
chain(Variables, Relation)	binary relational constraint for each pair of variables

Table 3.4: Global constraints mapping between CLP(FD) and Choco Library

reification predicates have been implemented converting relational constraints to boolean variables and combining them using **LogOp** logical operators. **reflection predicates** have been mapped using different properties of **IntVar** which allow to inspect different aspects of the variables such as domain, number of constraints, etc...

3.3.2 Constraint Logic Programming over Rationals and Reals

The constraint is used both for variable creation using **realVar(name,lowerBound,upperBound,precision)** and for imposing different constraints which are encoded using continuous arothmetic and relational expressions (**CArExpression** and **CReExpression**) in a similar way discussed for finite domain variables.

Satisfaction and optimization problem are implemented as discussed for finite domain variables using only the default configuration provided by SWI Prolog because differently from Choco, SWI Prolog does not allow to customize the searching strategies.

Mixed Integer Programming predicates have been mapped using some additional variables to constraint real variables to assume integer values.

entailed/1 predicate has been implemented exploiting on **isSatisfied** property of

the **Constraint** interface.
dump/3 has been implemented using a **DefaultTermVisitor** which allows to replace variables in different constraints.

3.3.3 Constraint Logic Programming over Boolean Variables

The common mapping strategy behind all clpb predicates is the following:

- add new variables to the model
- store constraint to the model
- check whether the particular condition (e.g satisfiability or tautology) is true on the current model

3.4 Case study

The aforementioned **clpfd** library has been used to realize a real case study called "SchoolTimetable". This case study involves both Multi-Agent systems and Constraint Programming technologies.

Multi-Agent Systems (MAS) is an extension of the agent technology where a group of loosely connected autonomous agents act in an environment to achieve a common goal. This is done either by cooperating or competing, sharing or not sharing knowledge with each other [**inbook**]. Agent-oriented programming is a paradigm which is based on the concept of software agents. Several definitions have been provided, one of the most used is described in [**10.5555/1695886**] and claims that an agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives.

The problem consists of finding a school timetable for each professor of a school such that different constraints related to different aspects of the problem are respected, and trying to satisfy all preferences of each professor as much as possible.

Italian schools have a person (usually a professor) who is responsible for the creation of an overall timetable where for each day of the week and for each hour, a professor must be assigned to a class according to the hours that must spend in each class as described by school regulation. This task is difficult and time-demanding because it is subjected to different constraints related to professors and classes where they

teach. Sometimes it can also happen that a professor asks for a lesson change due to own commitments or needs and this entails finding a lesson change which keeps still valid the aforementioned constraints.
More in details the requirements are the followings:

- There is a person, responsible for the creation of the timetable of each professor (time-scheduler), who receives from school direction all information to perform the aforementioned task
- After the creation of timetables, the time-scheduler must send to each professor the corresponding timetable
- Each professor must spend exactly in each class a number of hours described by the school regulation according to the number and type of class where the teaching happens
- Each professor must have a free day during the school week
- In the same class cannot be more than one professor per hour (this requirement has been imposed to simplify particular cases, e.g. laboratories, where two professor teach in the same class)
- Each professor can have own preferences related to the lessons in which he/she would rather not teach
- Each professor can begin a negotiation trying to satisfy own preferences, mediated by the time-scheduler

3.4.1 Implementation

The system has been implemented using a MAS framework called Jade [**10.1007/3-540-44631-1_7**]. Jade (Java Agent DEvelopment framework) is a MAS framework developed in Java which supports the notion of agent. It has been used to simulate the time-scheduler and the professors as agents which interact in the same environment.
To implement the Constraint Programming new features have been added to **clpfd** library which are not supported in SWI Prolog:

- **all_distinct_except_0**: a predicate which states that all variables must be different except for 0 which can occur with repetitions
- **tuples_in** with reification: reification operators supported only relational predicates or reified predicate themselves. Now this predicate is also suported

3.4.2 Design

Several design choices have been taken and will be explained according to the kind of paradigm they affect

3.4.2.1 Multi-Agent Systems (MAS)

There are two types of agents: **TimeScheduler** and **Professor**.
TimeScheduler agent has in its own state all information related to the amount of hours each professor must spend in each class and knows the Agent Identifier (AID) of each **Professor** agent. Moreover, it uses the Directory Facilitator (DF) agent to register its own service of negotiation mediator.
The **TimesScheduler** is responsible for the following tasks:

- creation of timetables
- communication of timetables to each professor
- mediation of negotiation among professors for the satisfaction of their preferences

These tasks are accomplished with the following Behaviors:

- **TimetableBehaviour**: it produces the different timetables, save and send each of them to the corresponding professor
- **WaitProposalBehavior**: it waits for a cfp message from a professor to begin a negotiation
- **MediationBehaviour**: it mediates negotiation among professors for preferences satisfaction

Professor agent is an abstract class which is used to keep a common architecture for all professors which differ only for their preferences.
Professor agents perform the following tasks:

- receive the timetable
- try to satisfy own preferences
- decide whether to accept a lesson change or not

In order to do these tasks they have the following Behaviours:

- **TimetableBehaviour**: it receives the timetable and save it

- **PreferenceBehaviour**: for a specific number of times it tries to satisfy unsatisfied preferences
- **NegotiationBehaviour**: it starts a negotiation for a specific unsatisfied preference
- **WaitProposalBehavior**: it waits for a request message from the timescheduler corresponding to a change proposal and manages it with a CandidateBehaviour
- **CandidateBehaviour**: it replies to the proposal of a specific lesson change

Professor agents, after receiving the timetable, register the classes where they teach as services interacting with the DF agent. This is done because when the **Timescheduler** agent will receive a lesson change request, it will look for all professors who teach in the same class where the proposed change happens.
Another important design choice is related to the following aspect: when a professor asks for a lesson change, it must be sure that before the end of the negotiation, the current lesson cannot be available to other incoming requests. To do this, in the **NegotiationBehaviour** the current lesson is added to a set of locked lessons. In this way if another professors ask him/her to change this lesson, the request will not be able to be satisfied.

3.4.2.2 Communication among agents

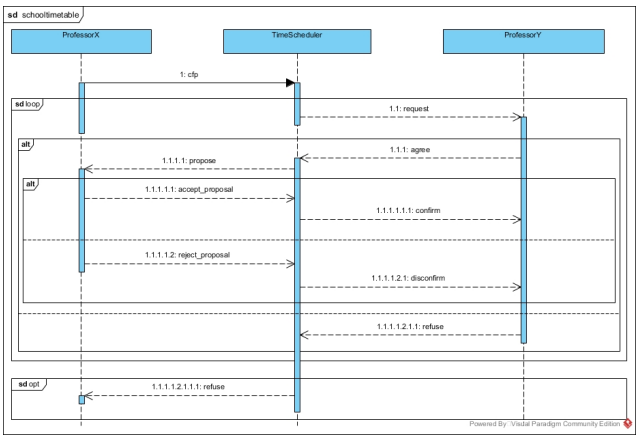

Figure 3.2: Sequence Diagram of the interaction among agents

The interaction among time-scheduler and professors can be explained as follows (as shown in figure **??**):

when a professor wants to change an own lesson in order to satisfy a preference, a call-for-proposal (cfp) message is sent to the time-scheduler to begin the negotiation. The time-scheduler will retrieve the list of all professors who teach in the same class of the proposer's lesson. To have a reasonable change it is also important to consider that for each candidate professor and for each candidate lesson:

- the proposer must be free during a candidate lesson
- the candidate professor must be free during the lesson the time-scheduler wants to propose.
- the proposer's lesson cannot be in the free day of the candidate professor and the candidate lesson cannot be in the free day of the proposer professor

Following these criteria, the time-scheduler will realize a list of pair professor-lesson and will try to satisfy the proposer's request with one of them. The time-scheduler sends to the candidate professor a proposal and this proposal can be accepted according to two criteria:

- the proposed lesson is not one of own preferences
- the lesson to give must not be in the locked preferences

If these criteria are met, the candidate professor replies with an agree message otherwise refuses. If the reply message contains agree as performative, the time-scheduler communicates to the proposer the possible change. The proposer will accept or reject according to the fact that the lesson change does not worsen own preferences. According to the last message received from the proposer, the time-scheduler will send to the candidate, who previously accepted the change, a confirm or disconfirm message. If all possible negotiations fail, the time-scheduler will send to the proposer a refuse message.

Ontologies

As described in [**10.1007/3-540-44631-1_7**] the content of a message is either a string or a raw sequence of bytes. In realistic case, like in this one, agents need to communicate complex information. When representing complex information, it is necessary to adopt a well-defined syntax so that the content of a message can be parsed by the receiver to extract each specific piece of information. According to FIPA terminology this syntax is known as a content language. FIPA does not

mandate a specific content language but defines and recommends the SL language to be used when communicating with the AMS and DF.
For this reason the SL language is used as content language for all messages involving the time-scheduler and the professors. The ontology shared by all agents consists of four concepts and three predicates. The different elements are described with SL syntax as follows:

Concepts

$$(Lesson : hour\langle hour\rangle : day\langle day\rangle)$$

$$(SchoolClass : year\langle year\rangle : letter\langle letter\rangle)$$

$$(Teaching : lesson\langle hour\rangle : schoolClass\langle class\rangle)$$

$$(TimetableConcept : teachings\langle list\ of\ teachings\rangle)$$

Predicates

$$(UpdateTimetable : timetable\langle timetable\ of\ an\ agent\rangle)$$

$$(Change : lessonChange\langle lesson\ related\ to\ a\ change\rangle)$$

$$(Substitution : proposedLesson\langle proposer's lesson\rangle : currentLesson\langle candidate's lesson\rangle)$$

TimetableConcept is used to encode as content message the timetable of each professor stored as an object of the class **Timetable**. The reason behind two different classes is due to the fact that **Timetable** uses a matrix to store information about lessons and it is faster to retrieve and add elements compared to the class **TimetableConcept**. On the other hand SL language does not support the matrix concept and this is why a concept modelled as a list of teachings is needed.

3.4.2.3 Constraint Programming (CP)

In the following part it is provided a mathematical formulation of the model and the corresponding code in Prolog. The model should be generated dynamically according to the single instance data but to speed up the development process, the current version of this project considers a specific instance of the problem.

3.4.2.4 Data provided by the problem

Days of the school week

$$numDays \in \{1..6\}$$

Hours for each school day

$$numHours \in \{1..24\}$$

Number of different professors

$$numProfessors \in \mathbb{N}^+$$

Number of different classes

$$numClasses \in \mathbb{N}^+$$

Function which assigns for each professor and each class the number of hours:

$$\forall i \in \{1..numProfessors\}$$

$$hours_i : \{1..numClasses\} \rightarrow \{0..(numHours * numDays)\}$$

Domain of variables

$$\forall i \in \{1..numProfessors\}, j \in \{1..numHours\}, k \in \{1..numDays\}$$

PROLOG

$$P_{ijk}\ in\ 0..numClasses$$

NOTE: 0 is used as joker value to state no class

3.4.2.5 Constraints

Each Professor must teach for a specified number of hours in each class:

$$\forall i \in \{1..numProfessors\}$$

PROLOG

$$global\_cardinality([P_{i**}], [1{-}hours_i(1), 2{-}hours_i(2), .., numClass{-}hours_i(numClass)])$$

NOTE: the number of same values assigned to professor's variables must be equals to the number of hours the professor must spend in that class.

$[P_{i**}]$ means the list of variables associated to professor i

In the same day and hour cannot be more than one professor in each class

$$\forall j \in \{1..numHours\}, k \in \{1..numDays\}$$

PROLOG

$$all\_distinct\_except\_0([P_{*jk}])$$

NOTE: $[P_{*jk}]$ is the list of variables associated to all professors having the same day and hour.

Each Professor must have a free day

$$\forall i \in \{1..numProfessors\}$$

PROLOG

$$tuples\_in([P_{i*1}],[[0..0]]) \vee tuples\_in([P_{i*2}],[[0..0]]) \vee ..tuples\_in([P_{i*(numDays)}],[[0..0]])$$

NOTE: $[P_{i*1}]$ is the list of all variables of professor i at day 1 for all hours.

3.4.2.6 Performance

The solution of the CP model requires a lot of time to be computed because of the huge amount of variables and constraints imposed on them. Therefore a **DummyBehaviour** has been realized to simulate the creation of the professors' timetable and then each timetable is sent to the corresponding professor.

Chapter 4

Labelled Prolog

Nowadays *pervasive systems* are a challenge which requires suitable models and technologies to support *distributed situated intelligence*. Logic Programming (LP) can be used as the core of such models and technologies thanks to its declarative interpretation and inferential capabilities but, it is unsuitable to capture different domain-specific computational models. For this reason, logic programming needs to be extended delegating other aspects, such as situated computations, to other languages, or, to other levels of computation [**10.3233/FI-2018-1695**].
In the following chapter two different approaches will be presented: *Labelled Variables in Logic Programming* (LVLP) and *Labelled Terms in Logic Programming* (LTLP).

Labelled Variables in Logic Programming consists of enabling different computational models adding information (called *labels*) to variables whereas *Labelled Terms in Logic Programming* extends the label concept to all possible kind of terms in a logic program.

Both LP extensions have been implemented in Prolog language. This first one has been described in [**10.3233/FI-2018-1695**] whereas the second one will be explained more in details in further section **??**.

4.1 Model

4.1.1 Labelled Variables

Let C be the set of constants, with c1, c2 $\in C$ being two generic constants. Let V be the set of variables, with v1, v2 $\in V$ being two generic variables. Let F be the set of function symbols, with f1, f2 $\in F$ being two generic function symbols; each f

$\in$ F is associated to an arity ar(f) > 0, stating the number of function arguments. Let T be the set of terms, with t1, t2 $\in T$ being two generic terms. Terms can be either simple, a constant (e.g., c1) and a variable (e.g., v2) are both simple terms, or compound, a functor of arity n applied to n terms (e.g., f1(c2, v1)) is a compound term. A term is said ground if it does not contain variables Let H denote the set of ground terms, also known as the Herbrand universe. A model for Labelled Variables in Logic Programming (LVLP) is defined as a triple $\langle B, f_L, f_C \rangle$, where

- $B = \{\beta_1, ..., \beta_n\}$ is the set of basic labels, the basic entities of the domain of labels
- $L \subseteq \mathcal{P}(B)$ is the set of labels, where each label $l \in L$ is a subset of B
- $f_L : L \times L \to L$ is the (label-)combining function computing a new label from two given ones
- $f_C : H \times L \to true, false$ is the compatibility function, assessing the compatibility between a ground term and a label when interpreted in the domain of labels
- a labelled variable is a pair $\langle$v, l$\rangle$ associating label l $\in$ L to variable v $\in$ V
- a *labelling* is a set of *labelled variables*

An LVLP program is a collection of LVLP rules of the form

$$Head :- Labelling, Body.$$

to be read as "Head if Body given Labelling". There, Head is an atomic formula, Labelling is the list of labelled variables in the clause, and Body is a list of atomic formulas. An atomic formula has the form $p(t_1, ..., t_n)$ where p is a predicate symbol and t_i are terms. Atom $p(t_1, ..., t_n)$ is said ground if $t_1, ..., t_n$ are ground.

The *unification* between terms happens as described in the following table:

	constant c_2	*variable* v_2	*labelled variable* $\langle v_2, \ell_2 \rangle$	*compound term* s_2
constant c_1	if $c_1 = c_2$ then true else false	true, $\{v_2/c_1\}$	if $f_C(c_1, \ell_2) =$ true then true, $\{v_2/c_1\}$, ℓ_2 else false	false
variable v_1	true, $\{v_1/c_2\}$	true, $\{v_1/v_2\}$	true, $\{v_1/v_2\}, \ell_2$	if v_1 does not occur in s_2 then true, $\{v_1/s_2\}$ else false
labelled variable $\langle v_1, \ell_1 \rangle$	if $f_C(c_2, \ell_1) =$ true then true, $\{v_1/c_2\}$, ℓ_1 else false	true, $\{v_1/v_2\}, \ell_1$	if $f_L(\ell_1, \ell_2) \neq \emptyset$ then true, $\{v_1/v_2\}$, $f_L(\ell_1, \ell_2)$ else false	if v_1 does not occur in s_2, and $f_L(\ell_1, \ell'_1, ..., \ell'_n) \neq \emptyset$ where $\ell'_1, ...\ell'_n$ are the labels in s_2 then true, $\{v_1/s_2\}$, $f_L(\ell_1, \ell'_1, ..., \ell'_n)$ else false
compound term s_1	false	if v_2 does not occur in s_1 then true, $\{v_2/s_1\}$ else false	if v_2 does not occur in s_1, and $f_L(\ell_2, \ell'_1, ..., , \ell'_n) \neq \emptyset$ where $\ell'_1, ...\ell'_n$ are the labels in s_1 then true, $\{v_2/s_1\}, f_L(\ell_2, \ell'_1, ..., , \ell'_n)$, else false	if s_1, s_2 have same functor / arity, and their arguments (recursively) unify then true else false

Table 4.1: Unification rules in LVLP, adopting standard LP unification rules and representation

The only case to be added to the standard unification table is represented by labelled variables. There, given two generic LVLP terms, the unification result is represented by the extended tuple

$$(true/false, \theta, l)$$

where true/false represents the existence of an answer, θ is the most general unifier mgu, and l is the new label associated to the unified variables defined by the (label-)combining function f_L It is important to define a concept which allows to check whether a label assigned to a variable is compatible with its substitution or not. To do this, the following *compatibility function* f_C is defined:

$$f_C : H \times L \rightarrow \{true, false\}$$

In particular, given a a ground term t $\in$ H and label l $\in$ L:

$$f_C(t,l) = \begin{cases} true & \text{if there exists at least one element of the domain of labels which the interpretatio} \\ false & \text{otherwise} \end{cases} \tag{4.1}$$

4.1.2 Labelled Terms

Labelled Terms can be seen as a generalization of the previous approach where it is possible to apply a label not only to a variable but to each kind of term. The main idea of this approach consists of using two functions:

- **shouldUnify(term1, labels1, term2, labels2)**: this function is used to check whether, according to *labels1* and *labels2*, it is possible to unify *term1* and *term2*. This function is very similar to f_L used in the unification process to allow the substitution of a variable when $f_L(labels1, labels2) \neq \emptyset$
- **merge(term1, labels1, term2, labels2)**: this function is used to generate new labels after unifying terms *term1* and *term2*. It has the same meaning of $f_L(labels1, labels2)$ but it is applied to every kind of term

Unification between terms happens as described by the following table:

	constant c_2	*labelled constant* $\langle c_2, \ell_2 \rangle$	*variable* v_2	*labelled variable* $\langle v_2, \ell_2 \rangle$	*co*
constant c_1	if $c_1 = c_2 \wedge$ shouldUnify(c_1,$\emptyset$,c_2,$\emptyset$) then **true**, merge(c_1,$\emptyset$,c_2,$\emptyset$) else **false**	if $c_1 = c_2 \wedge$ shouldUnify(c_1,$\emptyset$,c_2,ℓ_2) then **true**, merge(c_1,$\emptyset$,c_2,ℓ_2) else **false**	if shouldUnify(c_1,$\emptyset$,v_2,$\emptyset$) then v_2/c_1, merge(c_1,$\emptyset$,v_2,$\emptyset$) else **false**	if shouldUnify(c_1,$\emptyset$,v_2,ℓ_2) then v_2/c_1, merge(c_1,$\emptyset$,v_2,ℓ_2) else **false**	
labelled constant $\langle c_1, \ell_1 \rangle$		if $c_1 = c_2 \wedge$ shouldUnify(c_1,ℓ_1,c_2,ℓ_2) then **true**, merge(c_1,ℓ_1,c_2,ℓ_2) else **false**	if shouldUnify(c_1,ℓ_1,v_2,$\emptyset$) then v_2/c_1, merge(c_1,ℓ_1,v_2,$\emptyset$) else **false**	if shouldUnify(c_1,ℓ_1,v_2,ℓ_2) then v_2/c_1, merge(c_1,ℓ_1,v_2,ℓ_2) else **false**	
variable v_1			if shouldUnify(v_1,$\emptyset$,v_2,$\emptyset$) then v_1/v_2, merge(v_1,$\emptyset$,v_2,$\emptyset$) else **false**	if shouldUnify(v_1,$\emptyset$,v_2,ℓ_2) then v_1/v_2, merge(v_1,$\emptyset$,v_2,ℓ_2) else **false**	if v $\wedge$ s then
labelled variable $\langle v_1, \ell_1 \rangle$				if shouldUnify(v_1,ℓ_1,v_2,ℓ_2) then v_1/v_2, merge(v_1,ℓ_1,v_2,ℓ_2) else **false**	if v $\wedge$ sh then
compound term s_1					if s_1, s_2 ha sho arguments recu
labelled compound term $\langle s_1, \ell_1 \rangle$					

Table 4.2: Unification rules in LTLP, adopting standard LP unification rules and representation

A specific issue of this generalization of labels to all kind of terms consists of considering the compatibility between the labels of a predicate and the labels of its arguments.
This is reasonable because a predicate expresses a relationship among objects; therefore the information (labels) associated to these objects must be meaningful with respect to the information associated to their relation.
For this reason the function **stillValid(struct)** has been added to the resolution process to check the compatibility of the aforementioned labels.

4.2 Implementation

Implementation has been realized exploiting on 2P-Kt framework (previously described in **??**). Different choices have been made trying to extend as much as possible existing components provided by this framework. Choices will be described according to the aspects they affect.

4.2.1 Labels

Labels have been attached to each term using a map from *String* to *Any* called tags. *Tags* are a machanism which can be realized by all classes which implements **Taggable** interface. **Term** implements this interface therefore this map is used to associate to a tag (corresponding to labels concept) to a set of Label where a Label can be seen as a generic class which implements the **Label** interface. Some examples have been provided where *labels* are represented as strings.
Adding *labels* to a term required also to change how the new term is formatted. For this reason **LabelAwareTermFormatter** has been created to properly format labelled terms. Each label is written preceeded by '@' symbol and all labels of a term are surrounded bu angular brackets.
For example the struct **f(a,B)** having **a** associated with labels x,y is formatted as:

$$f(a < @x, @y >, B)$$

4.2.2 Unificator

Unificator interface has been extended to support the functions described in **??**. **AbstractUnificator** class has been modified in order to allow the customization of the final substitution. In the current scenario the substitution contains also all labels associated to terms that are compared during the generation of the most general

unifier (m.g.u.).
The class **AbstractLabelledUnificator** has been created to allow the user to define own callback functions (shouldUnify and merge) in order to deal with the specific scenario which requires specific kinds of labels and unification rules.

4.2.3 Solver

LabelledPrologSolver is a paticular Solver and deals with labels using a Unificator which implements the interface **LabelledUnificator**. The consistency among the labels of a predicate and the labels of its arguments is checked using **hijackStateTransition** which is a method used to intercept the transition between states in the Prolog Finite State Machine (FSM). In this case each time a Rule or a Primitive has been executed and the next state is the Goal Selection, **stillValid** method is called to check whether labels are still valid or not and in the latter case the next state is the Backtracking state whose main purpose is to choose a different rules or primitive to execute.
stillValid is a callback which is defined by the user according to the specific scenario.

Chapter 5

CLP as Labelled Prolog

Labelled Prolog is a monotonic extension of Prolog which allows to deal with specific domains without the need to use specific languages or libraries to deal with them.
There are different variants and extensions of logic programming which include:

- abductive logic programming [**10.1093/logcom/2.6.719**]
- metalogic programming [**10.5555/94469**]
- constraint logic programming [**JAFFAR1994503**]
- concurrent logic programming [**10.1145/800223.806776**]
- probabilistic logic programming [**NG1992150**]

With **Labelled Prolog** discussed in the previous chapter, all these extensions and variants could be framed in a single environment allowing to reduce the amount of languages to a single one (Labelled Prolog) and focusing more on how to adapt labels and unification mechanisms to the specific domain needs.

In the current chapter is shown how to adapt CLP to the labels mechanism using the following example.
Let's assume to have the following *Knowledge Base*:

```
problem(X<@1,@100>,Y<@1,@100>)<@>> :- [X,Y] ins 1..100, X #> Y.
problem(X<@1,@100>,Y<@1,@100>)<@<> :- [X,Y] ins 1..100, X #< Y.
problem(X<@1,@100>,Y<@1,@100>)<@>=> :- [X,Y] ins 1..100, X #>= Y.
```

A possible *Goal* could be:

```
?-problem(X<@2,@8>,Y<@7,@12>)<@>>
```

With the current example the idea would be to state that we want to solve a problem whose variables have a domain which is described by their labels and whose constraints are described by the labels of predicate *problem/2*.
In this way we are able to express a CLP problem in a much more expressive way wrt a way which is strictly related to a specific language or library. The unification process will check, according to the constraint described as label, which is the rule to execute. Domain of variables are combined using intersection and in this case the *stillValid* callback checks, in addition to the equality of the constraint, whether assignments of variables are between the first and the second label of the variables (they correspond to the lower and upper bounds).
In this example the body of the rules contains a specific domain range because with the current implementation it is not possible to use labels as arguments of a predicate but only to attach them to terms. Therefore, a default domain range is provided for both variables.

Chapter 6

Conclusions and future work

Acknowledgements

First, I would like to express my deepest gratitude to Professor Calegari and Professor Ciatto for all the support they provided me during the internship and thesis redaction processes.

Second, I would like to thank my family, my friends, my former classmates and all people who believed in me during this long study path.

Last but not least, I would like to thank myself to have been able to never give up during these two years.

Bologna, 03 February 2023 Giuseppe Boezio