

Project: Solving proximity constraints

Jan-Michael Holzinger^{*}

Sophie Hofmanninger[†]

JKU Linz — SS2019

Version Number	Changes Summary	Author
0.1		Jan-Michael
0.2	added System Model	Jan-Michael
0.3	modified Parser, added Workflow	Jan-Michael
0.4	add ConstraintSimplification	Sophie
0.5	Proximity Relation (as Matrix) added	Jan-Michael
0.6	added UML	Jan-Michael

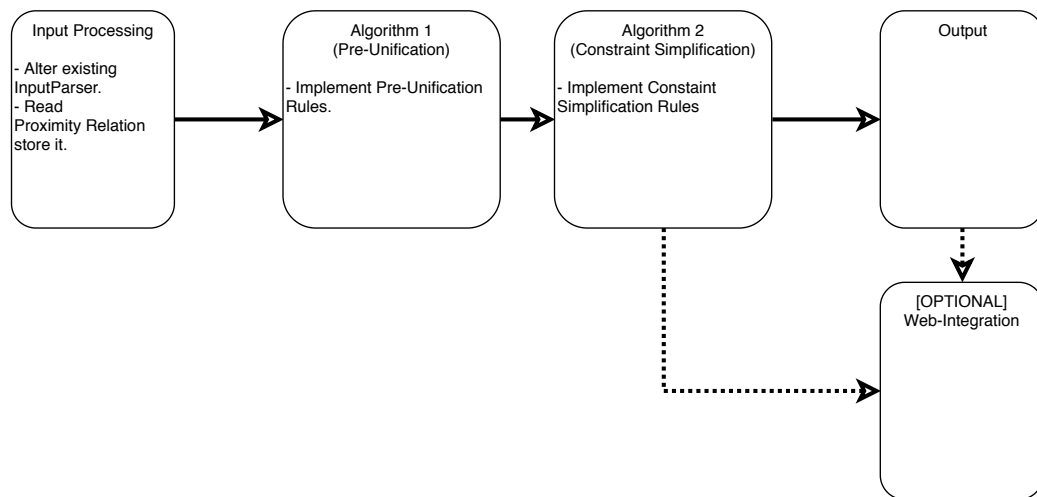
1 System Overview

We split the problem in 4 (5) smaller tasks:

1. Input Processing,
 2. Pre-Unification,
 3. Constraint Simplification,
 4. Output.
- O. Web-Integration.

^{*}jan.holzinger@gmx.at

[†]sophie@hofmanninger.co.at



1.1 Input Processing

The first idea here is to copy, alter and extend the existing code, in the class InputParser.

For the Proximity Relations \mathcal{R} and the λ -cut we have the following idea:

1. We try to get the number of function symbols (n), constants are treated as 0-ary functions.
2. We let the user input the values to construct a symmetric matrix that consists of values in $[0, 1]$. This matrix must have a 1 in the main diagonal. All values below are 0. Therefore they will not be stored in the implementation.
3. We let the user (later) input $\lambda \in [0, 1]$ and calculate the set \mathcal{R}_λ .

The current implementation of InputParser creates a list of all Functions. This list is then sorted by arity, i.e. let f, g be functions with arity $a, b \in \mathbb{N}$ respectively. Then $a < b \Rightarrow f \prec g$.

Let now $n \in \mathbb{N}$ be the size of the list, the list then represents the caption of a $n \times n$ matrix. Assume the problem contains functions with arity $0, 1, \dots, m$. Let k_l be the number of Functions with arity l . Then

$$\sum_{l=0}^m k_l = n.$$

The matrix then looks like

$$\begin{matrix} & f_{0_1} & f_{0_2} & \dots & f_{0_{k_0}} & f_{1_1} & f_{1_2} & \dots & f_{1_{k_1}} & \dots & f_{m_1} & f_{m_2} & \dots & f_{m_{k_m}} \\ \begin{matrix} f_{0_1} \\ f_{0_2} \\ \vdots \\ f_{0_{k_0}} \\ f_{1_1} \\ f_{1_2} \\ \vdots \\ f_{1_{k_1}} \\ \vdots \\ f_{m_1} \\ f_{m_2} \\ \vdots \\ f_{m_{k_m}} \end{matrix} & \left(\begin{array}{cccccccccccc} 1 & (*) & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & 1 & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & \ddots & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & (*) & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & 1 & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & \ddots & (*) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & (*) & (*) & (*) & (*) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & 1 & (*) & (*) & (*) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & \ddots & (*) & (*) \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & * & 1 & 1 \end{array} \right), \end{matrix}$$

where:

- the 1's in the diagonal are fixed, as a function must have proximity 1 to itself,
- the 0's are fixed, as functions with different arities can't be close,
- the values where the entry is * must be of the same value as their (*) counterpart, as the matrix is symmetric.

Hence, we only need to get the values for the positions marked with (*). We call such a position “open case”, and we make a list of open cases. We will then wait for user input.

If we want to retrieve a value, i.e. if we want to know $\mathcal{R}(s_1, s_2)$ for given s_1, s_2 , we can set

$$i := \text{sortedListOfFunctions.getIndex}(s_1) \quad \text{and} \quad j := \text{sortedListOfFunctions.getIndex}(s_2),$$

and then just get the value at position (i, j) .

1.2 Algorithms

We implement the Algorithms in an own class, that has two static functions, `preUnification` and `constraintSimplification`.

1.2.1 Pre-Unification Algorithm

The `preUnification` method consists of a loop, that runs until either $P = \emptyset$ or it is detected, that there is no solution to the problem.

Inside the loop body, the 7 pre-unification rules are iteratively applied to the first element (which gets popped by doing so). The method returns true, iff the pre-unification was successful.

The method changes the problems constraints and pre-unifier accordingly.

1.2.2 Constraint Simplification Algorithm

The `constraintSimplification` method will be called, if the `preUnification` method returns true. As the `preUnification` method, the `constraintSimplification` method consists of a loop. This loop runs until the set of constraints is empty. As input the method needs the problem, more precisely the set of constraints, and the relation matrix \mathcal{R} . If the set of constraints could be simplified, the method returns true and otherwise false.

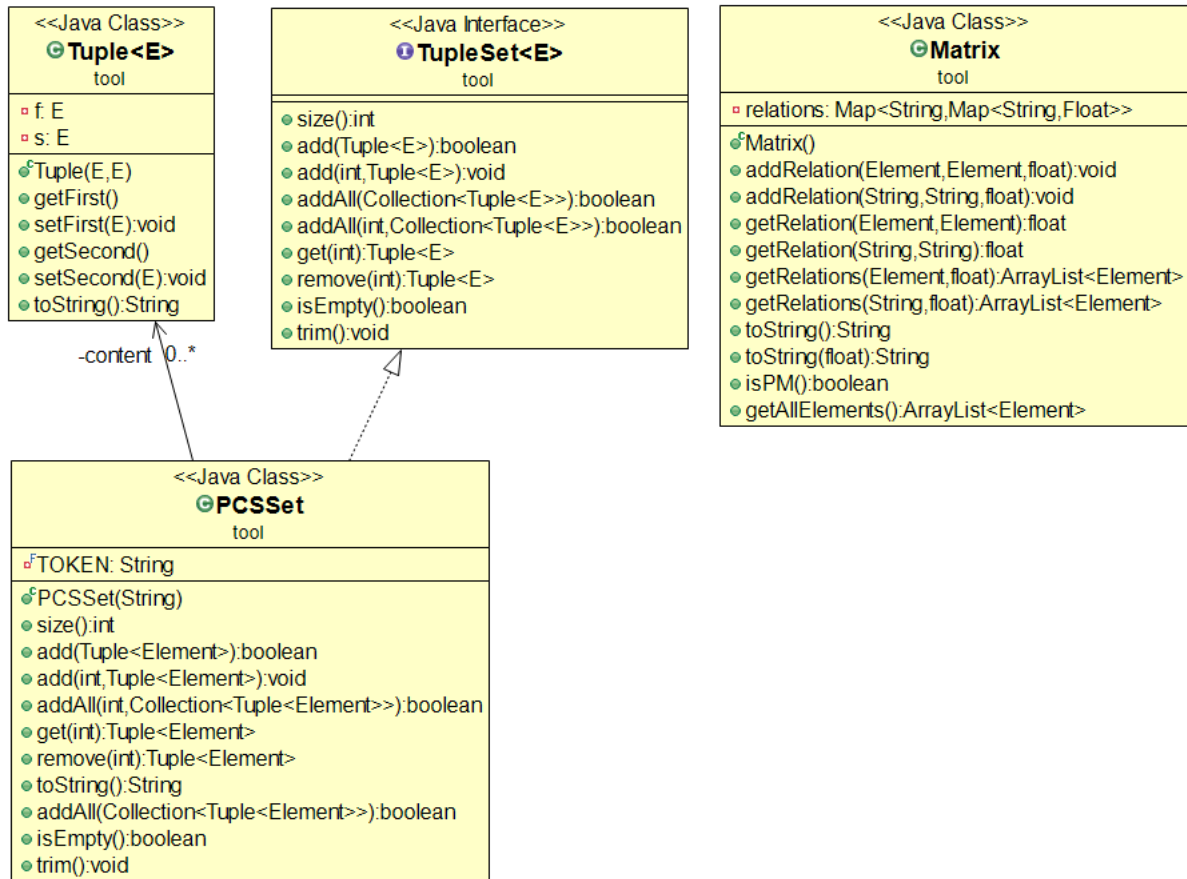
Inside the loop, the seven rules to simplify the constraint set, will be applied. Two of the seven rules are hidden, because we implement the FSN rule directly in the NFS rule and the NN2 rule is integrated in the NN1 rule.

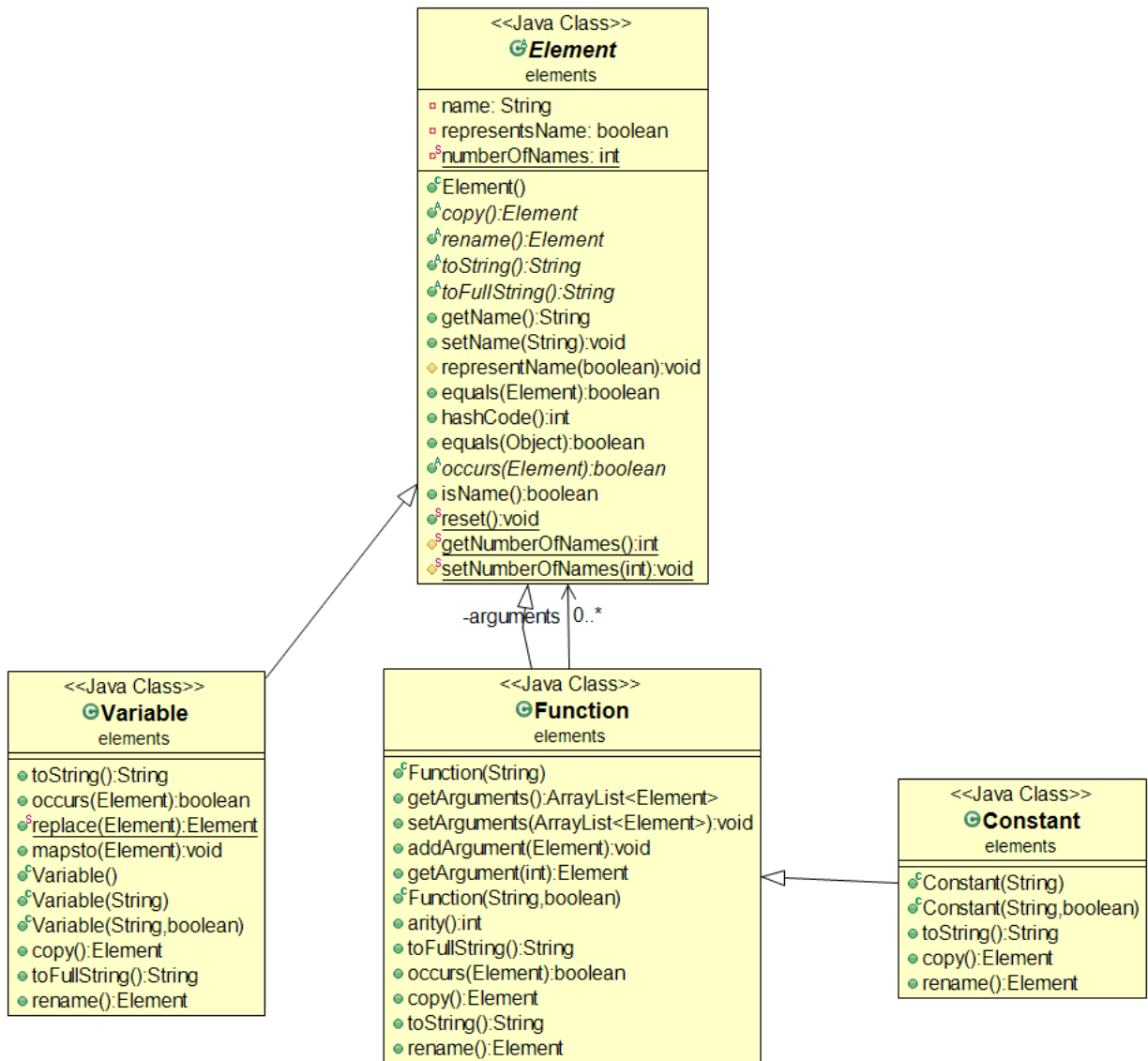
1.3 Output

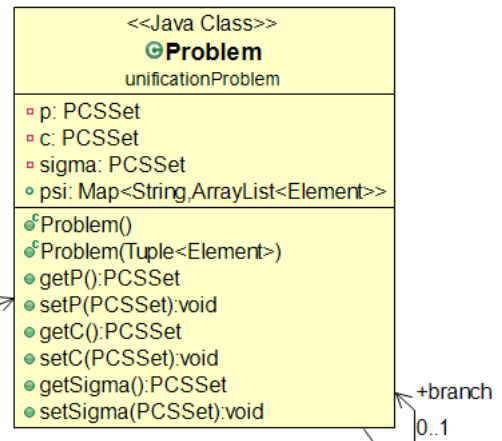
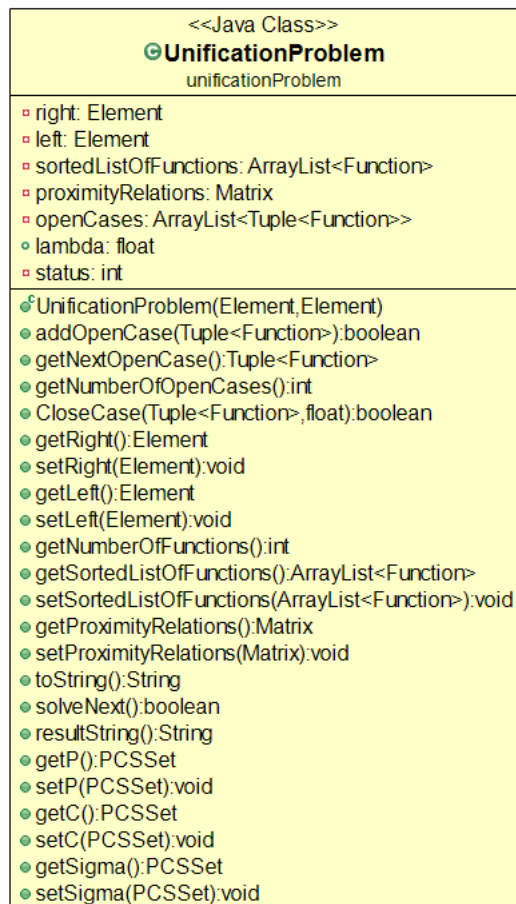
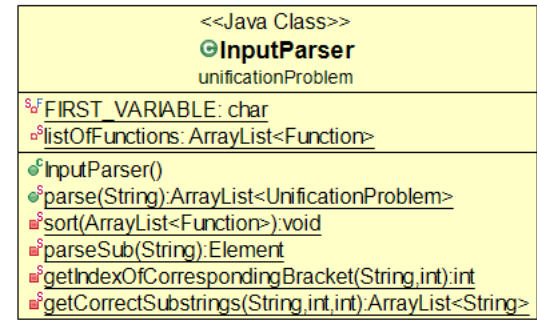
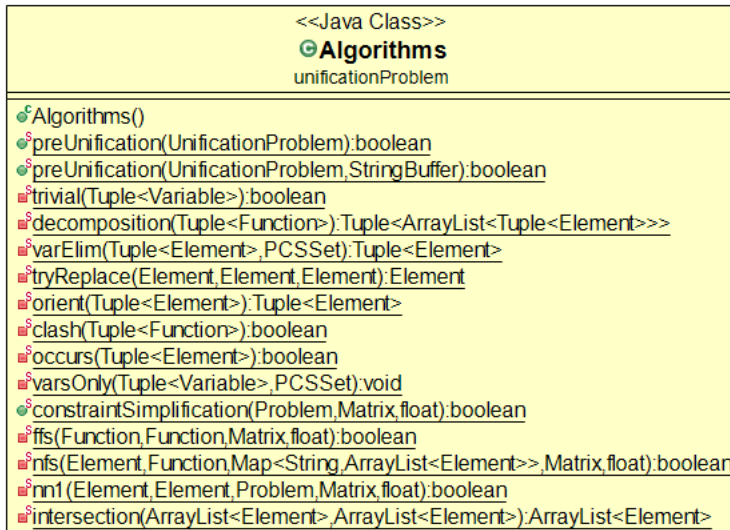
2 System Model

The program consists of 3 packages,

- tool
- elements
- unificationProblem







+prob
0..1

+branch
0..1

3 Work Flow

The typical workflow looks like this:

