

# Project: Solving proximity constraints

Jan-Michael Holzinger<sup>\*</sup>

Sophie Hofmanninger<sup>†</sup>

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	update system model, update matrix	Sophie
0.7	added UML	Jan-Michael
0.8	added User Interfaces	Jan-Michael
0.9	minor changes in matrix description update constraint simplification	Sophie
0.10	updated UML files	Jan-Michael
0.11	added Input Checker	Jan-Michael

## 1 System Overview

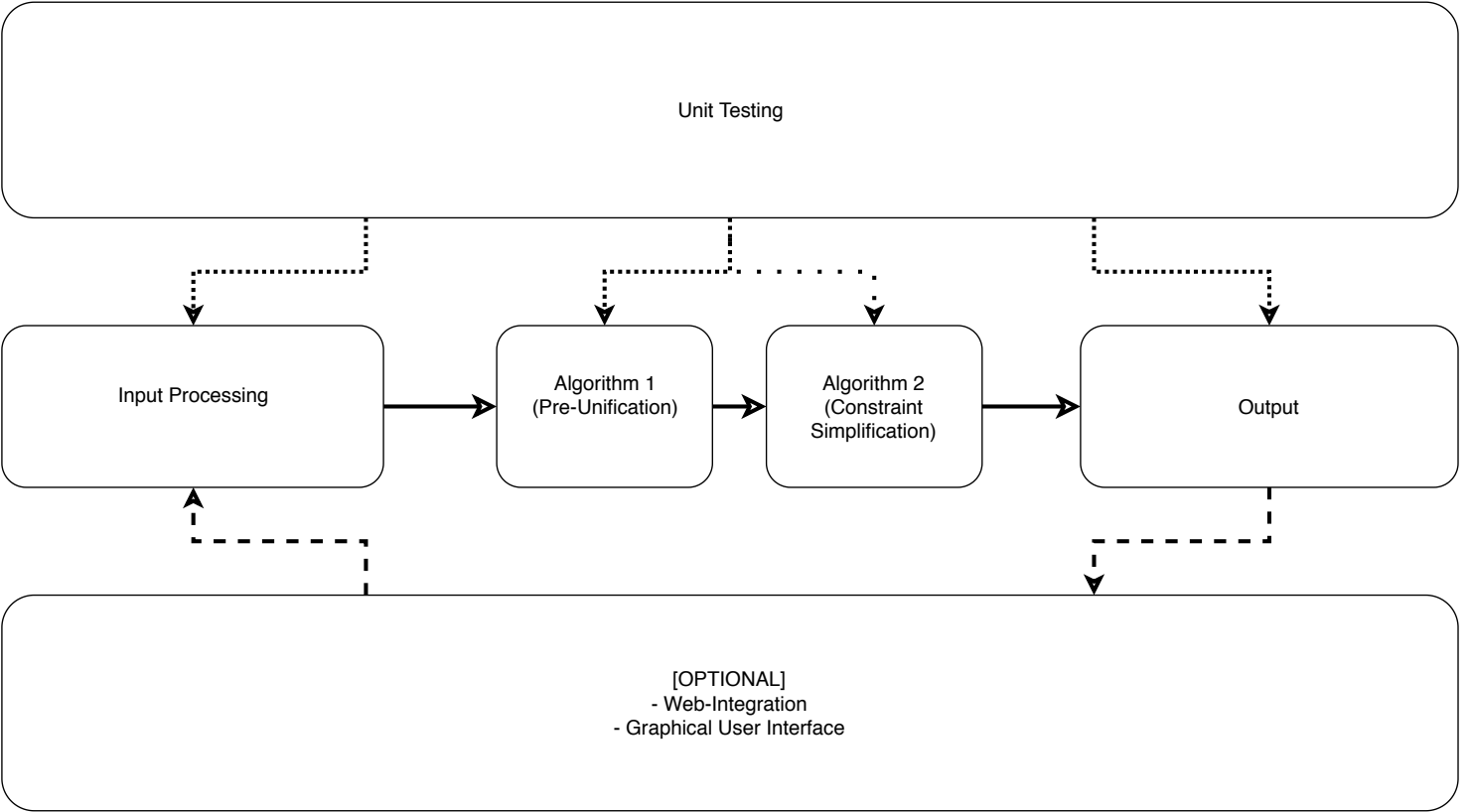
We split the problem in 4 (5) smaller tasks:

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

---

<sup>\*</sup>jan.holzinger@gmx.at

<sup>†</sup>sophie@hofmanninger.co.at



## 1.1 Input Processing

### 1.1.1 InputParser

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

### 1.1.2 Input Checker

We want to be able to check the input for example for missing parenthesis and so on. Therefore we will allow the user to specify an InputChecker. The idea is, to create an Interface, with just a method `check(String):boolean`. This then can be implemented by the implementation of the checkers.

### 1.1.3 Proximity Relations

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

1. We try to get the number of function symbols with the same arity.
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  $\mathcal{R}_\lambda$ .

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

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

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. Before asking the user to enter relation values, we will generate the list of open cases inside of the matrix. After generation of the list, we will wait for user input. With the valid user input we will build the proximity relation matrix.

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 call *getRelation*( $s_1, s_2$ ). To get all relation pairs of  $s_1$ , that have a relation value greater or equal to  $\lambda$ , the method *getRelations*( $s_1, \lambda$ ) can be used. Moreover it is possible to get all relation elements of the matrix by calling *getListOfFunctions*(). This will return all  $f_{i_k,j}$ , where  $0 \leq i \leq m$  and  $0 \leq j \leq m$ .

## 1.2 Algorithms

We implement the Algorithms in an own class, that has two public static functions, *preUnification* and *constraintSimplification*. The other methods are only used for the construction of the two simplification algorithms and therefore they are private and static. The methods should take two inputs, mandatory an unification problem, and optional a *StringBuffer*, to log certain events.

### 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.

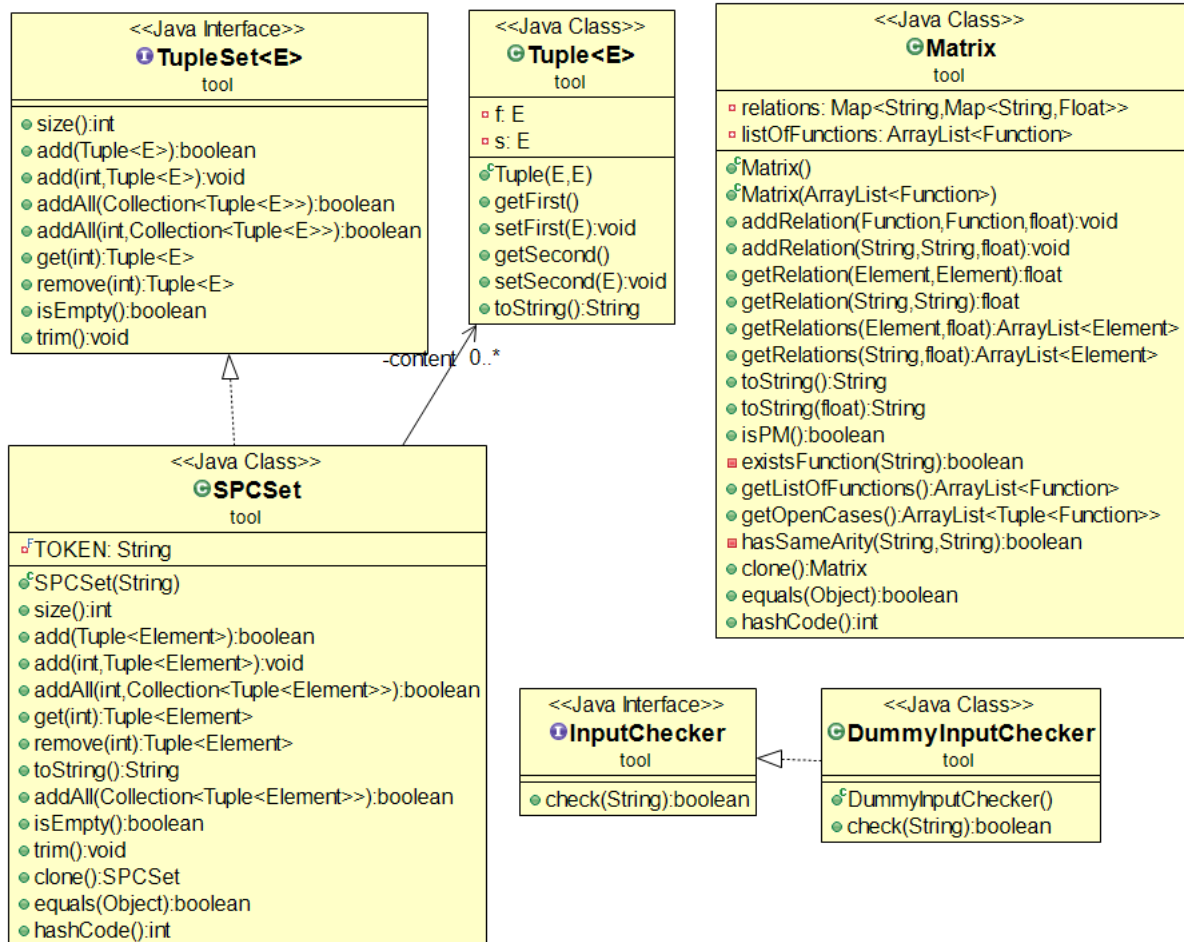
Inside the loop, the seven rules to simplify the constraint set, will be applied. Three of the seven rules are hidden, because the NN2 rule is integrated in the NN1 rule, the Fail1 is implemented in FFS and the Fail2 is integrated in the NN1 rule. Moreover it is possible to run the constraint simplification algorithm with and without logging.

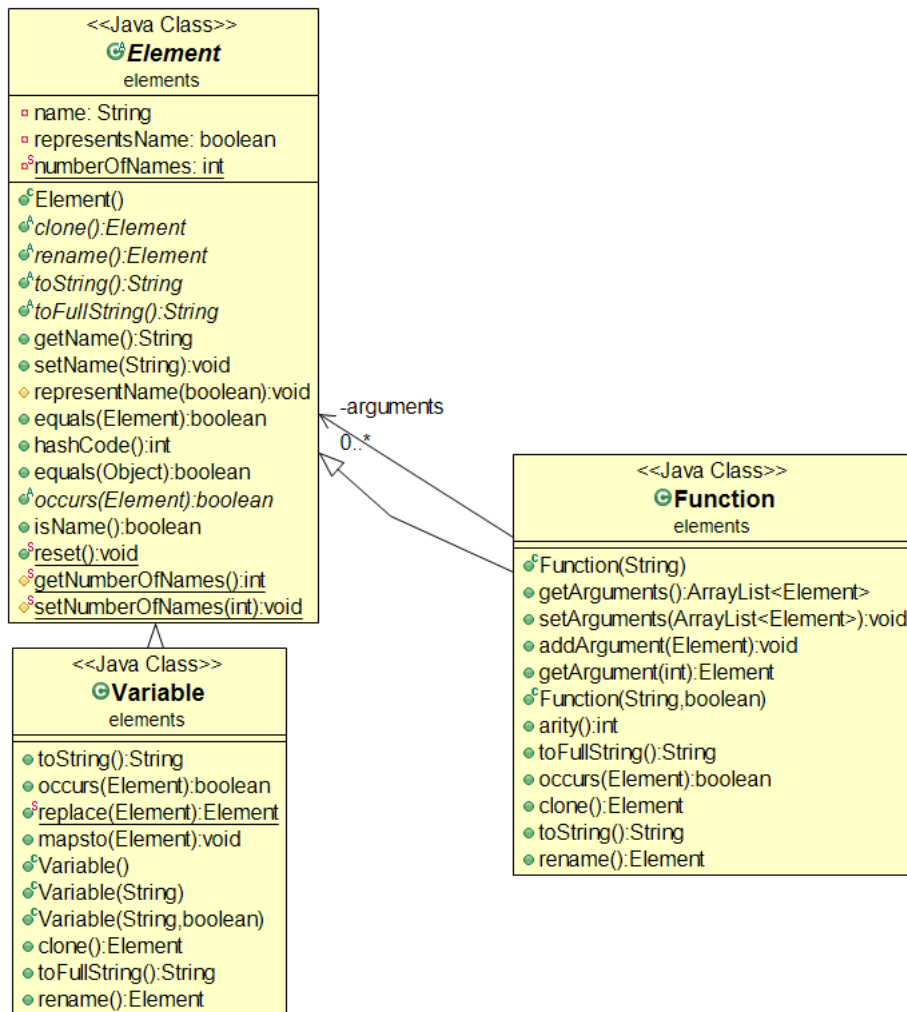
## 1.3 Output

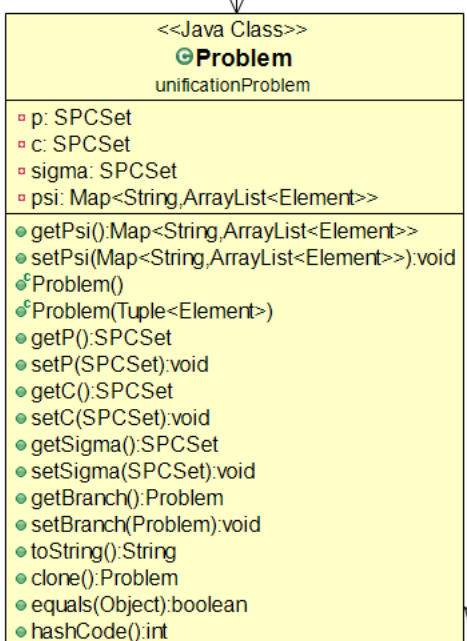
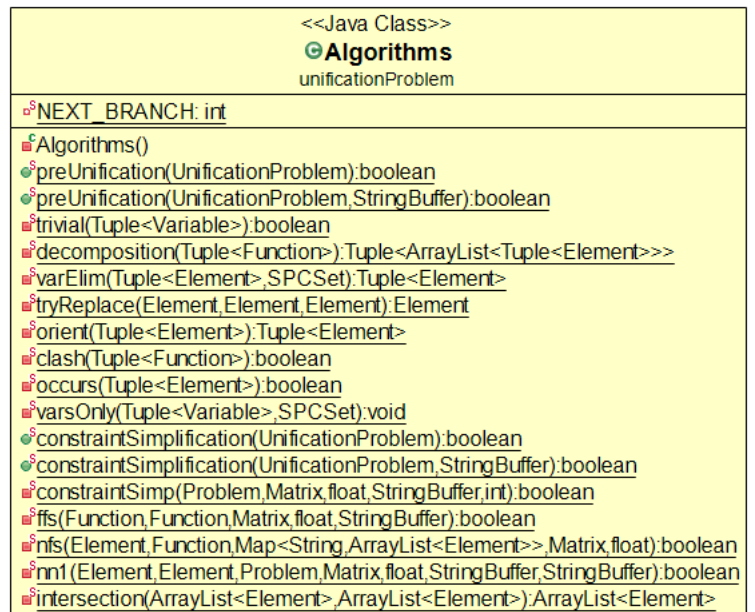
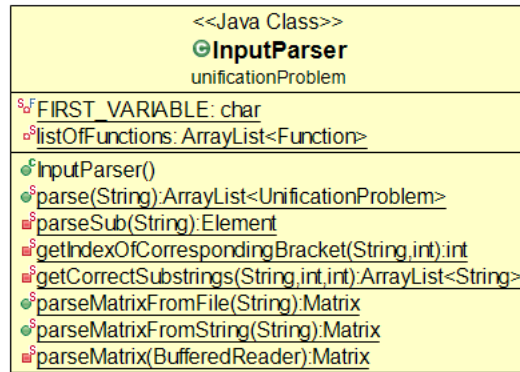
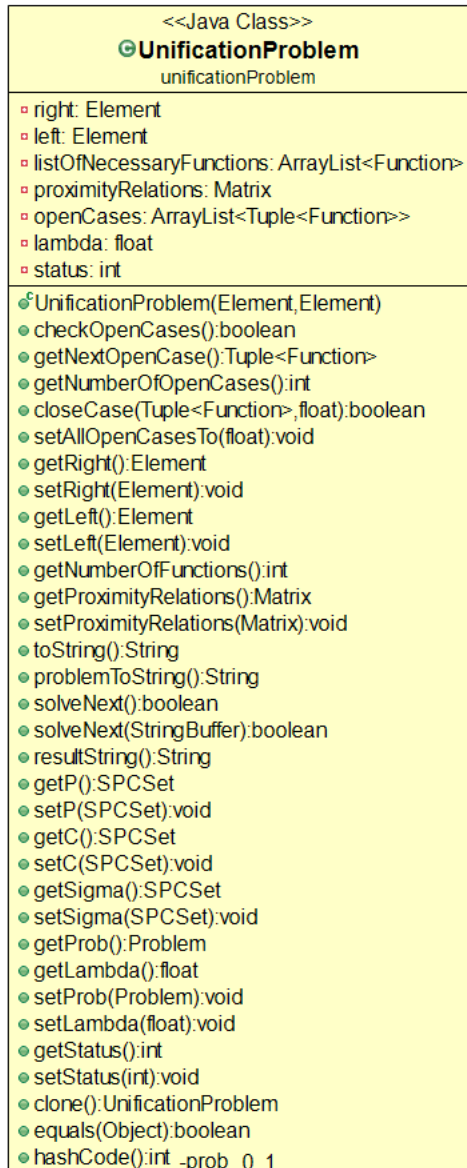
## 2 System Model

The program consists of 4 packages,

- tool
- elements
- unificationProblem
- userInterfaces







-branch

0..1

<<Java Class>> <b>SPC_CL</b> userInterfaces
br: BufferedReader intermediateResult: UnificationProblem IC: InputChecker
SPC_CL() main(String[]):int getLambda():fl... checkInput(String):boolean

### 3 Work Flow

The typical workflow looks like this:



