Jürgen Ebert
University Koblenz-Landau
http://www.uni-koblenz.de/~ebert

THE GRAPH QUERY LANGUAGE GREQL
The TGraph approach supports conceptual modeling with class diagrams and efficient implementation with graphs – seamlessly.

The query language GReQL is used to extract information from TGraphs.
TABLE OF CONTENTS

- Introduction to graph-based modeling
- Querying using GReQL
- Conclusion
MOST = Marrying Ontologies and Software Technology

EU-STREP Grant ICT 216691 (2008-2020)

Goals of MOST:

- a seamless integration technology for ontologies into model-driven software development (MDSD), resulting in ontology driven software development (ODSD).
Several approaches for ontology technology are being used with languages, like RDF, RDF-Schema, OWL-light, OWL-DL, OWL-Full, ...

Light-weight conceptual modeling can also be done by class diagrams and graphs.

Here, we use a subset of UML (grUML) and a suitable model category (TGraphs) for conceptual modeling
```java
public class Main {
    public static void main(String[] args) {
        int a = 26;
        int b = -5;
        System.out.println(compute(a, b));
    }

    private static int compute(int a, int b) {
        return Utility.twice(Utility.add(a, b));
    }
}

public class Utility {
    public static int add(int x, int y) {
        return x + y;
    }

    public static int twice(int x) {
        return 2 * x;
    }
}
```
public static void main(String[] args) {
    int a = 26;
    int b = -5;
    System.out.println(compute(a, b));
}
The instantiation relationship gives rise to a n-level instantiation hierarchy.

Precise Definition
+ of possible instances (TGraphs) and
+ of legal schemas (grUML).
TGRAPHS

TGraphs are
+ typed (supporting multiple inheritance):
  vertices and edges have a type
+ attributed (depending on the type):
  vertices and edges have valued attributes
+ ordered:
  vertices, edges, and incidences are ordered
+ directed:
  edges have a start and an end vertex
PROPERTIES OF TGRAPHS

TGraphs have powerful properties
+ Edges are first class citizens
+ Traversal is supported in both directions
+ Graphs as a whole are subject to algorithms
+ ...
+ Entities are modeled by vertices
+ Occurrences are modeled by edges
+ Sequences are expressed by edge order
TGraphs can be specified by a subset of UML-class diagrams (grUML):

- **Classes**: Vertex Types
- **Associations**: Edge Types
- **Attributes**: Element Attributes
- **Specialization**: Type Inheritance
- **Multiplicities**: Degree Restrictions (slightly more than EMOF)
A TGraph is compatible to a schema
- if the element types and the attribute assignments in the graph respect the schema,
- if the incidences of the edges respect the schema, and
- if the vertex degrees respect the multiplicities (under inheritance)
The `isInstanceOf` relation is precisely defined

The `isAbstractionOf` relation is formalized by

- Extractors (Parsers, Scanners) that generate the graph from the artifact
- Editors that keep the graph as the internal model of the artifact
- Renderers that generate the artifact from the graph
The approach is implemented in Java on top of the JGraLab class library – a highly efficient API for creating, manipulating, and traversing TGraphs (wrt their schema).

The representation is biased towards making traversal as efficient as possible. (symmetric incidence lists)

http://jgralab.uni-koblenz.de/
QUERYING
GReQL (Graph Repository Query Language) is a schema sensitive graph query language.
The goal of querying is to extract information from graph-based models and to make the extracted information available in software tools.

<table>
<thead>
<tr>
<th>Problem: Graph Querying</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> A graph $g$ and a query text $q$</td>
</tr>
<tr>
<td><strong>Output:</strong> Evaluation of $q$ on $g$</td>
</tr>
</tbody>
</table>
from caller, callee: V{Method}
with caller (  
    \=\{isStatementIn\}  
    [ \=\{isReturnValueOf\} ]  
    \=\{isActualParameterOf\}  
    \=\{isCalleeOf\}  
)\{callee\}
report  
caller.name as "Caller",  
callee.name as "Callee"  
end

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>System.out.println</td>
</tr>
<tr>
<td>main</td>
<td>compute</td>
</tr>
<tr>
<td>main</td>
<td>twice</td>
</tr>
<tr>
<td>add</td>
<td></td>
</tr>
<tr>
<td>compute</td>
<td>twice</td>
</tr>
<tr>
<td>add</td>
<td></td>
</tr>
</tbody>
</table>
PROPERTIES OF GReQL

- expression language
- schema sensitive
- dynamically typed
- mathematical semantics

There is a full optimizing evaluator for GReQL available on JGraLab.
To support all types used in grUML, a special union type (JValue) has been defined. Using the JValue-API the results of GReQL-queries can be used in arbitrary Java software.
**Expressions**

Expression ::= 
  Literal 
  | Variable 
  | ValueConstruction 
  | ValueAccess 
  | '(' Expression ')' 
  | FunctionApplication 
  | UnaryExpression 
  | BinaryExpression 
  | ConditionalExpression 
  | RestrictedExpression 

  | QuantifiedExpression 
  | ElementSetExpression 
  | SubgraphExpression 
  | LetExpression 
  | WhereExpression 
  | SimpleQuery 
  | PathDescription 
  | CfGrammar 
  | ForwardVertexSet 
  | BackwardVertexSet 
  | PathExistence
Since GReQL is heavily based on set-like expressions, also quantified expressions can be used using `forall`, `exists`, and `exists!`

```plaintext
forall v:V{Method}, v.name=~"^set" @ outDegree{isCalleeOf}=1
forall e:E{isCalleeOf} @ alpha(e).name=~"^set"
```
The available operations are stored in an extensible function library.

Examples:
... besides booleans, numbers, strings ...
alpha(), omega(), attributes(),
avg(), classes(), concat(),
connects(), cnt(), degree(), ...
The expression intended to extract data from the given graph is the \texttt{from-with-report} expression.

\begin{verbatim}
from <DeclarationList>    (1)
with <BooleanExpression> (2)
report <ExpressionList>  (3)
end
\end{verbatim}
The result is a bag of values or tuples of values (depending on the length of the report list) containing the results of the expressions in (3) evaluated for each variable binding in (1) which fulfills (2).
from v, w: V{Method}
with v -->{isCalleeOf}-->{isStatementIn} w
report w.name as "caller", v.name as "callee"
The most powerful expressions are built using regular path expressions (RPEs).

They describe (sets of) paths through a graph based on edge types (including inheritance) and edge direction.
PATH EXPRESSIONS

RPEs consist of

+ SimplePathDescriptions \((\text{--\text{--}},<\text{--},<\text{--}),\)possibly extended by type descriptions
+ SequentialPathExpressions \((p_1\ p_2)\)
+ AlternativePathExpressions \((p_1\mid\ p_2)\)
+ IteratedPathDescription \(p_1^*\)
Path expressions can be applied to vertices:

- \( v \rightarrow^* \) set of vertices reachable from \( v \)
- \( \rightarrow^* w \) set of vertices from which \( w \) is reachable
- \( v \rightarrow^* w \) decision whether \( w \) is reachable from \( v \)
from caller, callee: V{Method}
with caller(
    <--{isStatementIn}
    [ <--{isReturnValueOf} ]
    <--{isActualParameterOf}*
    <--{isCalleeOf}
    )+ callee

report
caller.name as "Caller",
callee.name as "Callee"

end

<table>
<thead>
<tr>
<th>Caller</th>
<th>Callee</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>System.out.println</td>
</tr>
<tr>
<td>main</td>
<td>compute</td>
</tr>
<tr>
<td>main</td>
<td>twice</td>
</tr>
<tr>
<td>main</td>
<td>add</td>
</tr>
<tr>
<td>compute</td>
<td></td>
</tr>
<tr>
<td>compute</td>
<td>twice</td>
</tr>
<tr>
<td>add</td>
<td></td>
</tr>
</tbody>
</table>
Path expressions are evaluated by searching the graph.

```java
void dfs (Vertex v) {
    v.setMark();
    // process vertex v
    for (Edge e: g.getAllOutEdges (v)) {
        w = e.omega();
        if (! w.isMarked()) {
            // process tree edge e
            v.setParent(e);
        }
    }
}
```
The regular path expressions are transformed to a deterministic finite automaton (DFA) using Thompson’s construction and the Myhill algorithm. Given the automaton, the reached vertices are marked by the states of the automaton.
A by-product of the search algorithms is the computation of the predecessor for each (non-root) vertex, which can easily be stored in a (temporary) parent-attribute. This hold also for the RPE-search.

Thus, given an expression like $v \rightarrow* w$, a path from $v$ can be returned for each $w$ (via backtracing over the parent-entries).
Since it is well-known that the size of the automaton is linear in the size of the expression in practice

the complexity of a path expression of size $k$ is $O(mk)$ for a connected graph.
CONCLUSION
The TGraph approach supports conceptual modeling with class diagrams and efficient implementation with graphs - in a seamless manner.

The query language GReQL can be used to extract information from TGraphs - in practice in polynomial time.
Thank you