Verigraph: a system for specification and analysis of graph grammars

Content

Content

High-Level Replacement

- Generalizes graph transformation
- Assumes adhesive category
  - Pushouts along monomorphisms
  - Pullbacks
  - Pushouts along monomorphisms are Van Kampen squares

High-Level Replacement

- A **production** $\rho = \langle l, r \rangle$ is a monomorphic span:

$$L \xleftarrow{\;l\;} K \xrightarrow{\;r\;} R$$

- Given a production $\rho = \langle l, r \rangle$ and a match m, a **transformation** is given by a double-pushout diagram:

$$\begin{tikzcd}
L \arrow[d, "m"'] & K \arrow[l, tail, "l"'] \arrow[r, tail, "r"] \arrow[d] & R \arrow[d] \\
G & D \arrow[l, tail] \arrow[r, tail] & H
\end{tikzcd}$$

HLR in Haskell

- Goal: generic algorithms with respect to the category
- Solution: type classes

Type Class: Morphism

```
class (Eq m) => Morphism m where
    type Obj m :: *
    compose  :: m -> m -> m
    domain   :: m -> Obj m
    codomain :: m -> Obj m
    id       :: Obj m -> m
    monomorphism :: m -> Bool
    epimorphism :: m -> Bool
    isomorphism :: m -> Bool
```

- Class for morphisms of a category
- Note: full subcategory of finite objects

Type Class: AdhesiveHLR

```haskell
class (Morphism m) => AdhesiveHLR m where
  -- Assumes one of the morphisms is mono
  po :: m -> m -> (m, m)

  hasPoc :: m -> m -> Bool

  -- Assumes a pushout complement exists
  poc :: m -> m -> (m, m)

  -- Assumes both morphisms are mono
  injectivePullback :: m -> m -> (m, m)
```

- Class for morphisms of an adhesive category
- Pullback restricted to monomorphisms for performance

Data Type: Production

```haskell
data Production m = Production
  { left :: m   -- ^ The morphism /K -> L/ of a production
  , right :: m  -- ^ The morphism /K -> R/ of a production
  }
  deriving (Show, Read)
```

$$
\begin{tikzcd} L & K \arrow[l, tail, "\texttt{left p}"'] \arrow[r, tail, "\texttt{right p}"] & R \end{tikzcd}
$$

Algorithm: Gluing Condition

```haskell
satsGluing :: AdhesiveHLR m => m -> Production m -> Bool
satsGluing m (Production l _) = hasPoc m l
```

$$
\begin{tikzcd}
L \arrow[d, "m"'] & K \arrow[l, tail, "l"'] \arrow[r, tail, "r"] \arrow[d, "k"] & R \arrow[d, "n"] \\
G & D \arrow[l, tail, "f"] \arrow[r, tail, "g"'] & H
\end{tikzcd}
$$

Algorithm: Transformation

```haskell
-- Assumes the match satisfies the gluing condition
dpo :: AdhesiveHLR m => m -> Production m -> (m, m, m, m)
dpo m (Production l r) =
  let (k, f) = poc m l
      (n, g) = po k r
  in (k, n, f, g)
```

$$
\begin{tikzcd}
L \arrow[d, "m"'] & K \arrow[l, tail, "l"'] \arrow[r, tail, "r"] \arrow[d, "k"] & R \arrow[d, "n"] \\
G & D \arrow[l, tail, "f"] \arrow[r, tail, "g"'] & H
\end{tikzcd}
$$

Negative Application Conditions

- A **NAC** is defined by a morphism $n : L \to N$.
- $m \models NAC(n) \iff \nexists q$ such that the diagram commutes:

$$\begin{tikzcd} L \arrow[r, "n"] \arrow[d, "m"'] & N \arrow[dl, dotted, "q"] \\ G & \end{tikzcd}$$

Data Type Revised: Production

```haskell
data Production m = Production
  { left :: m   -- ^ The morphism /K -> L/ of a production
  , right :: m  -- ^ The morphism /K -> R/ of a production
  , nacs :: [m] -- ^ The set of nacs /L -> Ni/ of a production
  }
  deriving (Show, Read)
```

Type Class: FindMorphism

```
data PROP = ALL | MONO | EPI | ISO
```

```
class Morphism m => FindMorphism m where
  -- Find all morphisms between two Obj m
  matches :: PROP -> Obj m -> Obj m -> [m]
```

- Used for checking NAC satisfaction.

Type Class: EpiPairs

```haskell
class Morphism m => EpiPairs m where
  -- Create all jointly epimorphic pairs of morphisms from the given objects
  createPairs :: PROP -> Obj m -> Obj m -> [(m,m)]

  -- Generates all epimorphisms from an object (up to isomorphism)
  partitions :: Bool -> Obj m -> [m]
```

- A jointly epimorphic pair from two objects:

$$
\begin{tikzcd}
A \arrow[dr] & & B \arrow[dl] \\
& X &
\end{tikzcd}
$$

Content

Valid

```haskell
class Valid a where
    valid :: a -> Bool
```

Relation

```haskell
module Abstract.Relation (
      compose, inverse, update
    , ...
    , apply
    , ...
    , orphans
    , ...
    , functional, injective, surjective, total ...)

data Relation a = Relation {
       domain   :: [a],
       codomain :: [a],
       mapping  :: Map.Map a [a]
   } deriving (Ord,Show,Read)
```

Graphs

```haskell
data Node a = Node { getNodePayload :: Maybe a
                   } deriving (Show, Read)
```

```haskell
data Edge a = Edge { getSource      :: NodeId
                   , getTarget      :: NodeId
                   , getEdgePayload :: Maybe a
                   } deriving (Show, Read)
```

```haskell
newtype NodeId = NodeId Int deriving (Eq, Ord, Read)
newtype EdgeId = EdgeId Int deriving (Eq, Ord, Read)
```

```haskell
data Graph a b = Graph {
    nodeMap :: [(NodeId, Node a)],
    edgeMap :: [(EdgeId, Edge b)]
    } deriving (Read)
```

Graph Morphisms

- Graph Morphisms

```
data GraphMorphism a b = GraphMorphism {
                            getDomain    :: Graph a b
                          , getCodomain  :: Graph a b
                          , nodeRelation :: Relation NodeId
                          , edgeRelation :: Relation EdgeId
                        } deriving (Read)
```

- Typed Graphs

```
type TypedGraph a b = GraphMorphism a b
```

Morphism (Graph Morphism)

```haskell
instance Morphism (GraphMorphism a b) where
    type Obj (GraphMorphism a b) = Graph a b

    domain = getDomain
    codomain = getCodomain
    compose m1 m2 =
        GraphMorphism (domain m1)
                      (codomain m2)
                      (R.compose (nodeRelation m1) (nodeRelation m2))
                      (R.compose (edgeRelation m1) (edgeRelation m2))
    id g = GraphMorphism g g (R.id $ nodes g) (R.id $ edges g)
    monomorphism m =
        R.injective (nodeRelation m) &&
        R.injective (edgeRelation m)
    epimorphism m =
        R.surjective (nodeRelation m) &&
        R.surjective (edgeRelation m)
    isomorphism m =
        monomorphism m && epimorphism m
```

Typed Graph Morphisms

```haskell
data TypedGraphMorphism a b =
  TypedGraphMorphism {
    getDomain   :: GraphMorphism a b
  , getCodomain :: GraphMorphism a b
  , mapping  :: GraphMorphism a b
} deriving (Show, Read)
```

```haskell
instance Valid (TypedGraphMorphism a b) where
    valid (TypedGraphMorphism dom cod m) =
        valid dom &&
        valid cod &&
        dom == compose m cod
```

Morphism (Typed Graph Morphisms)

```haskell
instance Morphism (TypedGraphMorphism a b) where
    type Obj (TypedGraphMorphism a b) = GraphMorphism a b

    domain = getDomain
    codomain = getCodomain
    compose t1 t2 =
        TypedGraphMorphism (domain t1)
                           (codomain t2)
                          $ compose (mapping t1)
                                    (mapping t2)
    id t = TypedGraphMorphism t t (M.id $ domain t)
    monomorphism = monomorphism . mapping
    epimorphism = epimorphism . mapping
    isomorphism = isomorphism . mapping
```

First-Order Rules

```
type GraphRule a b = Production (TypedGraphMorphism a b)
...
createdNodes :: GraphRule a b -> [NodeId]
createdNodes rule = orphanNodesTyped (right rule)
...
satsIncEdges :: TypedGraphMorphism a b -> TypedGraphMorphism a b -> Bool
satsIdent :: TypedGraphMorphism a b -> TypedGraphMorphism a b -> Bool

ruleDeletes ...

instance AdhesiveHLR (TypedGraphMorphism a b) where
  hasPoc ...
  po ...
  poc ...
```

First-Order Rule Morphisms

```haskell
data RuleMorphism a b =
  RuleMorphism {
    getDomain         :: Production (TypedGraphMorphism a b)
  , getCodomain       :: Production (TypedGraphMorphism a b)
  , mappingLeft       :: TypedGraphMorphism a b
  , mappingInterface :: TypedGraphMorphism a b
  , mappingRight      :: TypedGraphMorphism a b
  } deriving (Show)
```

```haskell
instance Valid (RuleMorphism a b) where
    valid (RuleMorphism dom cod mapL mapK mapR) =
        valid dom && valid cod &&
        valid mapL && valid mapK && valid mapR &&
        compose mapK (left cod) == compose (left dom) mapL &&
        compose mapK (right cod) == compose (right dom) mapR
```

First-Order Rule Morphisms

```
type Obj (RuleMorphism a b) = Production (TypedGraphMorphism a b)

compose t1 t2 =
    RuleMorphism (domain t1) (codomain t2)
                 (compose (mappingLeft t1) (mappingLeft t2))
                 (compose (mappingInterface t1) (mappingInterface t2))
                 (compose (mappingRight t1) (mappingRight t2))

monomorphism rm =
  monomorphism (mappingLeft rm) &&
  monomorphism (mappingInterface rm) &&
  monomorphism (mappingRight rm)

isomorphism (RuleMorphism dom cod mapL mapK mapR) =
  isomorphism mapL &&
  isomorphism mapK &&
  isomorphism mapR &&
  compose (left dom) mapL == compose mapK (left cod) &&
  compose (right dom) mapR == compose mapK (right cod)
```

Second-Order Rules

```
type SndOrderRule a b = Production (RuleMorphism a b)
...

danglingSpan ...
addMinimalSafetyNacs ...

...
instance AdhesiveHLR (RuleMorphism a b) where
  hasPoc ...
  po ...
  poc ...
```

Pushout (Second-Order Rules)

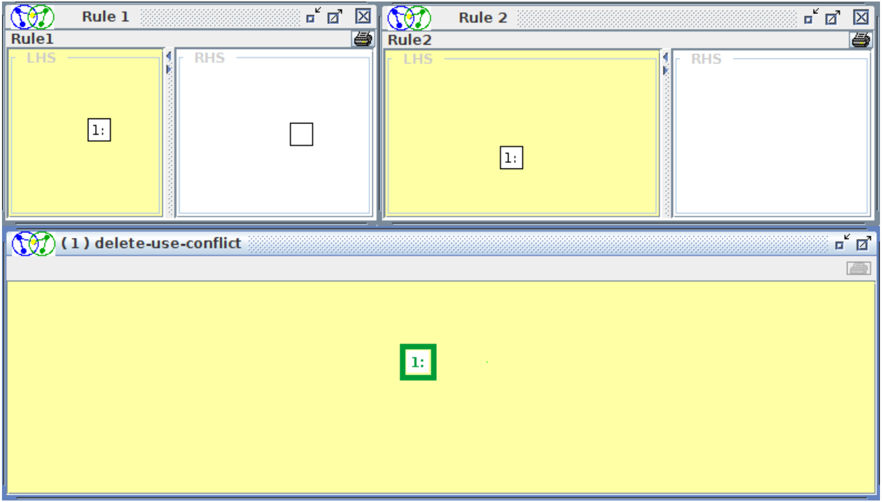
```
po (RuleMorphism _ ruleD matchL matchK matchR)
   (RuleMorphism _ ruleR rightL rightK rightR) =  (m',r')
   where
     (matchL', rightL') = po matchL rightL
     (matchK', rightK') = po matchK rightK
     (matchR', rightR') = po matchR rightR
     l = commutingMorphismSameDomain
           rightK' (compose (left ruleD) rightL')
           matchK' (compose (left ruleR) matchL')
     r = commutingMorphismSameDomain
           rightK' (compose (right ruleD) rightR')
           matchK' (compose (right ruleR) matchR')
     newRule = production l r []
     m'  = RuleMorphism ruleR newRule matchL' matchK' matchR'
     r'  = RuleMorphism ruleD newRule rightL' rightK' rightR'
```

Content

Implemented Algorithms

- Critical Pairs
  - Delete-Use
  - Produce-Dangling
  - Produce-Forbid
- Critical Sequences
  - Produce-Use
  - Remove-Dangling
  - Deliver-Delete
- Concurrent Rules
  - Concurrent Rules
  - Downward Shift
  - Left Shift

Delete-Use Example



Delete-Use

$$
\begin{tikzcd}
 & & N_1 & & N_2 & & \\
R_1 & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"'] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"] \arrow[dlll, dotted, bend right, "!_{h_{21}}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
 & D_1 \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}
$$

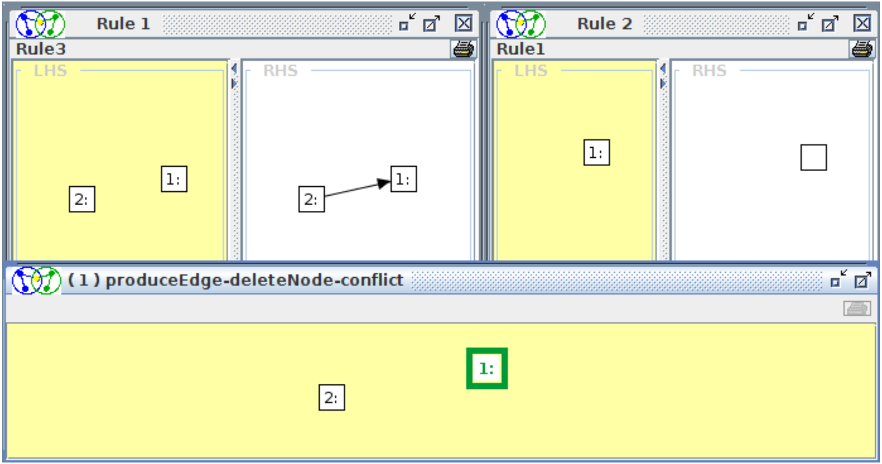
There is not exists $h_{21} : L_2 \to D_1 : d_1 \circ h_{21} = m_2$ and (m_1, m_2) jointly surjective.

Delete-Use

```haskell
allDeleteUse :: (EpiPairs m, DPO m) =>
  Bool -> Bool -> Production m -> Production m -> [CriticalPair m]
allDeleteUse nacInj inj l r =
 map (\match -> CriticalPair match Nothing Nothing DeleteUse) delUse
  where
    pairs = createPairsCodomain inj (left l) (left r)
    gluing = filter (\(m1,m2) -> satsGluingNacs nacInj inj (l,m1) (r,m2)) pairs
    delUse = filter (deleteUse inj l) gluing

deleteUse :: DPO m => Bool -> Production m -> (m, m) -> Bool
deleteUse inj l (m1,m2) = null matchD
  where
    (_,d1) = RW.poc m1 (left l)
    l2TOd1 = matches (flagInj inj) (domain m2) (domain d1)
    matchD = filter (\x -> m2 == compose x d1) l2TOd1
```

Produce-Dangling Example



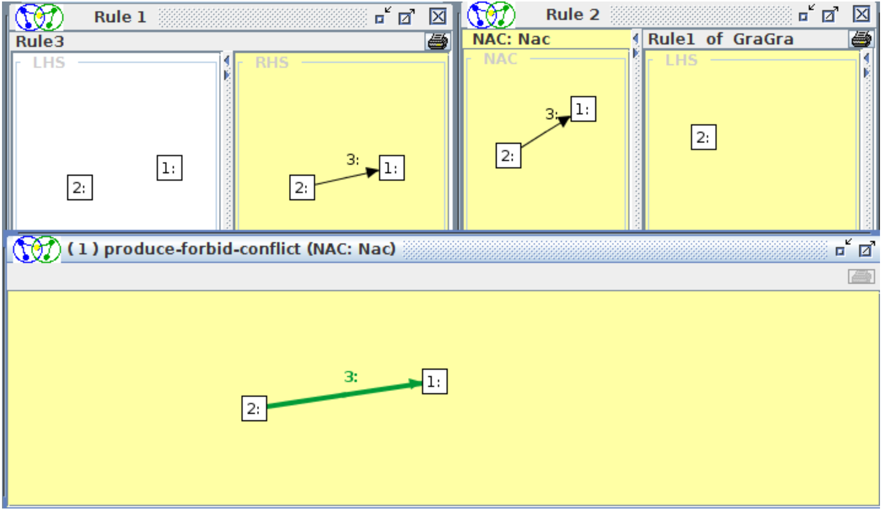
Produce-Dangling

$$
\begin{tikzcd}
& & N_1 & & N_2 & & \\
R_1 \arrow[d, "m_1'"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"] & L_1 \arrow[u, "n_1"] \arrow[dr, "m_1"] & & L_2 \arrow[u, "n_2"'] \arrow[dl, "m_2"] \arrow[dllll, bend right, "he_{21}"'] \arrow[dlll, bend right, "h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
P_1 & D_1 \arrow[l, "e_1"] \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}
$$

There exists $h_{21} : L_2 \to D_1 : d_1 \circ h_{21} = m_2$ and $he_{21} : L_2 \to P_1 : h_{21} \circ e_1 = he_{21}$, let (1) the *initial pushout* of he_{21}, there is not exists $b_2^* : B_2 \to K_2 : l_2 \circ b_2^* = b_2$, and (m_1, m_2) jointly surjective.

$$
\begin{tikzcd}
B_2 \arrow[r, "b_2"] \arrow[d] \arrow[rr, dotted, bend left, "!b_2^*"] \arrow[dr, phantom, "(1)"] & L_2 \arrow[d, "he_{21}"] & K_2 \arrow[l, "l_2"'] \\
C_2 \arrow[r] & P_1 &
\end{tikzcd}
$$

Produce-Forbid Example



Produce-Forbid

$$
\begin{tikzcd}
 & & N_1 & & N_2 \arrow[ddllll, bend right, "q21"'] & & \\
R_1 \arrow[d, "m_1'"'] & K_1 \arrow[l, "r_1"'] \arrow[r, "l_1"] \arrow[d, "k_1"] & L_1 \arrow[u, "n_1"'] \arrow[dr, "m_1"] & & L_2 \arrow[u, "n_2"] \arrow[dl, "m_2"] \arrow[dlll, bend right, "h_{21}"'] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
P_1 & D_1 \arrow[l, "e_1"] \arrow[rr, "d_1"'] & & G & & &
\end{tikzcd}
$$

There exists $h_{21} : L_2 \to D_1 : d_1 \circ h_{21} = m_2$ but for one of the NACs $n_2 : L_2 \to N_2$ of p_2 there exists an injective morphism $q_{21} : N_2 \to P_1 : q_{21} \circ n_2 = e_1 \circ h_{21}$ and thus, $e_1 \circ h_{21} \not\models n_2$, and (m_1', q_{21}) jointly surjective.

Critical Sequences

- Same algorithm of Critical Pairs
- Inverts left rule
- Shift all nacs

$$
\begin{tikzcd}
 & & N_1' & & N_2 & & \\
L_1 \arrow[d, "m_1"'] & K_1 \arrow[l, "l_1"'] \arrow[r, "r_1"] \arrow[d] & R_1 \arrow[u, "n_1'"'] \arrow[dr, "m_1'"'] & & L_2 \arrow[u, "n_2"] \arrow[dl, "m_2'"] & K_2 \arrow[l, "l_2"'] \arrow[r, "r_2"] & R_2 \\
G & D_1 \arrow[l] \arrow[rr] & & \widehat{P} & & &
\end{tikzcd}
$$

-

Implemented Algorithms

- Concurrent Rules:
  - $n = 0$ The *concurrent rule* p_c with NACs for rule p_0 with NACs is p_0 with NACs itself.
  - $n \geqslant 1$ A concurrent rule $p_c = p'_c *_E p_n$ with NACs for the rule sequence $p_0, \ldots, p_{n-1}, p_n$is defined recursively as
    $p_c = (l_c \circ k_c : K \to L, r_n \circ k_n : K \to R)$

$$
\begin{tikzcd}
N_i & & & & N_j & & \\
L'_c \arrow[u, "n_i"] \arrow[d, "l'"'] \arrow[dr, phantom, "(3)"] & K'_c \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & R'_c \arrow[dr, "e_1"] & & L_n \arrow[u, "n_j"] \arrow[dl, "e_2"'] & K_n \arrow[l] \arrow[r] \arrow[d] \arrow[dl, phantom, "(2)"] \arrow[dr, phantom, "(4)"] & R_n \arrow[d] \\
L & C_c \arrow[l, "l_c"] \arrow[rr, "r_c"'] & & E \arrow[d, phantom, "(5)"] & & C_n \arrow[ll, "l_n"] \arrow[r, "r_n"'] & R \\
& & & K \arrow[ull, dotted, "k_c"] \arrow[urr, dotted, "k_n"'] & & &
\end{tikzcd}
$$

Implemented Algorithms

```
concurrentRuleForPair :: (DPO m, EpiPairs m, Eq (Obj m)) => Bool
  -> Production m -> Production m -> (m, m) -> Production m
concurrentRuleForPair inj c n pair = production l r (dmc ++ lp)
  where
    pocC = poc (fst pair) (right c)
    pocN = poc (snd pair) (left n)
    poC = po (fst pocC) (left c)
    poN = po (fst pocN) (right n)
    pb = injectivePullback (snd pocC) (snd pocN)
    l = compose (fst pb) (snd poC)
    r = compose (snd pb) (snd poN)
    dmc = concatMap (downwardShift inj (fst poC)) (nacs c)
    inverseP = production (snd pocC) (snd poC) []
    den = concatMap (downwardShift inj (snd pair)) (nacs n)
    lp = concatMap (shiftLeftNac inj inverseP) den
```

Implemented Algorithms

- NAC shifted over a morphism:

$$\begin{tikzcd} N'_j \arrow[r, dotted, "e_{ji}"] & N_i \\ A \arrow[u, "n'_j"] \arrow[r, "m"'] \arrow[ur, phantom, "="] & B \arrow[u, "n_i"'] \end{tikzcd}$$

  For each $NAC(n'_j)$ on A with $n'_j : A \to N'_j$ and $m : A \to B$, let

  $$D_m(NAC(n'_j)) = \{NAC(n_i) | i \in I, n_i : B \to N_i\}$$

  where I and n_i are constructed as follows:

  - $i \in I$ iff (e_{ji}, n_i) with $e_{ji} : N'_j \to N_i$ jointly surjective
  - $e_{ji} \circ n_i = n_i \circ m$
  - e_{ji} injective

Implemented Algorithms

- NAC shifted over a morphism:

```
downwardShift :: EpiPairs m => Bool -> m -> m -> [m]
downwardShift inj m n = newNacs
  where
    pairs = commutingPairsAlt (n,True) (m,inj)
    newNacs = map snd pairs
```

Implemented Algorithms

- Left NACs from Right NACs
  For each $NAC(n_i)$ on R of a rule, the equivalent left application condition $L_p(NAC(n_i))$ is defined in the following way:

$$\begin{tikzcd} L \arrow[d, "n_i'"'] \arrow[r, phantom, "(2)"] & K \arrow[l] \arrow[r] \arrow[d] \arrow[r, phantom, "(1)"] & R \arrow[d, "n_i"] \\ N_i' & D \arrow[l] \arrow[r] & N_i \end{tikzcd}$$

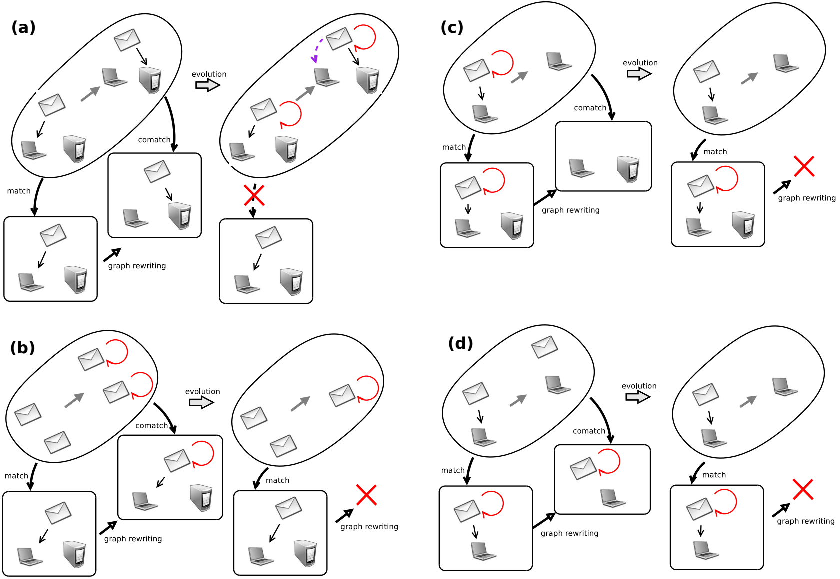
  - If the pair $(K \to R, R \to N_i)$ has a pushout complement, we construct $(K \to D, D \to N_i)$ as the pushout complement (1). Then we construct pushout (2) with the morphism $n_i' : L \to N_i'$.
  - If the pair $(K \to R, R \to N_i)$ does not have a pushout complement, we define $L_p(NAC(n_i)) = true$

```
shiftLeftNac inj rule n = [comatch n rule | satsGluing inj n
↪  (left rule)]
```

Interlevel Conflicts

$$
\begin{tikzcd}
a_3 \arrow[ddd, "m_3"] & & & b_3 \arrow[lll] \arrow[rrr] \arrow[ddd] & & & c_3 \arrow[ddd] & & \\
& a_2 \arrow[ul] \arrow[dr] \arrow[ddd, "m_2"] & & & b_2 \arrow[lll] \arrow[rrr] \arrow[ul] \arrow[dr] \arrow[ddd] & & & c_2 \arrow[ul] \arrow[dr] \arrow[ddd] & \\
& & a_1 \arrow[ddd, "m_1"] & & & b_1 \arrow[lll] \arrow[rrr] \arrow[ddd] & & & c_1 \arrow[ddd] \\
R \arrow[ddd] & & & R' \arrow[lll] \arrow[rrr] & & & R'' & & \\
& K \arrow[ul, "r"] \arrow[dr, "l"] \arrow[ddd] & & & K' \arrow[lll] \arrow[rrr] \arrow[ul] \arrow[dr] & & & K'' \arrow[ul, "r''"] \arrow[dr, "l''"] & \\
& & L \arrow[ddd, "m_0"] & & & L' \arrow[lll, "f_L"] \arrow[rrr, "g_L"] & & & L'' \arrow[ddllllll, "m''_0"] \\
H & & & & & & & & \\
& D \arrow[ul] \arrow[dr] & & & & & & & \\
& & G & & & & & &
\end{tikzcd}
$$

Interlevel Conflicts Examples



Content

IO

- AGG
  - Input
    - .ggx
  - Output
    - .cpx
    - .ggx
- Haskell *Read* and *Show*

Usage

- Commands
  - analysis
    - –snd-order
    - –conflicts-only
    - –dependencies-only
  - concurrent-rule
    - –max-rule
    - –all-rules
    - –by-dependency
  - snd-order
- Options
  - –all-matches
  - –inj-nac-satisfaction

Content

Functionalities under study

- Attributed Graph Grammars
  - DPO for the Category of Algebras
  - Critical Pairs
  - Concurrent Rules
- Second-order with non-injective matches
- Improvement of inter level CP algorithm

Content

Future Work

- Graphical User Interface
- Initial Pushout and Critical Objects
- Inheritance
- AGREE