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

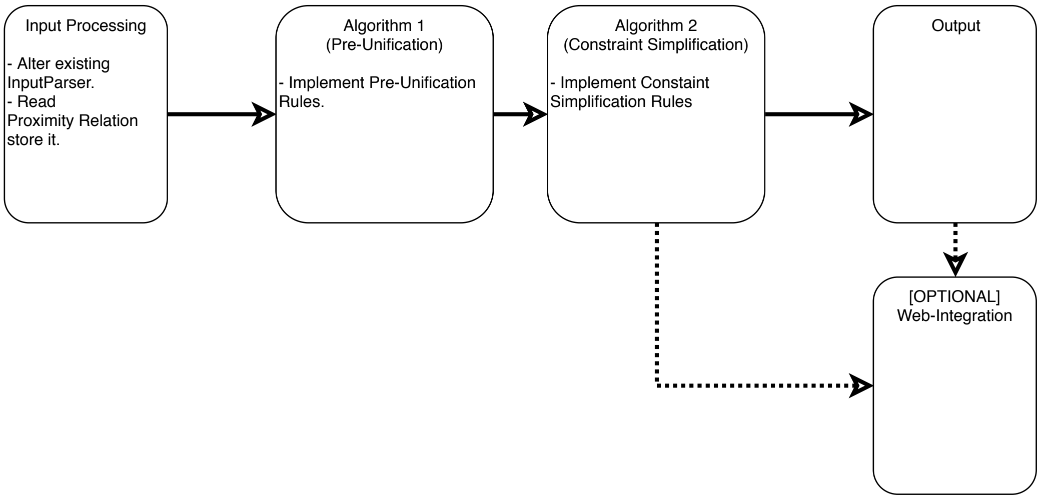
1 System Overview

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


Input Processing
- Alter existing InputParser.
- Read Proximity Relation store it.
Algorithm 1 (Pre-Unification)
- Implement Pre-Unification Rules.
Algorithm 2 (Constraint Simplification)
- Implement Constaint Simplification Rules
Output
[OPTIONAL] Web-Integration

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 $\mathcal{R}_\lambda$.

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,. . . ,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

$$\begin{array}{c|ccccccccccccc|}
 & f_{0_1} & f_{0_2} & \cdots & f_{0_{k_0}} & f_{1_1} & f_{1_2} & \cdots & f_{1_{k_1}} & \cdots & f_{m_1} & f_{m_2} & \cdots & f_{m_{k_m}} \\
f_{0_1} & 1 & (*) & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
f_{0_2} & * & 1 & (*) & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\vdots & * & * & \ddots & (*) & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
f_{0_{k_0}} & * & * & * & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
f_{1_1} & 0 & 0 & 0 & 0 & 1 & (*) & (*) & (*) & 0 & 0 & 0 & 0 & 0 \\
f_{1_2} & 0 & 0 & 0 & 0 & * & 1 & (*) & (*) & 0 & 0 & 0 & 0 & 0 \\
\vdots & 0 & 0 & 0 & 0 & * & * & \ddots & (*) & 0 & 0 & 0 & 0 & 0 \\
f_{1_{k_1}} & 0 & 0 & 0 & 0 & * & * & * & 1 & 0 & 0 & 0 & 0 & 0 \\
\vdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 & 0 & 0 \\
f_{m_1} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & (*) & (*) & (*) \\
f_{m_2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & 1 & (*) & (*) \\
\vdots & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & \ddots & (*) \\
f_{m_{k_m}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * & * & * & 1
\end{array},$$

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 aritys 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 retrive 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 $\mathrm{P} = \emptyset$ 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

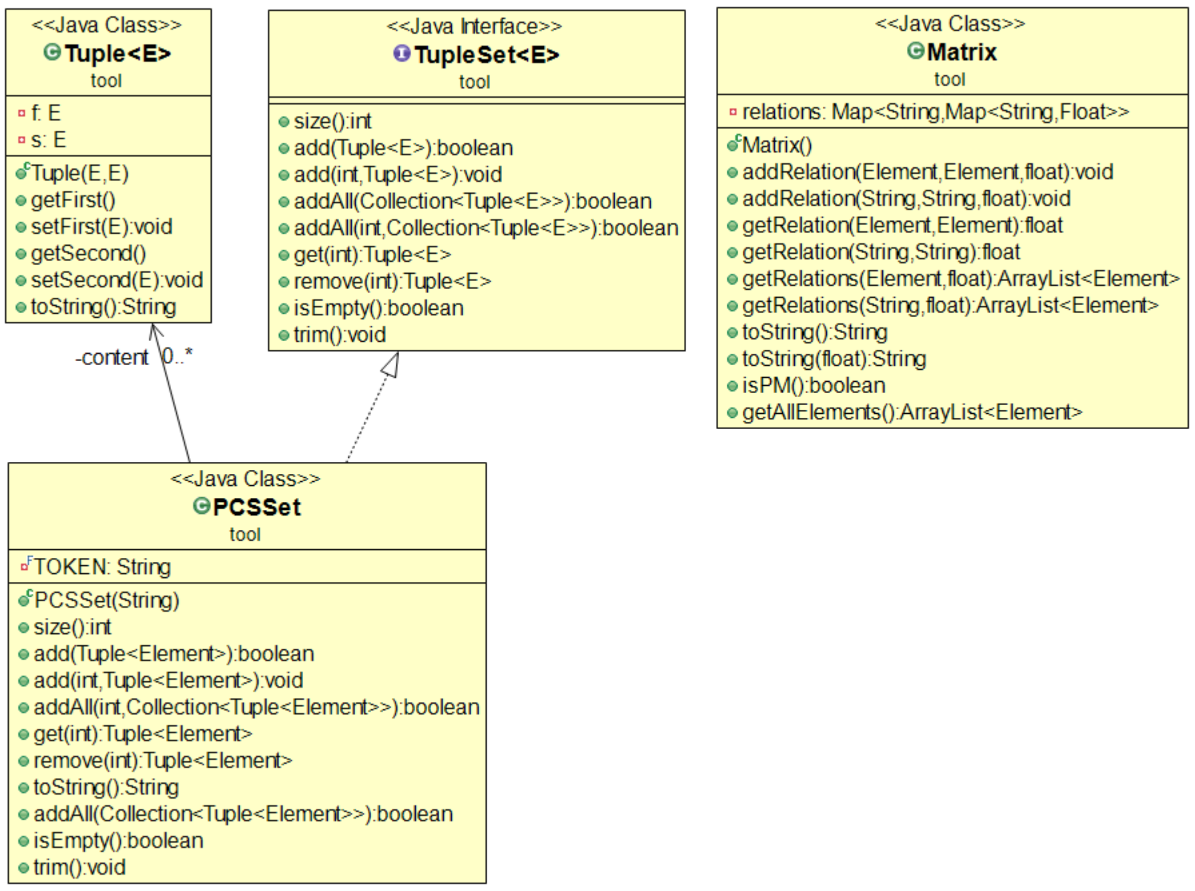
The constraintSimplification method will be called, if the preUnification method returns true. As the preUnification method, the constraintSimplification method consists of a loop. This loop runs until the set of constraints is empty. As input the method needs the problem, more precisely the set of constraints, 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.

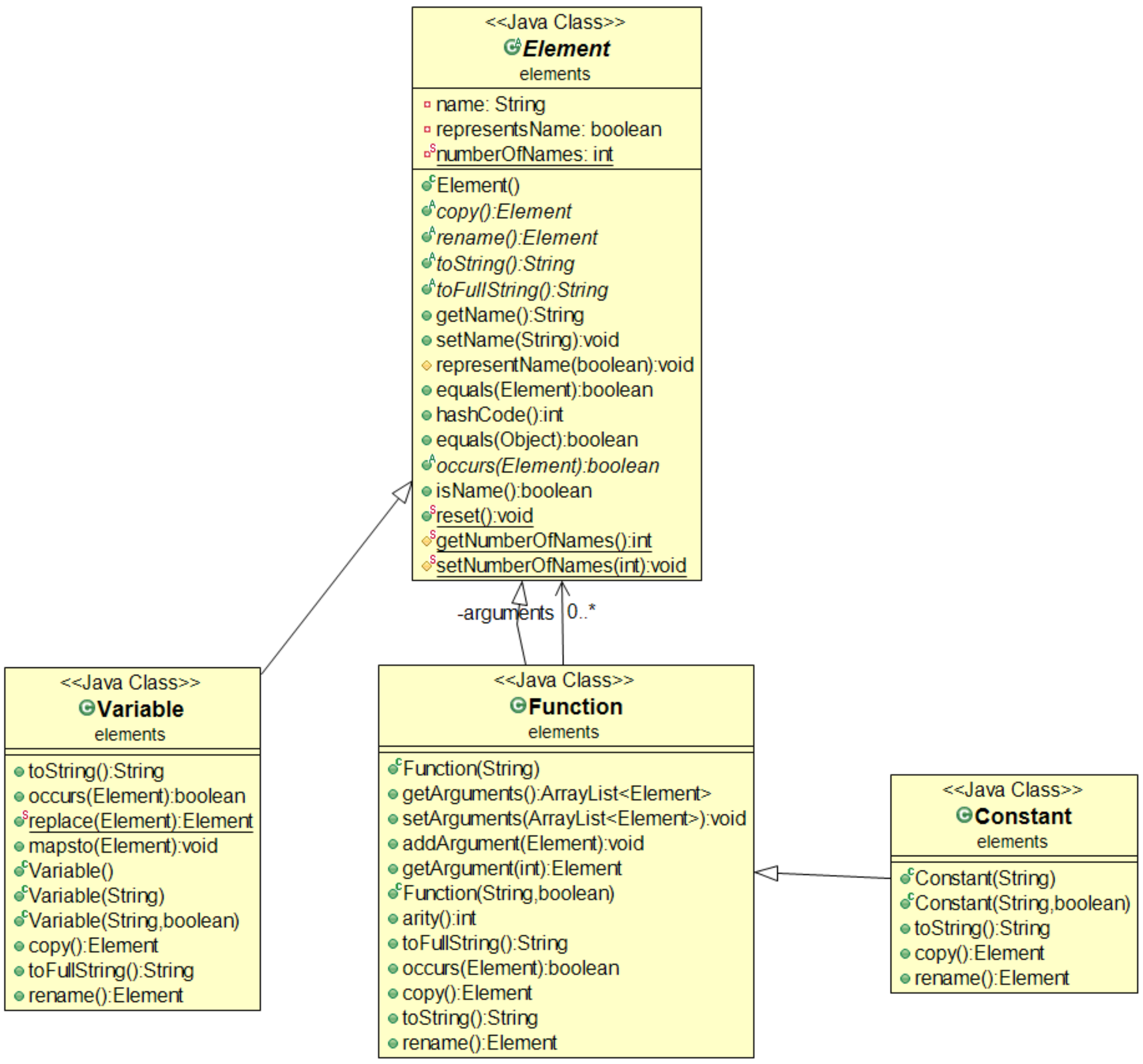
1.3 Output

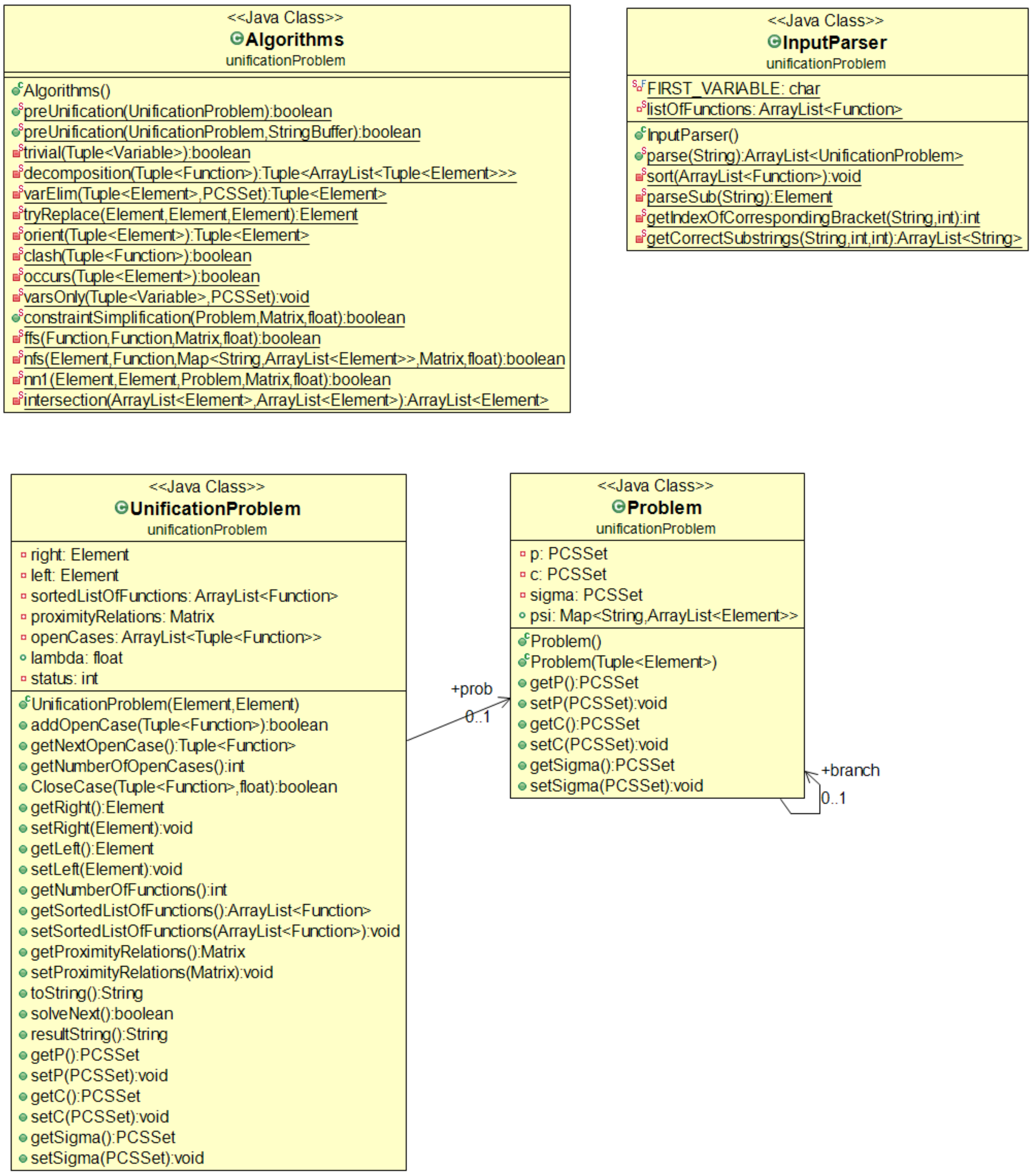
2 System Model

The program consists of 3 packages,

- tool
- elements
- unificationProblem

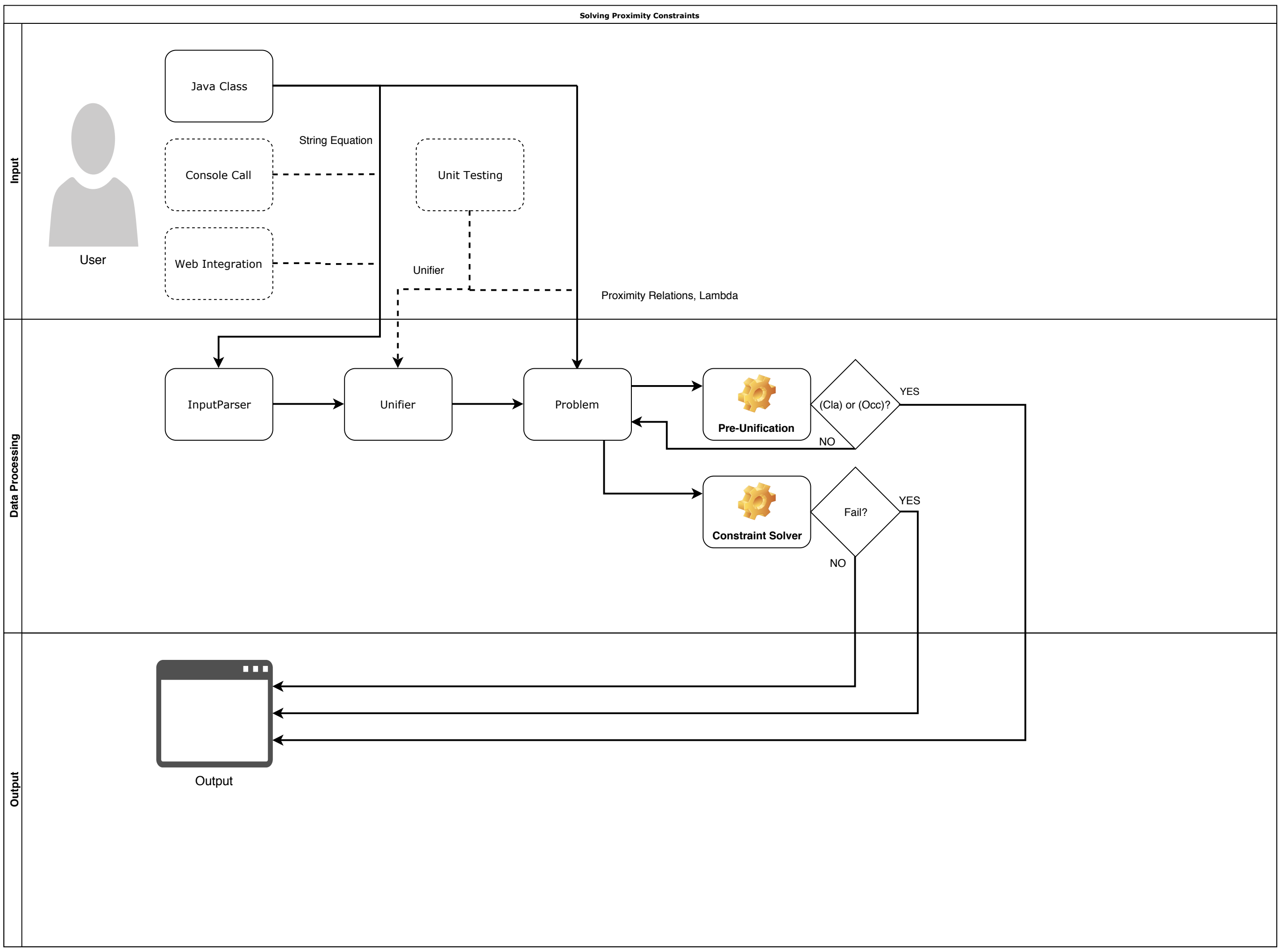

<<Java Class>>
Tuple<E>
tool
f: E
s: E
Tuple(E,E)
getFirst()
setFirst(E):void
getSecond()
setSecond(E):void
toString():String
<<Java Interface>>
TupleSet<E>
tool
size():int
add(Tuple<E>):boolean
add(int,Tuple<E>):void
addAll(Collection<Tuple<E>>):boolean
addAll(int,Collection<Tuple<E>>):boolean
get(int):Tuple<E>
remove(int):Tuple<E>
isEmpty():boolean
trim():void
<<Java Class>>
Matrix
tool
relations: Map<String,Map<String,Float>>
Matrix()
addRelation(Element,Element,float):void
addRelation(String,String,float):void
getRelation(Element,Element):float
getRelation(String,String):float
getRelations(Element,float):ArrayList<Element>
getRelations(String,float):ArrayList<Element>
toString():String
toString(float):String
isPM():boolean
getAllElements():ArrayList<Element>
-content
0..*
<<Java Class>>
PCSSet
tool
TOKEN: String
PCSSet(String)
size():int
add(Tuple<Element>):boolean
add(int,Tuple<Element>):void
addAll(int,Collection<Tuple<Element>>):boolean
get(int):Tuple<Element>
remove(int):Tuple<Element>
toString():String
addAll(Collection<Tuple<Element>>):boolean
isEmpty():boolean
trim():void


<<Java Class>>
Element
elements
name: String
representsName: boolean
numberOfNames: int
Element()
copy():Element
rename():Element
toString():String
toFullString():String
getName():String
setName(String):void
representName(boolean):void
equals(Element):boolean
hashCode():int
equals(Object):boolean
occurs(Element):boolean
isName():boolean
reset():void
getNumberOfNames():int
setNumberOfNames(int):void
-arguments
0..*
<<Java Class>>
Variable
elements
toString():String
occurs(Element):boolean
replace(Element):Element
mapsto(Element):void
Variable()
Variable(String)
Variable(String,boolean)
copy():Element
toFullString():String
rename():Element
<<Java Class>>
Function
elements
Function(String)
getArguments():ArrayList<Element>
setArguments(ArrayList<Element>):void
addArgument(Element):void
getArgument(int):Element
Function(String,boolean)
arity():int
toFullString():String
occurs(Element):boolean
copy():Element
toString():String
rename():Element
<<Java Class>>
Constant
elements
Constant(String)
Constant(String,boolean)
toString():String
copy():Element
rename():Element


<<Java Class>>
Algorithms
unificationProblem
Algorithms()
preUnification(UnificationProblem):boolean
preUnification(UnificationProblem,StringBuffer):boolean
trivial(Tuple<Variable>):boolean
decomposition(Tuple<Function>):Tuple<ArrayList<Tuple<Element>>>
varElim(Tuple<Element>,PCSSet):Tuple<Element>
tryReplace(Element,Element,Element):Element
orient(Tuple<Element>):Tuple<Element>
clash(Tuple<Function>):boolean
occurs(Tuple<Element>):boolean
varsOnly(Tuple<Variable>,PCSSet):void
constraintSimplification(Problem,Matrix,float):boolean
ffs(Function,Function,Matrix,float):boolean
nfs(Element,Function,Map<String,ArrayList<Element>>,Matrix,float):boolean
nn1(Element,Element,Problem,Matrix,float):boolean
intersection(ArrayList<Element>,ArrayList<Element>):ArrayList<Element>
<<Java Class>>
InputParser
unificationProblem
FIRST_VARIABLE: char
listOfFunctions: ArrayList<Function>
InputParser()
parse(String):ArrayList<UnificationProblem>
sort(ArrayList<Function>):void
parseSub(String):Element
getIndexOfCorrespondingBracket(String,int):int
getCorrectSubstrings(String,int,int):ArrayList<String>
<<Java Class>>
UnificationProblem
unificationProblem
right: Element
left: Element
sortedListOfFunctions: ArrayList<Function>
proximityRelations: Matrix
openCases: ArrayList<Tuple<Function>>
lambda: float
status: int
UnificationProblem(Element,Element)
addOpenCase(Tuple<Function>):boolean
getNextOpenCase():Tuple<Function>
getNumberOfOpenCases():int
CloseCase(Tuple<Function>,float):boolean
getRight():Element
setRight(Element):void
getLeft():Element
setLeft(Element):void
getNumberOfFunctions():int
getSortedListOfFunctions():ArrayList<Function>
setSortedListOfFunctions(ArrayList<Function>):void
getProximityRelations():Matrix
setProximityRelations(Matrix):void
toString():String
solveNext():boolean
resultString():String
getP():PCSSet
setP(PCSSet):void
getC():PCSSet
setC(PCSSet):void
getSigma():PCSSet
setSigma(PCSSet):void
+prob
0..1
<<Java Class>>
Problem
unificationProblem
p: PCSSet
c: PCSSet
sigma: PCSSet
psi: Map<String,ArrayList<Element>>
Problem()
Problem(Tuple<Element>)
getP():PCSSet
setP(PCSSet):void
getC():PCSSet
setC(PCSSet):void
getSigma():PCSSet
setSigma(PCSSet):void
+branch
0..1

3 Work Flow

The typical workflow looks like this:



Solving Proximity Constraints
Input
Data Processing
Output
User
Java Class
Console Call
Web Integration
String Equation
Unit Testing
Unifier
Proximity Relations, Lambda
InputParser
Unifier
Problem
Pre-Unification
(Cla) or (Occ)?
YES
NO
Constraint Solver
Fail?
YES
NO
Output