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	update system model, update matrix	Sophie
0.7	added UML	Jan-Michael
0.8	added User Interfaces	Jan-Michael

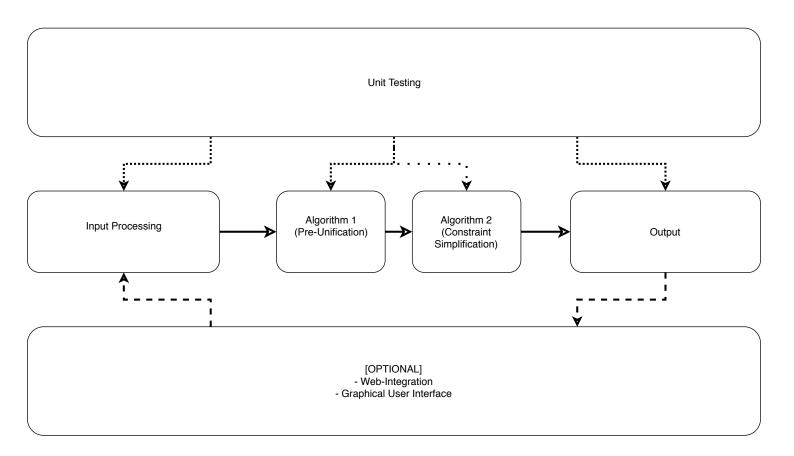
1 System Overview

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 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}_{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 $0,1,\ldots,m$. Let k_l be the number of Functions with arity l. Then

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

The matrix then looks like

		f_{0_1}	f_{0_2}		$f_{0_{k_0}}$	f_{1}	f_{1_2}		$f_{1_{k_1}}$		f_{m_1}	f_{m_2}		$f_{m_{k_m}}$	
$f_{0}{}_1$	1	1	(*)	(*)	(*)	0	0	0	0	0	0	0	0	0	\
f_{0_2}		*	1	(*)	(*)	0	0	0	0	0	0	0	0	0	
÷		*	*	٠.	(*)	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	
÷		0	0	0	0	*	*	٠	(*)	0	0	0	0	0	
$f_{^{1}k_{^{1}}}$		0	0	0	0	*	*	*	1	0	0	0	0	0	
÷		0	0	0	0	0	0	0	0	٠.	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	(*)	(*)	
÷		0	0	0	0	0	0	0	0	0	*	*	٠	(*)	
$f_{m_{k_m}}$	/	0	0	0	0	0	0	0	0	0	*	*	*	1 ,	J

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. Before asking the user to enter relation values, we will generate the list of open cases outside 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 λ , the method $getRelations(s_1,\lambda)$ can be used. Moreover it is possible to get all reation elements of the matrix by calling getAllElements(). This will return all f_{i_k} , where $0 \le i \le m$ and $0 \le j \le m$.

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. 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.

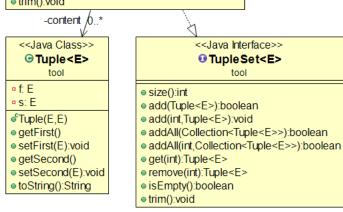
1.3 Output

2 System Model

The program consists of 4 packages,

- tool
- elements
- unificationProblem
- · userInterfaces

<<Java Class>> <<Java Class>> **GSPCSet** tool tool relations: Map<String,Map<String,Float>> content: ArrayList<Tuple<Element>> ■TOKEN: String Matrix() addRelation(Element, Element, float):void size():int addRelation(String,String,float):void oadd(Tuple<Element>):boolean getRelation(Element, Element):float getRelation(String,String):float add(int,Tuple<Element>):void getRelations(Element, float):ArrayList<Element> addAll(int,Collection<Tuple<Element>>):boolean get(int):Tuple<Element> o getRelations(String,float):ArrayList<Element> o remove(int):Tuple<Element> toString():String toString():String toString(float):String • addAll(Collection<Tuple<Element>>):boolean • isPM():boolean isEmpty():boolean getAllElements():ArrayList<Element> trim():void



<<Java Class>> @Element elements

- name: String
- representsName: boolean
- ^SnumberOfNames: int
- dcopy():Element
- delta to String (): String

 delta to String (): String (): String

 delta to String (): String (): String (): String

 delta to String (): String ():
- toFullString():String
 getName():String
- setName(String):void
- representName(boolean):void
- equals(Element):boolean
- hashCode():int
- equals(Object):boolean
- display="block">
 occurs(Element):boolean
- isName():boolean
- §reset():void
- getNumberOfNames():int
- setNumberOfNames(int):void

-arguments 0..*

<<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
- ocopy():Element
- toString():String
- rename():Element

<<Java Class>>

⊕Constant

elements

- Constant(String,boolean)
- toString():String
- ocopy():Element
- rename():Element

<<Java Class>>

⊙Variable

elements

- toString():String
- occurs(Element):boolean
- sreplace(Element):Element
- mapsto(Element):void
- Variable()
- √String)
- Variable(String,boolean)
- copy():Element
- toFullString():String rename():Element

<<Java Class>>

⊕UnificationProblem

unificationProblem

- □ right: Element
- □ left: Element
- sortedListOfFunctions: ArrayList<Function>
- proximityRelations: Matrix
- openCases: ArrayList<Tuple<Function>>
- lambda: float
- a status: int

√UnificationProblem(Element, Element)

- addOpenCase(Tuple<Function>):boolean
- o 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
- solveNext(StringBuffer):boolean
- resultString():String
- getP():SPCSet
- setP(SPCSet):void
- getC():SPCSet
- setC(SPCSet):void
- getSigma():SPCSet
- setSigma(SPCSet):void
- getProb():Problem
- getLambda():float
- setProb(Problem):void
- setLambda(float):void

-prob 0..1

<<Java Class>>

⊕Problem

unificationProblem

- p: SPCSet
- c: SPCSet
- □ sigma: SPCSet
- psi: Map<String,ArrayList<Element>>
- Problem()
- ◆Problem(Tuple<Element>)
- getP():SPCSet
- setP(SPCSet):void
- getC():SPCSet setC(SPCSet):void
- getSigma():SPCSet
- setSigma(SPCSet):void

+branch 0..1

<<Java Class>>

unificationProblem

- €Algorithms()
- SpreUnification(UnificationProblem):boolean

- decomposition(Tuple<Function>):Tuple<ArrayList<Tuple<Element>>>
- silvarElim(Tuple<Element>,SPCSet):Tuple<Element>
- stryReplace(Element,Element,Element):Element
- sorient(Tuple<Element>):Tuple<Element>
- soccurs(Tuple<Element>):boolean
 soccurs(Tuple≤Element>):boolean
 soccurs(Tuple≤Element>):bool
- svarsOnly(Tuple<Variable>,SPCSet):void
- SconstraintSimplification(UnificationProblem):boolean
- SconstraintSimplification(UnificationProblem,StringBuffer):boolean
- SconstraintSimp(Problem,Matrix,float):boolean
- ffs(Function, Function, Matrix, float): boolean
- snfs(Element, Function, Map<String, ArrayList<Element>>, Matrix, float):boolean
- sinn1 (Element, Element, Problem, Matrix, float): boolean

 € 100 (Element, Element, Problem, Matrix, float): boolean

 € 200 (Element, Element, Problem, Matrix, float): boolean

 € 200 (Element, Element, Problem, Matrix, float): boolean
- Sintersection(ArrayList<Element>.ArrayList<Element>):ArrayList<Element>



3 Work Flow

The typical workflow looks like this:

