Distributed Hierarchical Hyper-TGraphs: Modeling beyond plain graphs

Daniel Bildhauer, Jürgen Ebert
Institute for Software Technology
University of Koblenz-Landau, Germany
dbildh@uni-koblenz.de
jgralab.uni-koblenz.de
Overview

• Motivation ➔ Why?
• DHHTGraphs ➔ What?
• Details ➔ How?
• Implementation ➔ Does it work?
• Conclusion ➔ Current state and what comes next?
Why?
Simple graph representation

- Symbols are represented by nodes
- Lines connecting symbols are represented by edges
- But: Some symbols are relation-like and some lines belong together
Issues of graph representation

- Models contain ad modeling profits from:
  - N-ary relationships
  - Abstraction and refinement
  - Distribution
- Graphs are a suitable representation of models
- Above concepts are not supported by plain graphs
- Workarounds & existing solutions not sufficient
Workarounds & existing solutions

• Workarounds cause extra effort to handle graphs
  • Relation-like vertices simulate n-ary relations
  • Marking of elements simulates abstraction levels

• Existing extended graph concepts are not sufficient
  • Many variants of hyperedges, hierarchy and distribution, but no integrated solution
  • Graph model too restricted or too complex
  • Not fully implemented/no convenient API
What?
Demands to a modern graph framework

- Seamless integration of distribution, hierarchy and hyperedges with established graph concepts
- Precise and well-defined graph formalism
- Ability to specify domain-specific aspects & constraints by graph schemas
- Efficient implementation to handle large graphs
- Seamless integration in modern software by API
Proposal: DHHTGraphs

- Distribution of graphs over networks
- Hierarchical structuring and refinement
  - Refinement of elements by nested graphs
  - Refinement of graphs by visibility layers
- Hyperedges with labeled directed ends
- Typing and attribution of vertices and edges
- Ordering of incidences at vertices and edges
- Compatible to existing concepts as far as possible
How?
Hypergraphs

- Typing, attribution and ordering of vertices and edges
- Connection by labeled, directed & ordered incidences traversable in both directions
- Equality and duality of vertices and edges
- Vertices represent entities, edges their relationships
Metamodeling Hypergraphs

- Modeling language grUML (graph UML)
- Vertex- and Edge classes define types and attributes
- Incidence classes define labeled connections
- Specialization of vertex-, edge-, and incidence classes
- Multiplicities and incidence inheritance for vertex- and edge classes
Hierarchical graphs

- Element refinement by tree-like nesting of graphs in elements
- Connections across boundaries
- Border of nested graphs are of the same kind as nested elements
- Graph refinement by visibility layers for elements
- Refinements are DHHTGraphs on their own

![Diagram showing hierarchical graphs with labelled nodes and links]
Metamodeling hierarchy

- Compositions define possible nesting relationships
- Tree-like on instance level
- Stereotyped compositions define edge nesting
- Compositions also define edge classes (compatibility to classical graph technology)
- Allowed visibility indicated by constraints (kappa)
Distributed graphs

- Partitioning and distribution across several stations
- Treatment of local and global graphs in the same way
- Compatibility to hierarchy
- Distributed graphs are full DHHTGraphs with support of distribution and hierarchy
- No domain specific features, thus no metamodeling
Does it work?
Implementation

- Extension of Java library JGraLab for plain TGraphs
- Typing, attribution... realized by native Java constructs
- Extended symmetric incidence lists as datastructure
- Distribution by Java Remote Method Invocation, efficient access by element-ids identify machine
- In-memory storage
  - 1GB: $10^6$ vertices, $10^6$ edges, $5 \times 10^6$ incidences (creation 7s)
  - Breadth first search on that graph in 2.5s on 2.3GHz
**API**

- Object-oriented access to all DHHTGraph properties
- Equal treatment of complete graphs and all subgraphs
- Seamless integration in applications by generated schema-specific API (interface+implementation)

```java
public interface TraceabilityLink extends Edge {
    public int get_id();
    public void set_id(int _id);
    public TraceabilityLink getNextTraceabilityLink();
    public TraceabilityLink_source getFirst_source();
    public TraceabilityLink_rule getFirst_rule();
}

public interface FeatureTraceabilityLink extends TraceabilityLink {
    public FeatureTraceabilityLink getNextFeatureTraceabilityLink();
    public FeatureTraceabilityLink_process getFirst_process();
    public FeatureTraceabilityLink_activity getFirst_activity();
}
```
Current state & what comes next?
Main design decisions

• Vertices and edges are dual
  • Vertices represent entities, edges their relations
• Distribution and hierarchy are compatible
  • Subgraphs are DHHTGraphs on their own
• Nested vertices and edges have only vertices or edges on their border, respectively
• Seamless from definition to implementation
The future: Querying DHHTGraphs

- Based on existing TGraph query language GReQL

```gsql
from process:V{Process}
with process (-->{FeatureTraceabilityLink_process}
  <--{TraceabilityLink_rule})+
  & {TransformationRule @
    thisVertex.name="MyRule"}
report process -->{FeatureTraceabilityLink} end
```
Conclusion

• Seamless realization of
  • Hyperedges with labeled ends
  • Nesting of graphs in elements
  • Abstraction levels by visibility layers
• Distribution
• Based on formal mathematical definition
• Metamodeling of domain-specific aspects
• Efficient implementation & convenient APIs
• Available soon at jgralab.uni-koblenz.de under GPL
Thanks for your attention!
Any questions?