Planning on Graph Transformation Systems

2018-03-30

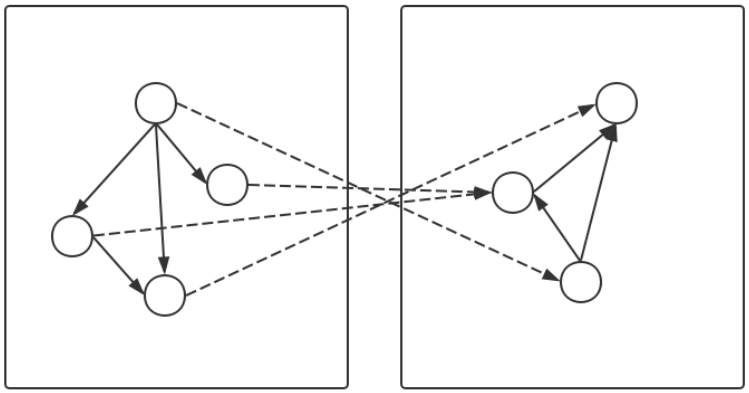
Graph Transformation

- *In computer science, graph transformation, or graph rewriting, concerns the technique of creating a new graph out of an original graph algorithmically. It has numerous applications, ranging from software engineering (software construction and also software verification) to layout algorithms and picture generation.*
- *Graph transformations can be used as a computation abstraction. The basic idea is that the state of a computation can be represented as a graph, further steps in that computation can then be represented as transformation rules on that graph. Such rules consist of an original graph, which is to be matched to a subgraph in the complete state, and a replacing graph, which will replace the matched subgraph.*

Some Definitions (1)

Definition 1 (Graph, Graph Morphism). *A* graph $G = (V_G, E_G, src_G, tgt_G)$ *consists of a set of nodes* V_G, *a set of edges* E_G, *and source and target functions* $src_G, tgt_G : E_G \to V_G$. *A* graph morphism $f : G \to H$ *between two graphs is a pair of mappings* $f = (f_E, f_V)$ *with* $f_E : E_G \to E_H$ *and* $f_V : V_G \to V_H$ *such that* $f_V \circ src_G = src_H \circ f_E$ *and* $f_V \circ tgt_G = tgt_H \circ f_E$. *A graph morphism* $f = (f_E, f_V)$ *is* injective *if* f_E *and* f_V *are injective.*

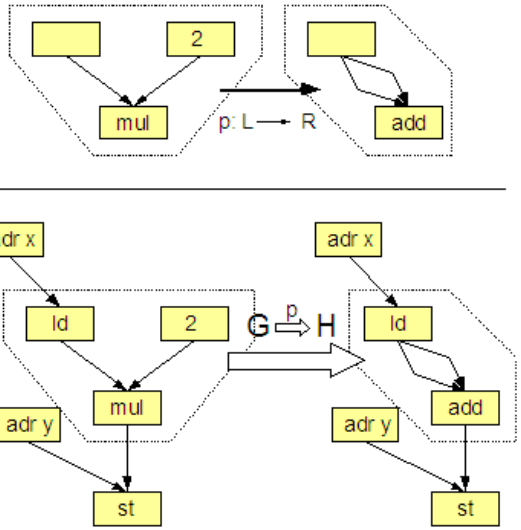
- A graph morphism is a mapping of nodes and edges of one graph to nodes and edges of another graph such that the source and target nodes of edges are preserved. Such morphisms are used in graph transformation rules to define which nodes and edges are created, deleted, or preserved when the rule is applied to a graph.



Some Definitions (2)

Definition 2 (Graph Transformation Rule)**.** *A* graph transformation rule $p = (L, R, r)$ *consists of two graphs L and R, called* left-hand side (LHS) *and* right-hand side (RHS)*, and an injective partial graph morphism* $r : L \to R$*, called* rule morphism. *Given a graph transformation rule p and a match* $m : L \to G$ *of its LHS into a* host graph *G, the* direct derivation *from G with p at m, written* $G \overset{p,m}{\Longrightarrow} H$*, is the pushout of r and m in* $Graph^P$*, the category of graphs and partial graph morphisms, as shown below.*

$$\begin{array}{ccc} L & \xrightarrow{\quad r \quad} & R \\ {\scriptstyle m}\downarrow & (PO) & \downarrow{\scriptstyle m'} \\ G & \xrightarrow{\quad r' \quad} & H \end{array}$$



- Whether a graph transformation rule can be applied to a graph depends on whether a match of its LHS to the graph can be found.
- Multiple such matches result in multiple direct derivations and thus in multiple successor graphs.

Some Definitions (3)

Definition 3 (Negative Application Condition)**.** *Let* $p = (L, R, r)$ *be a graph transformation rule,* G *a graph, and* $m : L \to G$ *a match. A* negative application condition (NAC) *is a tuple* $NAC = (N, n)$ *with* $n : L \to N$ *and* n *being injective. If* $\neg\exists q : N \to G$ *such that* $q \circ n = m$*, then* m satisfies NAC*, written* $m \models NAC$.

We also write $p = (L, R, r, \mathcal{N})$, where $\mathcal{N}$ is a set of NACs, when we want to explicitly refer to the NACs of a graph transformation rule p. Now we have everything we need for the definition of graph transformation systems.

- Intuitively, an NAC depicts a pattern which will prevent the rule applied to a graph if the pattern can be matched by the graph, even if the LHS is matched by the graph.
- Without NAC, a rule can only describe what is "required" in the system for triggering the action, but cannot tell what is "forbidden".

Some Definitions (4)

Definition 4 (Graph Transformation System)**.** *A* graph transformation system $S = (\mathscr{R}, G_0)$ *consists of a set of graph transformation rules* $\mathscr{R}$ *and an initial graph* G_0.

A planning problem on a graph transformation system also involves a graph pattern, which defines valid target configurations. The specification of graph patterns also support NACs.

Definition 5 (Graph Pattern)**.** *A* graph pattern $P = (L, \mathscr{N})$ *consists of a graph* L *and a set of NACs* $\mathscr{N}$ *where each* $NAC \in \mathscr{N}$ *is a tuple* $NAC = (N, n)$ *with* $n : L \to N$ *and* n *being injective.*

Having a means of specifying valid target configurations of a planning problem, we can now define the planning problem on a graph transformation system.

Definition 6 (Planning Problem)**.** *A* graph transformation planning problem $\mathscr{P} = (\mathscr{R}, G_0, P_{tgt})$ *consists of a set of graph transformation rules* $\mathscr{R}$*, an initial graph* G_0*, and a target graph pattern* $P_{tgt} = (L_{tgt}, \mathscr{N}_{tgt})$*. A* plan *for* $\mathscr{P}$ *is a sequence of direct derivations* $G_0 \Rightarrow \ldots \Rightarrow G_k$ *such that the target graph pattern* P_{tgt} *has a match in* G_k.

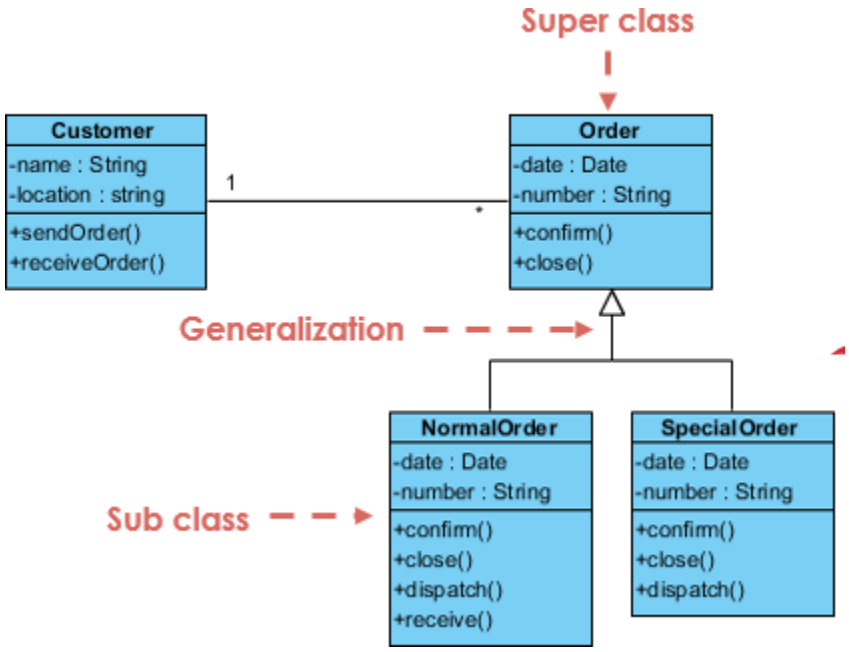
- The planning problem of GTS is domain-independent

UML

- General graphs contain too little information for us to depict problems, while
- UML provides us with several powerful graph-based modeling tools.
- *The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system*

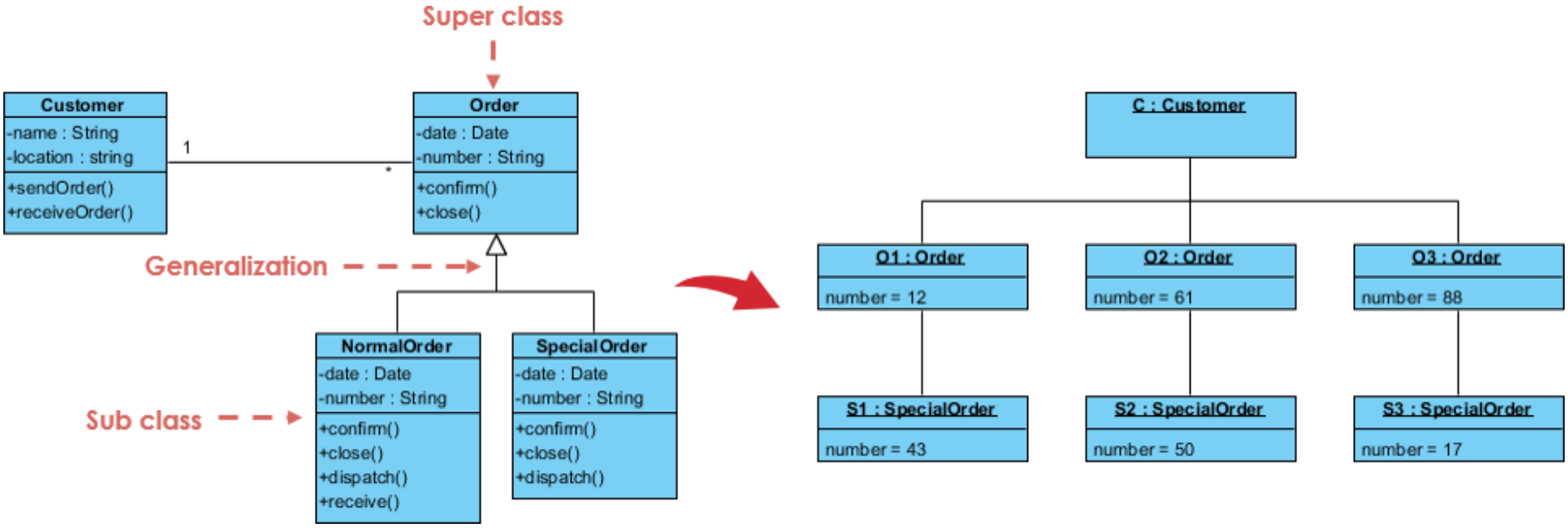
Class Diagram

- *A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.*



Object Diagram

- *In the Unified Modeling Language (UML), an object diagram focuses on some particular set of objects and attributes, and the links between these instances.*



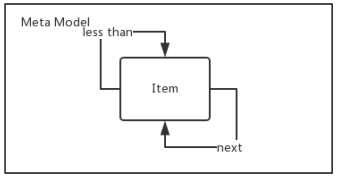
Simplification

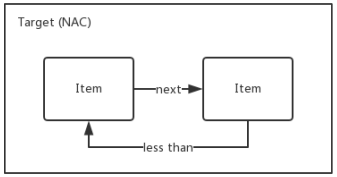
- No attributes & operations for either class diagram or object diagram
- No relation types for either class diagram or object diagram
- Class with label as name
- Relation with label as name
- Object with label of name and type

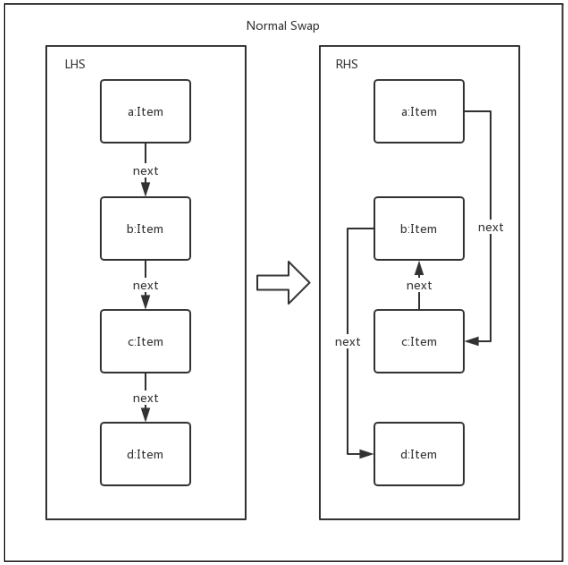
Problem definition

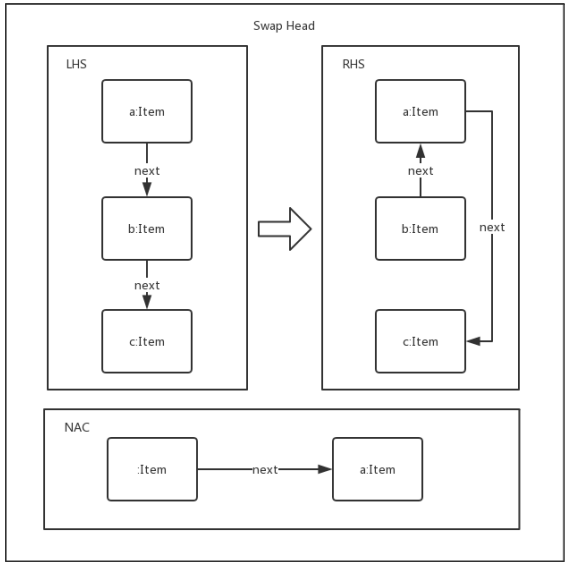
- A problem $P = (cd, R, g)$ where:
  - *cd* is a class diagram
  - *R* is a set of transformation rules
  - *g* is a goal pattern
- The input of a problem is an object diagram instantiated by the class diagram
- For any input of any given problem, the system is supposed to find out a solution:
  - transformation sequence
  - “unreachable” if the goal is unreachable

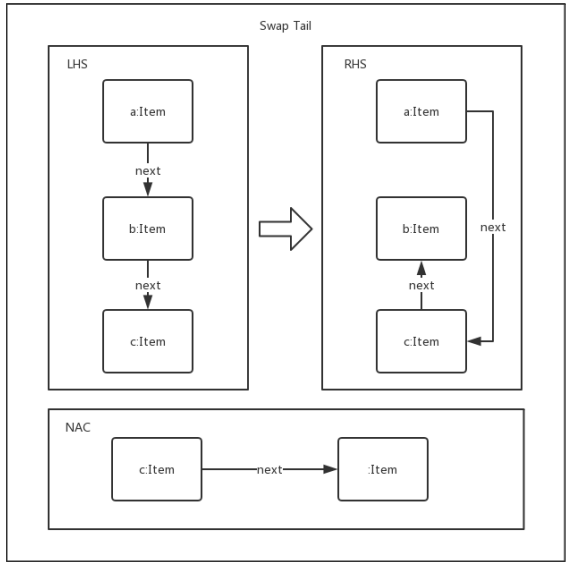
Example: Sorting Problem

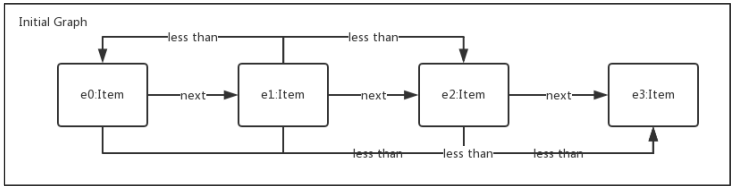












Three Steps to Go

- Implement the graph transformation
- Define a problem and several test cases per group
  - The problem can be from an algorithm practice, or any realistic problems
  - There should be moderate complexity for the problem
  - Different test cases should have different scale
  - All the problems & test cases make up the benchmark for the planning algorithms
- Design a domain-independent planning algorithm for GTS, reducing the time cost of the planning
  - No domain knowledge of any specific problem can be used
  - Cost of time can be measured by times of rules application
  - Only planning time is concerned, which means for any problem given, you can somehow make the system (but not yourself) analyze it and do some preparation regardless of the cost
  - Heuristic search (A* typically) is a common choice of optimization (but how?)