



Bridging Query Languages in Semantic and Graph Technologies

Hannes Schwarz, Jürgen Ebert

Institute for Software Technology
University of Koblenz-Landau

3 September 2010





Contents

- **Introduction and Motivation**
- **Basics of the Bridging Approach**
- **Bridging RDF and TGraphs (incl. exercise)**
- **Bridging SPARQL and GReQL (incl. exercise)**
- **Applications**
- **Summary and Outlook**



Contents

- **Introduction and Motivation**
- Basics of the Bridging Approach
- Bridging RDF and TGraphs (incl. exercise)
- Bridging SPARQL and GReQL (incl. exercise)
- Applications
- Summary and Outlook



Introduction and Motivation

Technological Spaces



Different Technologies in Software Engineering

- Software systems and software development itself make use of various technologies, e.g.:
 - ♦ programming languages
 - ♦ XML technology
 - ♦ relational database systems
 - ♦ semantic technologies (**Ontoware**)
 - ♦ modeling technologies (**Modelware**)
- Different technologies or families of languages can be referred to as **technological spaces**.



Introduction and Motivation

Technological Spaces

The Notion of Technological Spaces

- Definition of a technological space:

“[...] working context with a set of associated knowledge, tools, required skills, and possibilities.”

[Kurtev, Bézivin, Aksit: Technological Spaces: an Initial Appraisal, 2002]

- The notion of a technological space is not standardized and there is no common agreement on the categorization of spaces.
- It is a rather informal concept which eases communication.



Introduction and Motivation Technological Spaces



Ontoware in Software Engineering

- **Ontoware** refers to logic-based approaches to modeling (i.e. ontologies).
- The Ontoware technological space can be subdivided into subspaces, e.g.:
 - ◆ F-Logic
 - ◆ OWL
 - ◆ RDF



Introduction and Motivation

Technological Spaces



Ontoware in Software Engineering (cont'd)

- A current trend is to try to solve software engineering problems by applying Ontoware services, e.g.
 - ♦ Consistency checking
 - ♦ Satisfiability checking
 - ♦ Subsumption
 - ♦ Classification
- Possible applications (see MOST project):
 - ♦ Guidance for software development processes
 - ♦ Guiding and validating network devices configuration
 - ♦ Guiding and validating process model refinements



Introduction and Motivation

Technological Spaces



Modelware in Software Engineering

- **Modelware** refers to “UML-like” modeling approaches and associated languages (e.g. for constraints, querying, and transformations).
- The Modelware technological space can be subdivided into subspaces, e.g.:
 - ◆ MOF
 - ◆ EMF
 - ◆ TGraphs
- Modelware is widely used in software engineering.



Introduction and Motivation

Bridging



Overview

- Concerted usage of different technological spaces in a common context requires their **bridging**.
- The term “bridging” refers to making it possible to apply **services** and **tools** of one technological space to artifacts of another one.
- There exist different **types** of bridges between technological spaces.



Introduction and Motivation

Bridging



Types of Bridges

- Adapter
 - ◆ From the point of view of the technological space whose services and tools are to be applied, special **interfaces** are developed to treat “foreign” artifacts as native.
- Transformation
 - ◆ Artifacts of different technological spaces are **transformed to equivalent artifacts** native to the technological space whose services and tools are to be applied.
- Integration
 - ◆ Two technological spaces are **merged** to a single one.



Introduction and Motivation

Bridging

Mapping of Technological Spaces' Concepts

- All bridging approaches require the **mapping** of the concepts provided by the concerned technological spaces.
- Example mapping (for illustration only):

OWL	MOF
Class	Class
Individual	Instance
Object Property	Association
Data Property	Attribute
...	...



Introduction and Motivation

Scope of this Lecture



Concerned Technological Spaces

- **Transformation-based bridging** approach between modeling and associated query languages of Ontoware and Modelware.
- Chosen representatives:
 - ♦ **RDF** and **SPARQL** (well-known Ontoware languages)
 - ♦ **TGraphs** and **GReQL** (no standards comparable to MOF or EMF together with OCL, but more expressive)
- Since the syntax of OWL can be serialized to RDF, it is also covered by the bridging approach.



Introduction and Motivation

Scope of this Lecture



Restrictions of the Bridging Approach

- The transformation between RDF and TGraphs only considers the conversion of the **data structures**, i.e. **without** preservation of the underlying **semantics**.
- The most important difference is the adoption of the **open-world assumption** by RDF, while TGraphs are based on the **closed-world assumption**.
- Since SPARQL (1.0) adopts the closed-world assumption, the **differences in semantics are irrelevant** for the bridging between query languages.



Introduction and Motivation

Scope of this Lecture

Expected Benefits

- When transforming artifacts between Ontoware and Modelware in a **Ontology-driven Software Development** scenario (as advocated by the MOST project), **queries** on these artifacts have to be **formulated only once**.
- GReQL with its expressive **path descriptions** can be used to query RDF, involving the **computation of transitive closure** which is currently not supported by SPARQL.
- GReQL queries are **more concise** than their SPARQL counterparts.



Contents

- Introduction and Motivation
- **Basics of the Bridging Approach**
- Bridging RDF and TGraphs (incl. exercise)
- Bridging SPARQL and GReQL (incl. exercise)
- Applications
- Summary and Outlook



Idea of the Bridging Approach Mapping



- Prior to the development of the actual transformations between RDF and TGraphs, and SPARQL and GReQL, the concepts of the respective languages have to be **mapped** to each other.
- The structure of RDF documents and TGraphs **can** be quite different, although without necessarily being so.
- Different transformations can be applied, depending on whether an RDF document is “TGraph-like” or not.
- Transformation of non-TGraph-like RDF documents results in TGraphs with an awkward modeling style.



Idea of the Bridging Approach Transformations

- **Schema-aware** transformation between **RDF** and **TGraphs** (for “TGraph-like” RDF documents only, bidirectional)
- **Simple** transformation from **RDF** to **TGraphs** (for any RDF document, unidirectional)
- **Schema-aware** transformation between **SPARQL** and **GReQL** (if schema-aware transformation of queried data was used)
- **Simple** transformation from **RDF** to **TGraphs** (if simple transformation of queried data was used)



Contents

- Introduction and Motivation
- Basics of the Bridging Approach
- **Bridging RDF and TGraphs (incl. exercise)**
- Bridging SPARQL and GReQL (incl. exercise)
- Applications
- Summary and Outlook



Bridging RDF and TGraphs

Introduction to RDF



General Information

- **Resource Description Framework**, issued by the World Wide Web Consortium (W3C)
- RDF is a standard for **describing and structuring information** on the WWW.
- RDF incorporates the **XML Schema datatypes**
- RDF is extended by **RDF Schema (RDFS)**.

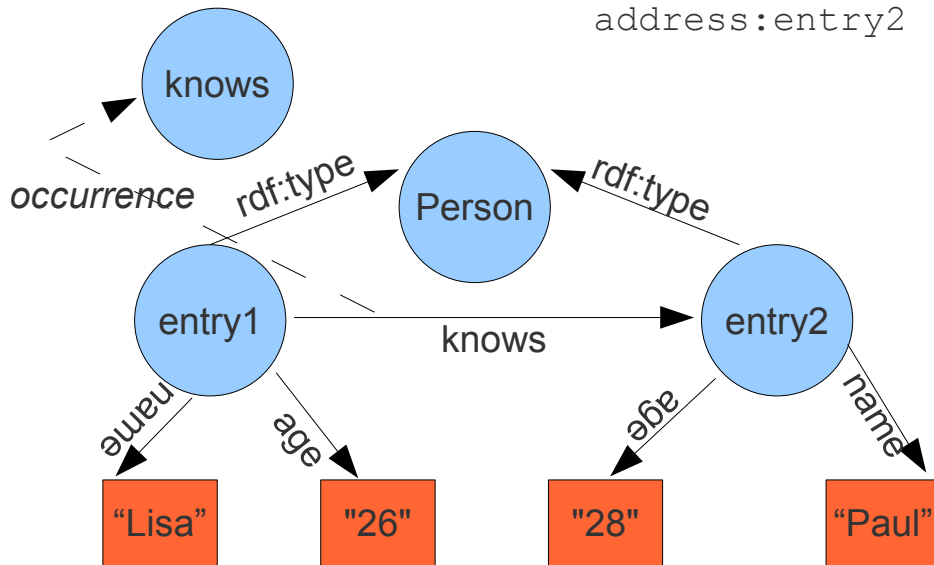


Bridging RDF and TGraphs

Introduction to RDF

Basic RDF Language Concepts

```
address:entry1 rdf:type address:Person
address:entry1 address:name "Lisa"
address:entry1 address:age "26"
address:entry2 rdf:type address:Person
address:entry2 address:name "Paul"
address:entry2 address:age "28"
address:entry1 address:knows
address:entry2
```



- RDF documents consist of **triples (subject, predicate, object)**.
- Subjects and objects are **resources, blank nodes, or literals** (data values).
- Predicates are **occurrences of properties**.
- RDF documents can be represented as **graphs**, with **nodes** denoting subjects and objects, and **arcs** denoting predicates.



Bridging RDF and TGraphs

Introduction to RDF



Some important RDF Characteristics

- Literals are **plain**, i.e. a simple string, or **typed**.
- All properties are an instance of the predefined resource `rdf:Property`.
- The predefined property `rdf:type` denotes instance-of relationships.
- `rdf:type` chains can be of any length.
- Resources can be instance of more than one other resource.
- Properties may act as subjects or objects in triples.



Bridging RDF and TGraphs

Introduction to RDF

RDF Schema Concepts

```
address:entry1 rdf:type address:Person
address:entry1 address:name "Lisa"
address:entry1 address:age "26"
address:entry2 rdf:type address:Person
address:entry2 address:name "Paul"
address:entry2 address:age "28"
address:entry1 address:knows
                    address:entry2
```

```
address:Person rdf:type rdfs:Class
address:name rdfs:domain address:Person
address:name rdfs:range rdf:PlainLiteral
address:age rdfs:domain address:Person
address:age rdfs:range xsd:int
address:knows rdfs:domain address:Person
address:knows rdfs:range address:Person
```

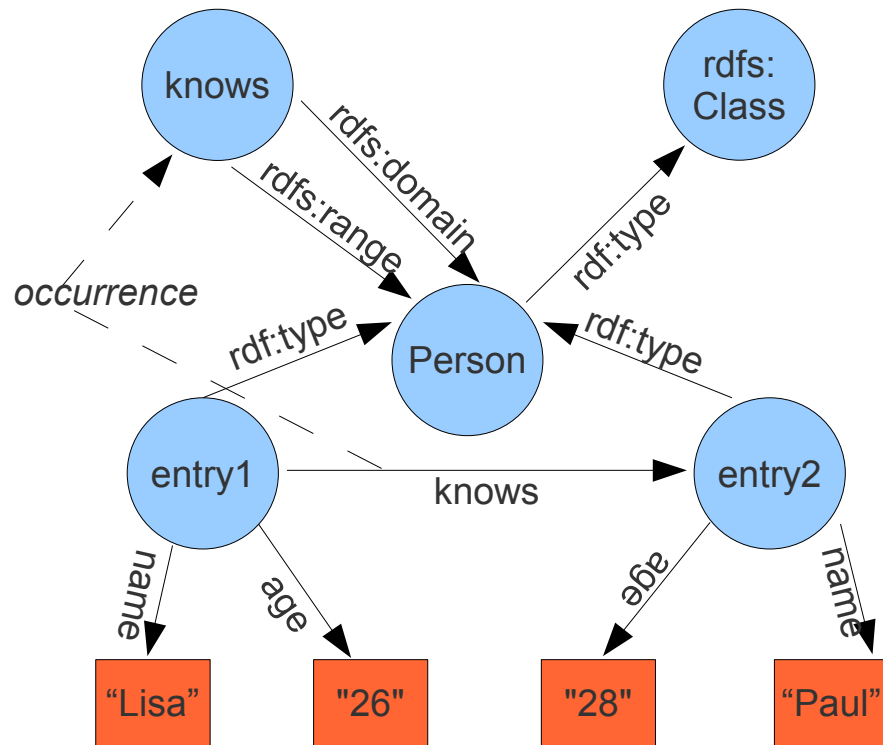
- RDF Schema **extends** the RDF vocabulary by predefined
 - ◆ resources such as `rdfs:Resource`, and `rdfs:Class`
 - ◆ properties such as `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, and `rdfs:range`



Bridging RDF and TGraphs

Introduction to RDF

RDF Schema Concepts (cont'd)



- RDF Schema **extends** the RDF vocabulary by predefined
 - ♦ resources such as `rdfs:Resource`, and `rdfs:Class`
 - ♦ properties such as `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, and `rdfs:range`



Bridging RDF and TGraphs

Introduction to RDF

Datatypes

- Besides the two datatypes provided by RDF, `rdf:PlainLiteral` (for plain literals) and `rdf:XMLLiteral`, the datatypes of XML Schema are adopted.

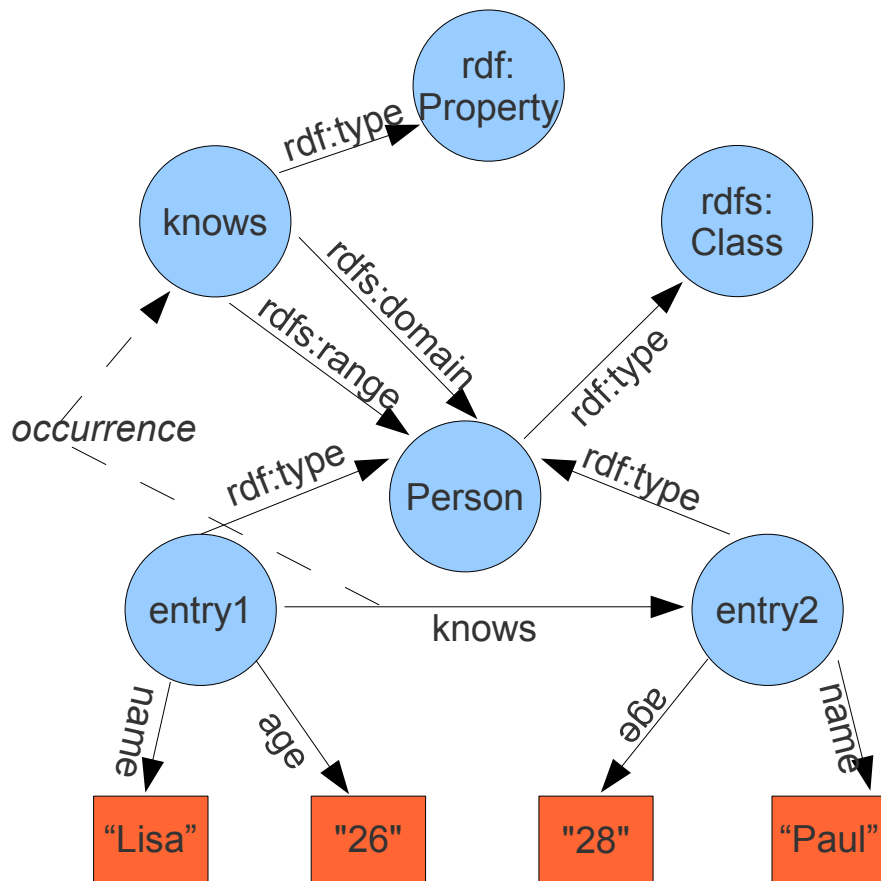
RDF datatype	Description	RDF datatype	Description
<code>rdf:PlainLiteral</code>	a plain literal	<code>xsd:gDay</code>	a day recurring every month in every year
<code>rdf:XMLLiteral</code>	XML content	<code>xsd:gMonth</code>	a month recurring every year
<code>xsd:anyURI</code>	a URI reference	<code>xsd:gMonthDay</code>	a day in a specific month recurring every year
<code>xsd:base64Binary</code>	Base64-encoded binary data	<code>xsd:gYear</code>	a calendar year
<code>xsd:boolean</code>	a boolean value	<code>xsd:gYearMonth</code>	a month in a specific year
<code>xsd:date</code>	a calendar date	<code>xsd:hexBinary</code>	hex-encoded binary data
<code>xsd:dateTime</code>	a point in time	<code>xsd:int</code>	a 32-bit signed integer number
<code>xsd:decimal</code>	a decimal number of arbitrary precision	<code>xsd:long</code>	a 64-bit signed integer number
<code>xsd:double</code>	a 64-bit signed floating point number	<code>xsd:string</code>	a character string
<code>xsd:float</code>	a 32-bit signed floating point number	<code>xsd:time</code>	a point in time recurring every day



Bridging RDF and TGraphs

Introduction to RDF

Open World Assumption (OWA) and Entailment



- RDF features the OWA, i.e. not explicitly specified information is not necessarily non-existent.
- Entailment serves to infer new triples from existing ones.
- Entailment types (regimes):
 - ◆ Simple Entailment
 - ◆ RDF Entailment (see figure)
 - ◆ RDF Schema Entailment
 - ◆ D-Entailment



Bridging RDF and TGraphs

Introduction to TGraphs



General Information

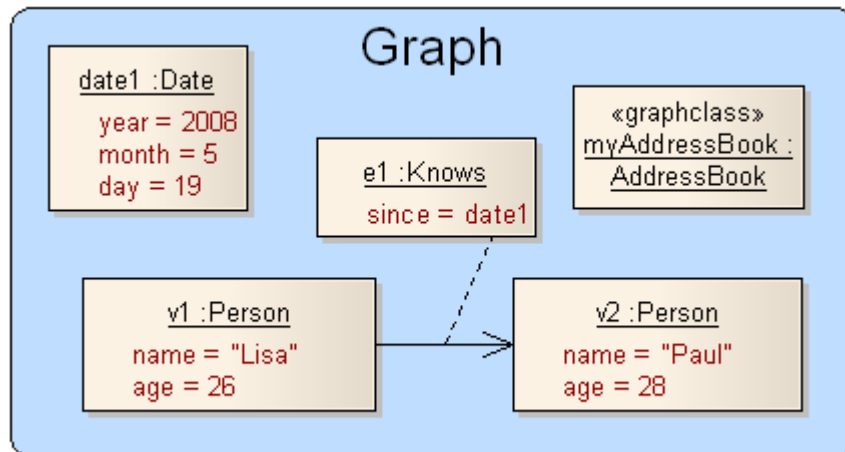
- TGraphs are a very general kind of graphs, developed by the Software Technology Group of the University of Koblenz-Landau.
- TGraphs have been applied for meta-case tools, reengineering, and the storage of traceability information, for instance.
- TGraphs are comparable to the standard modeling approaches EMOF and EMF, but are more expressive.



Bridging RDF and TGraphs

Introduction to TGraphs

Basic TGraph Concepts



- TGraphs are graphs whose vertices and edges (and the graph itself) are **typed** and **attributed**.
- Edges are **directed**.
- Vertices and edges are globally **ordered**. Further, the incident edges of a vertex are ordered.



Bridging RDF and TGraphs

Introduction to TGraphs

Mathematical Definition

Let

- *Vertex* be the universe of vertices,
- *Edge* be the universe of edges,
- *TypeId* be the universe of type identifiers,
- *AttrId* be the universe of attribute identifiers, and
- *Value* be the universe of attribute values.

Assuming two finite sets,

- a vertex set $V \subseteq \text{Vertex}$ and
- an edge set $E \subseteq \text{Edge}$,

be given. $G = (Vseq, Eseq, Aseq, type, value)$ is a *TGraph* iff

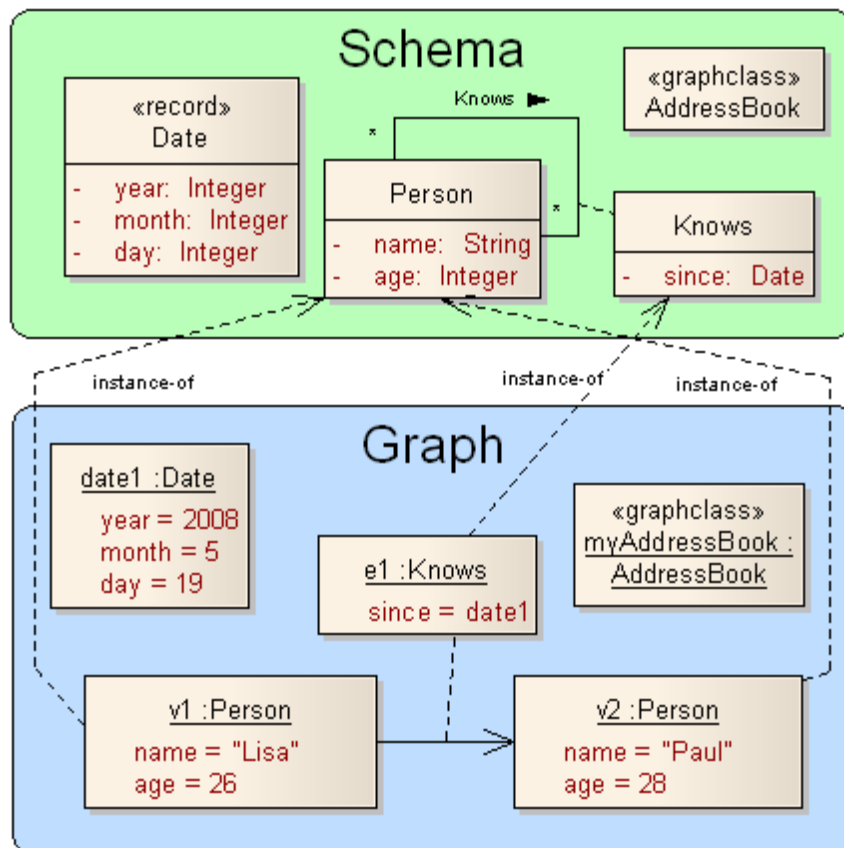
- $Vseq \in \text{iseq } V$ is a permutation of V ,
- $Eseq \in \text{iseq } E$ is a permutation of E ,
- $Aseq : V \rightarrow \text{iseq}(E \times \{in, out\})$ is an *incidence function* where
 $\forall e \in E : \exists! v, w \in V : (e, out) \in \text{ran } Aseq(v) \wedge (e, in) \in \text{ran } Aseq(w)$,
- $type : V \cup E \rightarrow \text{TypeId}$ is a *type function*, and
- $value : V \cup E \rightarrow (\text{AttrId} \multimap \text{Value})$ is an *attribute function* where
 $\forall x, y \in V \cup E : type(x) = type(y) \Rightarrow \text{dom}(value(x)) = \text{dom}(value(y))$.



Bridging RDF and TGraphs

Introduction to TGraphs

TGraph Schemas and grUML



- A TGraph conforms to a schema modeled in grUML (graph UML).
- A schema specifies:
 - ◆ the graph class,
 - ◆ vertex and edge classes,
 - ◆ their associated attributes,
 - ◆ generalizations between classes,
 - ◆ start and end vertex classes for edge classes,
 - ◆ multiplicity restrictions



Bridging RDF and TGraphs

Introduction to TGraphs

grUML Metaschema

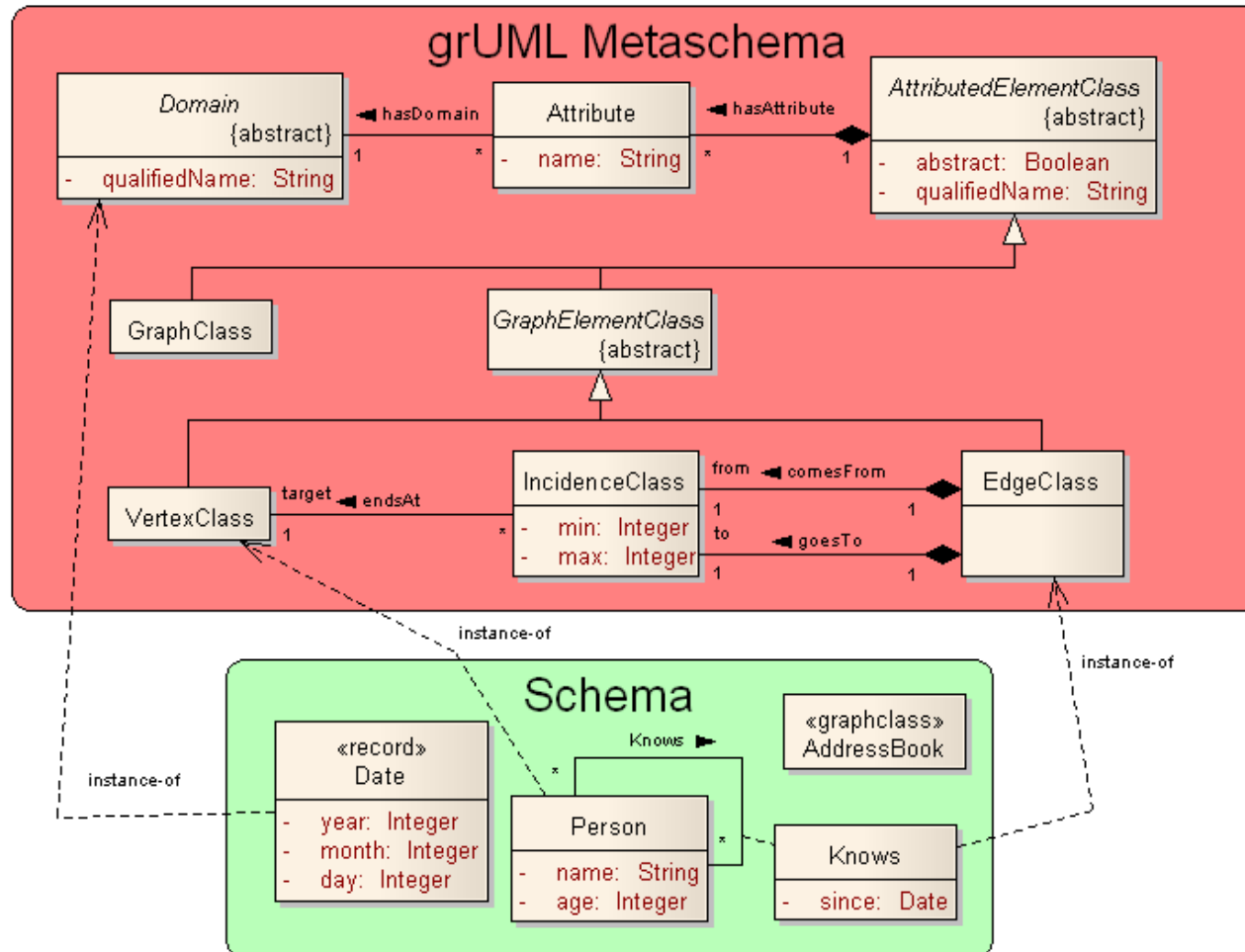
- A grUML schema again conforms to the predefined **grUML metaschema**.
- The grUML metaschema defines concepts (metaclasses) such as
 - ◆ GraphClass
 - ◆ VertexClass
 - ◆ EdgeClass
 - ◆ Attribute
 - ◆ Domain (for attributes)
 - ◆ ...



Bridging RDF and TGraphs

Introduction to TGraphs

grUML Metaschema (cont'd)





Bridging RDF and TGraphs

Introduction to TGraphs

grUML Metaschema Datatypes

- grUML specifies various domains for attributes, i.e. subclasses of the metaclass `Domain`:

grUML domain	Description
<code>Boolean</code>	a boolean value
<code>Integer</code>	a signed 32-bit integer number
<code>Long</code>	a signed 64-bit integer number
<code>Double</code>	a signed 64-bit floating point number
<code>String</code>	a string of any length
<code>List<BaseDomain></code>	an ordered sequence of values of the <code>BaseDomain</code>
<code>Set<BaseDomain></code>	an (unordered) set of values of the <code>BaseDomain</code>
<code>Map<KeyDomain, ValueDomain></code>	a mapping of values of the <code>KeyDomain</code> to values of the <code>ValueDomain</code>
<code>Enum</code>	a user-defined enumeration of possible attribute values
<code>Record</code>	a user-defined composition of any number of <i>identifier-domain</i> pairs



Bridging RDF and TGraphs

Introduction to TGraphs



Closed World Assumption (CWA)

- The TGraph approach is based on the CWA, i.e. information which is not modeled in a graph is assumed to not exist.



Bridging RDF and TGraphs

Introduction to TGraphs



Why TGraphs?

- TGraphs offer some distinctive **advantages** over more popular Modelware approaches such as (E)MOF and EMF:
 - ◆ edges may carry attributes
 - ◆ edges are first-class objects instead of anonymous references
 - ◆ TGraph elements are ordered



Bridging RDF and TGraphs

Mapping between RDF and TGraphs

Relevant Differences (selection, see paper)

RDF (is more “general”)

- Predicates are occurrences of properties (without identity)
- `rdf:type` chains can be of any length.
- Resources can be an instance of any number of classes.
- Properties may also act as subjects or objects.

TGraphs (are more “structured”)

- Edges are first-class objects with an identity.
- TGraphs have three meta-levels, i.e. “instance-of chains” are of length two.
- A graph element is an instance of exactly one schema class.
- Edges may not be incident to other edges.



Bridging RDF and TGraphs

Mapping between RDF and TGraphs

Types of Mapping

- **Schema-aware mapping** can be applied for RDF documents which are TGraph-like, e.g.
 - ♦ which only have `rdf:type` chains of length two,
 - ♦ whose resources are instance of exactly one `rdfs:Class`
 - ♦ whose resources (properties) do not occur as predicate and as subject or object
- Schema-aware mapping is the basis for a **bidirectional transformation** between RDF and TGraphs.
- **Simple mapping** can be used for transforming any RDF document to a TGraph (**unidirectional**).



Bridging RDF and TGraphs

Schema-aware Mapping

Main Language Concepts

RDF concept	TGraph concept	Comment
URI of the RDF document	Graph Class	
Resource		
instance of <code>rdfs:Class</code> (object in an <code>rdf:type</code> triple)	Vertex Class	URI corresponds to vertex class name
as subject in an <code>rdf:type</code> triple	Vertex	URI is stored in special attribute
as instance of <code>rdf:Property</code> (with <i>non-datatype range</i>)	Edge Class	URI corresponds to edge class name
as instance of <code>rdf:Property</code> (with <i>datatype range</i>)	Attribute	URI corresponds to attribute name
occurrence of property	Edge	
Literal	Attribute Value	
Blank Node	Vertex	<i>instance of a special Vertex Class</i>



Bridging RDF and TGraphs

Schema-aware Mapping

Datatypes – Domains

RDF datatype	grUML domain	RDF datatype	grUML domain
<code>rdf:PlainLiteral</code>	String	<code>xsd:gMonth</code>	Integer
<code>rdf:XMLLiteral</code>	String	<code>xsd:gMonthDay</code>	Record
<code>xsd:anyURI</code>	String	<code>xsd:gYear</code>	Record
<code>xsd:base64Binary</code>	String	<code>xsd:gYearMonth</code>	Record
<code>xsd:boolean</code>	Boolean	<code>xsd:hexBinary</code>	String
<code>xsd:date</code>	Record	<code>xsd:int</code>	Integer
<code>xsd:dateTime</code>	Record	<code>xsd:long</code>	Long
<code>xsd:decimal</code>	Double	<code>xsd:string</code>	Enum
<code>xsd:double</code>	Double	<code>xsd:string</code>	String
<code>xsd:float</code>	Double	<code>xsd:time</code>	Record
<code>xsd:gDay</code>	Integer		

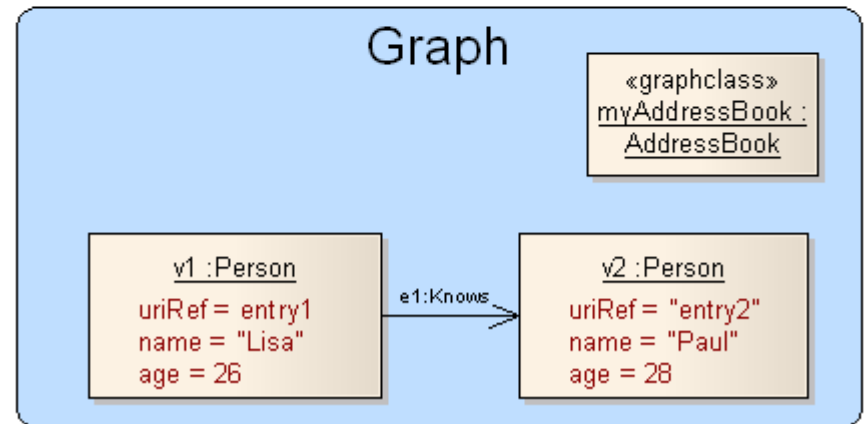
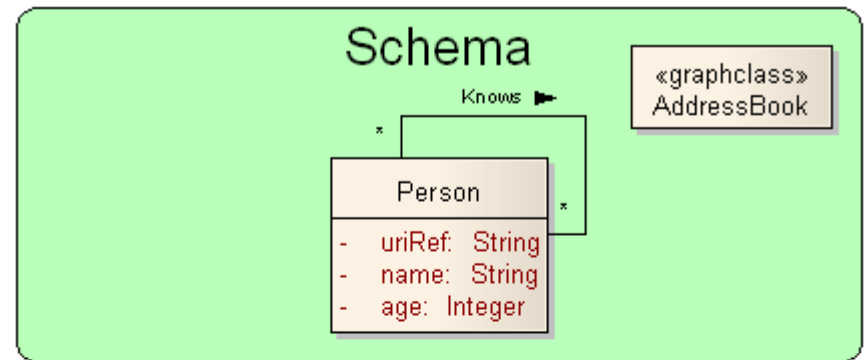
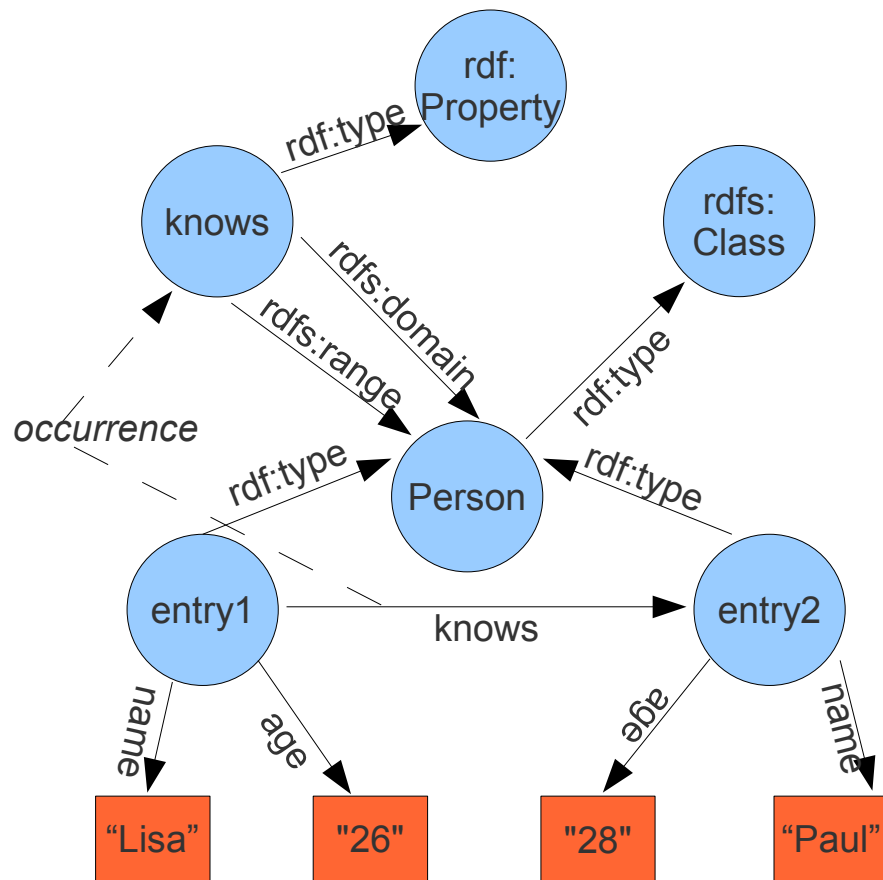
(the mapping of grUML List, Set, Map, Record domains is possible, but of scope here)



Bridging RDF and TGraphs

Schema-aware Mapping

Example Transformations (in both directions)





Bridging RDF and TGraphs

Simple Mapping

Basic Idea

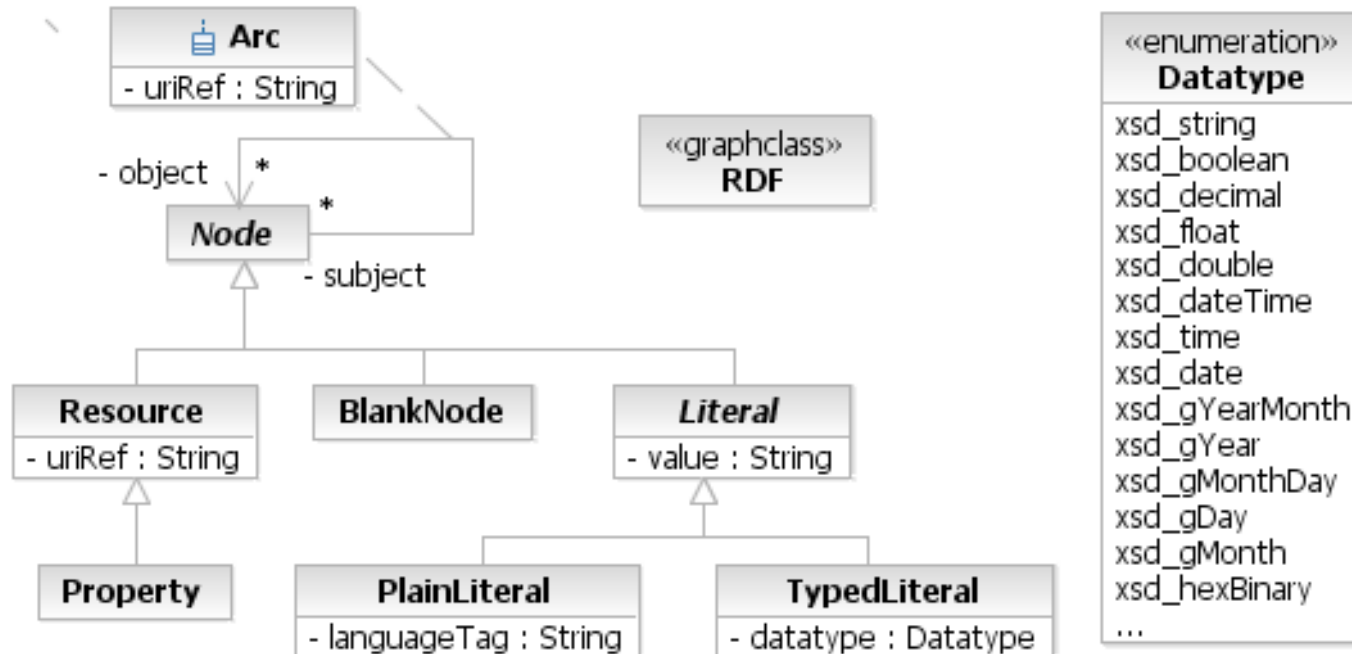
- The **type hierarchy** of an RDF document is “**flattened**” to a TGraph conforming to a fixed schema (RDF Schema).
- The RDF Schema reflects the language concepts of RDF.
- **Predicates** in the RDF document **are mapped to edges**, so they get an identity which can be exploited when posing queries.
- The resulting TGraph representation features an awkward, “**unnatural**” **modeling style**.



Bridging RDF and TGraphs

Simple Mapping

RDF Schema

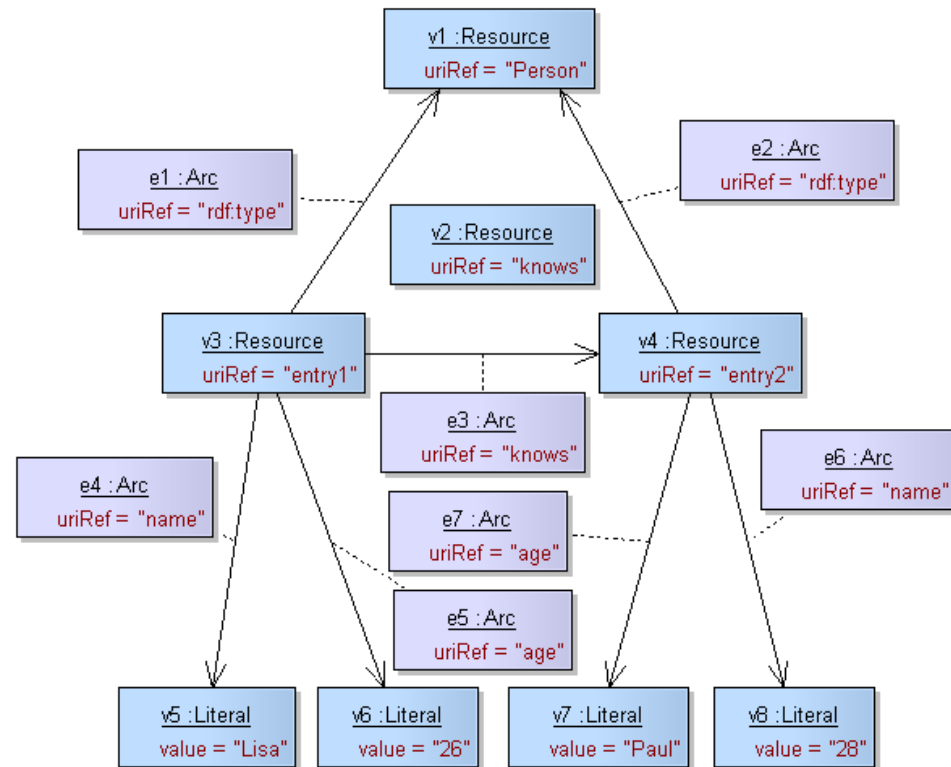
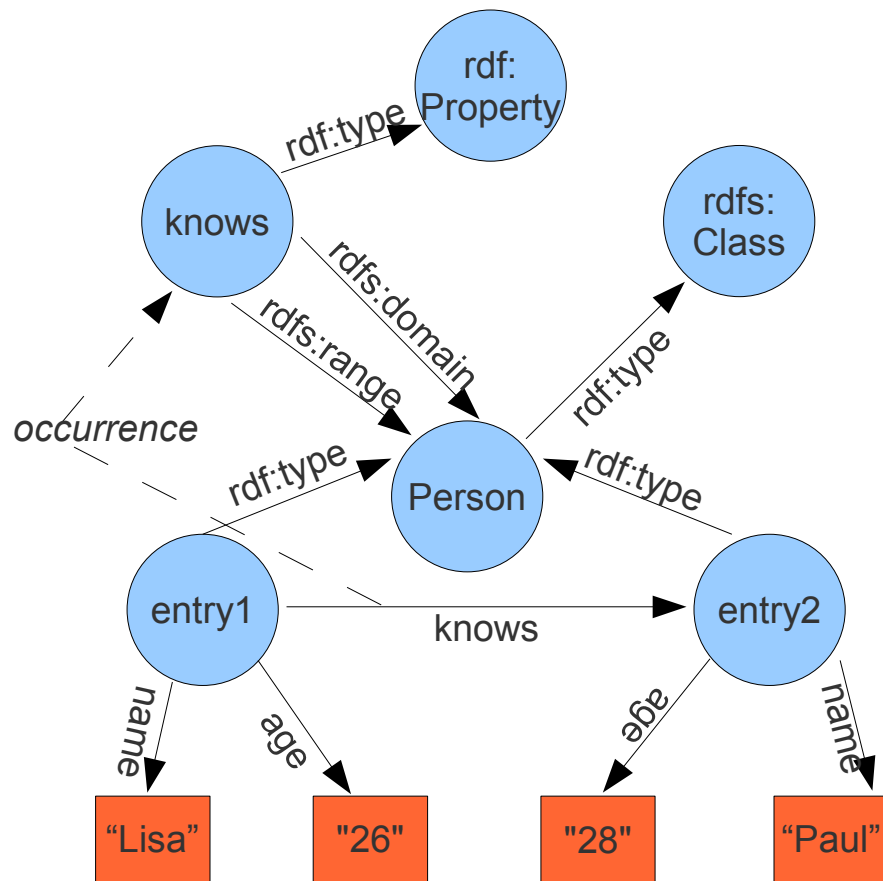




Bridging RDF and TGraphs

Simple Mapping

Example Transformation (from RDF to TGraph)

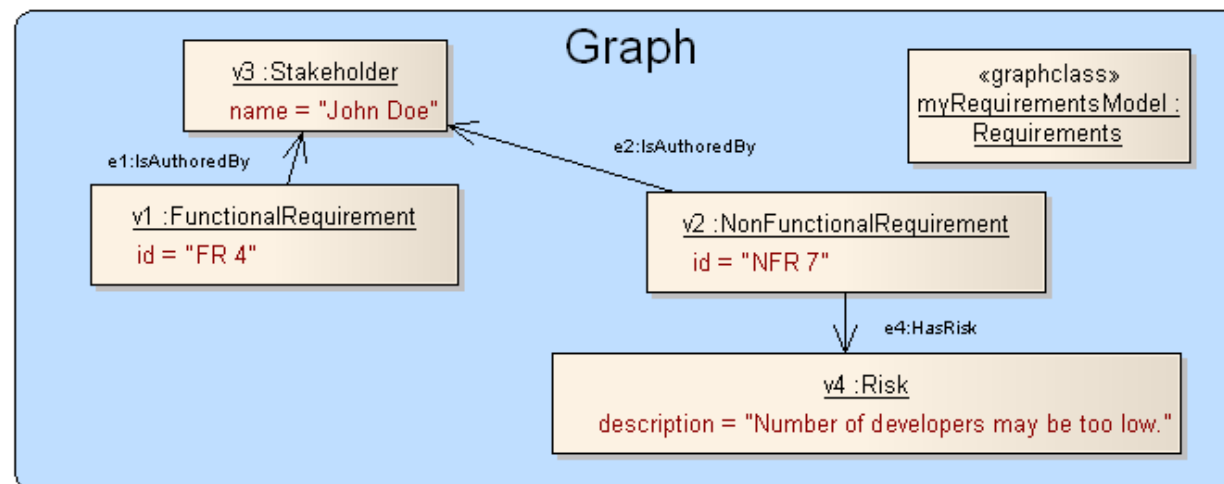
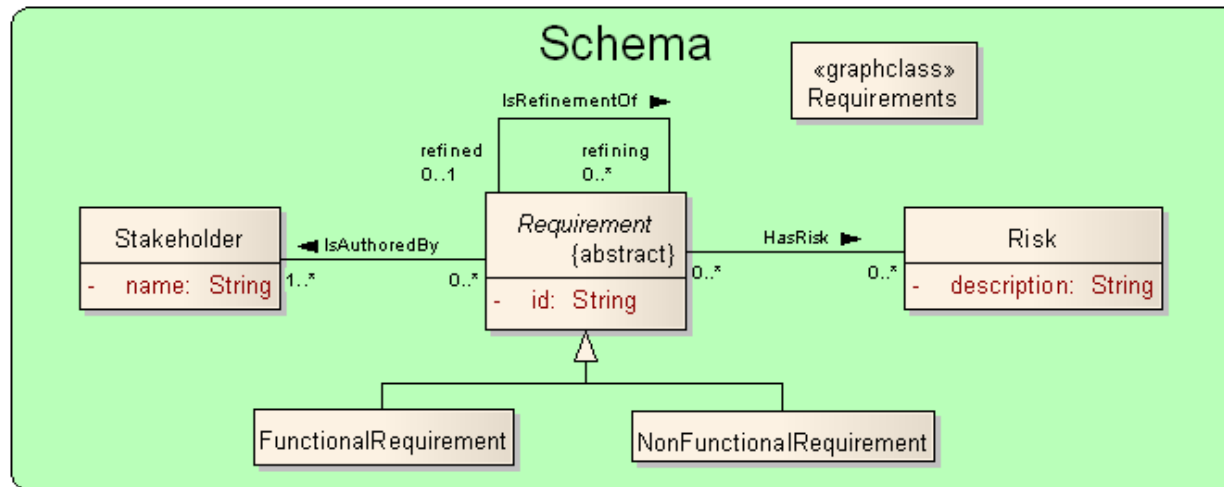




Bridging RDF and TGraphs

Exercise

TGraph to be transformed





Contents

- Introduction and Motivation
- Basics of the Bridging Approach
- Bridging RDF and TGraphs (incl. exercise)
- **Bridging SPARQL and GReQL (incl. exercise)**
- Applications
- Summary and Outlook



Bridging SPARQL and GReQL

Introduction to SPARQL



General Information

- **SPARQL Protocol and RDF Query Language**, issued by the World Wide Web Consortium (W3C)
- SPARQL is a standard query language for **querying RDF data**.
- SPARQL is also widely used for **querying OWL ontologies**, for they can be serialized to RDF.



Bridging SPARQL and GReQL

Introduction to SPARQL

Basic SPARQL Language Concepts

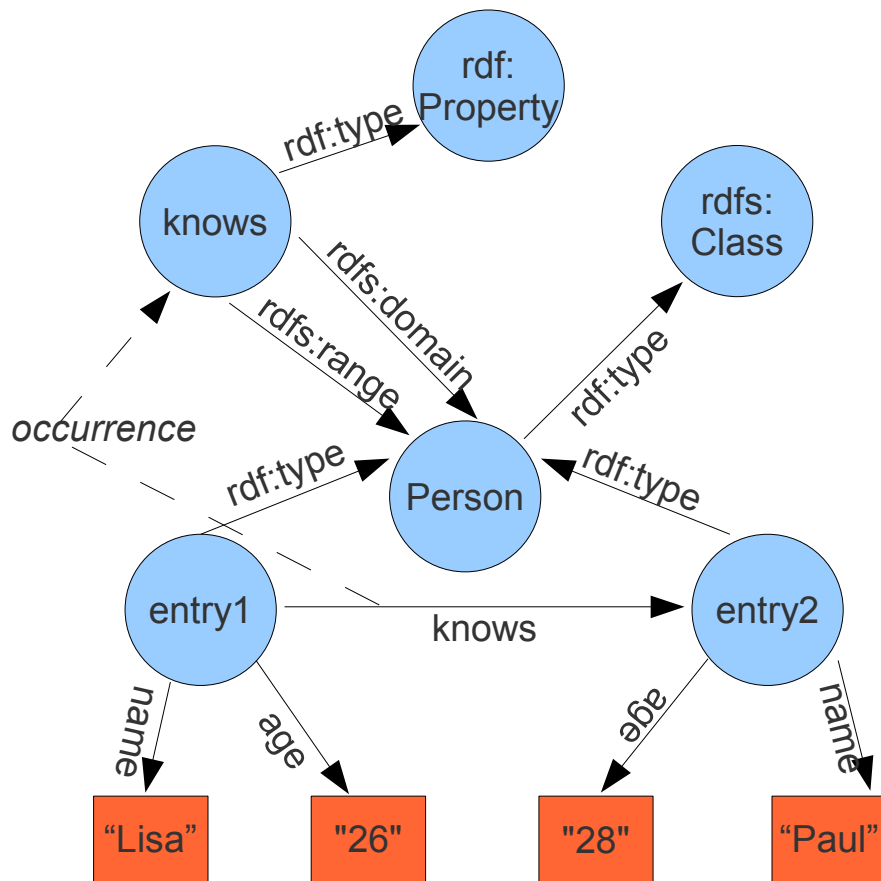
- SPARQL queries consist of up to **five query parts**:
 - ◆ **Prologue** – defines prefixes for abbreviating URIs
 - ◆ **Query form** – determines the structure of a query's result
 - ◆ **Dataset** – specifies the RDF documents to be queried
 - ◆ **Where clause** – contains **triple patterns** to restrict eligible solutions (answers) to the query
 - ◆ **Solution modifiers** – serves to sort or to restrict the number of query solutions



Bridging SPARQL and GReQL

Introduction to SPARQL

Triple Patterns and Solution Mappings



- Triple patterns resemble RDF triples with parts possibly replaced by variables, e.g.

a) `?entry address:name "Lisa"`
 b) `address:entry2 ?p ?o`

- A solution mapping μ is a mapping from variables to RDF terms (resources, blank nodes, identifiers).

$\mu_a(?entry) = address:entry1$
 $\mu_{b_1}(?p) = rdf:type, \mu_{b_1}(?o) = address:Person$
 $\mu_{b_2}(?p) = address:age, \mu_{b_2}(?o) = "28"$
 $\mu_{b_3}(?p) = address:age, \mu_{b_3}(?o) = "Paul"$



Bridging SPARQL and GReQL

Introduction to SPARQL



Query Forms

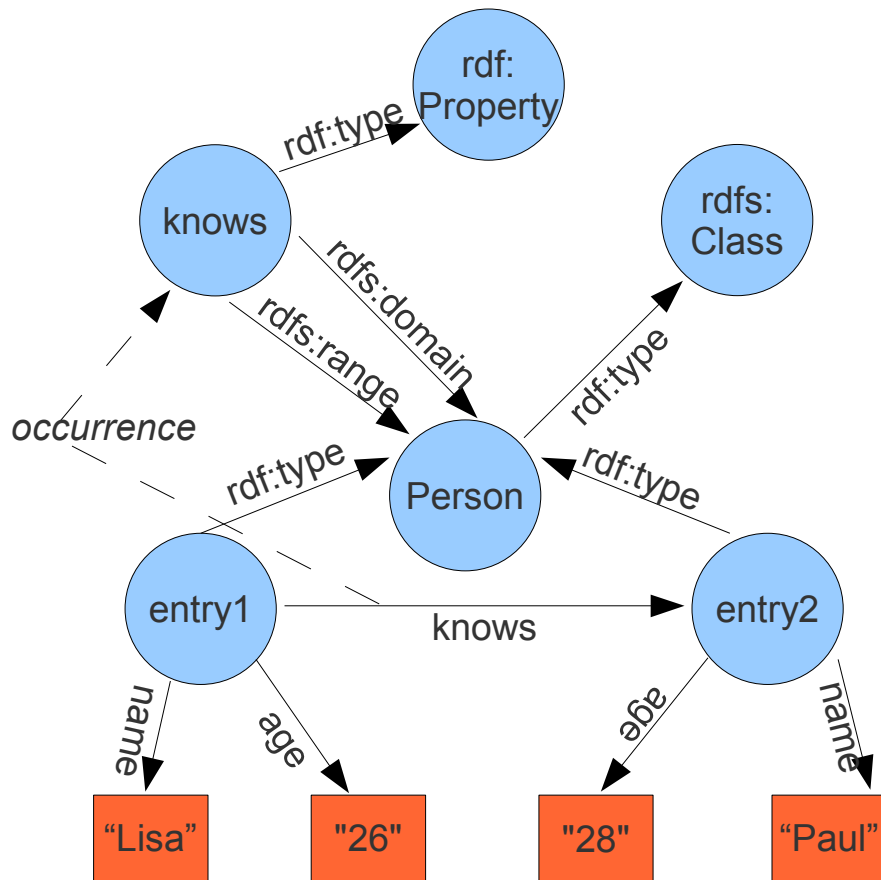
- **SELECT** – returns a **bag of solution mappings**, i.e. mappings from variables to RDF terms (resources, blank nodes, literals), which fulfill the triple patterns in the where clause
- **ASK** – returns a **boolean value** specifying if there is at least one solution mapping fulfilling the triple patterns in the where clause
- **CONSTRUCT** – builds a **new RDF graph** from the solution mappings fulfilling the triple patterns in the where clause
- **DESCRIBE** – returns an **RDF graph describing resources**, with the graph structure determined by the provider of the queried RDF graph



Bridging SPARQL and GReQL

Introduction to SPARQL

Examples – Queries with Join and Optional



```
SELECT ?p ?name
```

```
WHERE {
  address:entry1 ?p ?o .
  ?o address:name ?name .
  ?o rdf:type address:Person .
}
```

Result: {"Paul", address:knows}

```
SELECT ?entry ?city
```

```
WHERE {
  ?s rdf:type address:Person .
  OPTIONAL {
    ?s address:city ?city .
  }
}
```

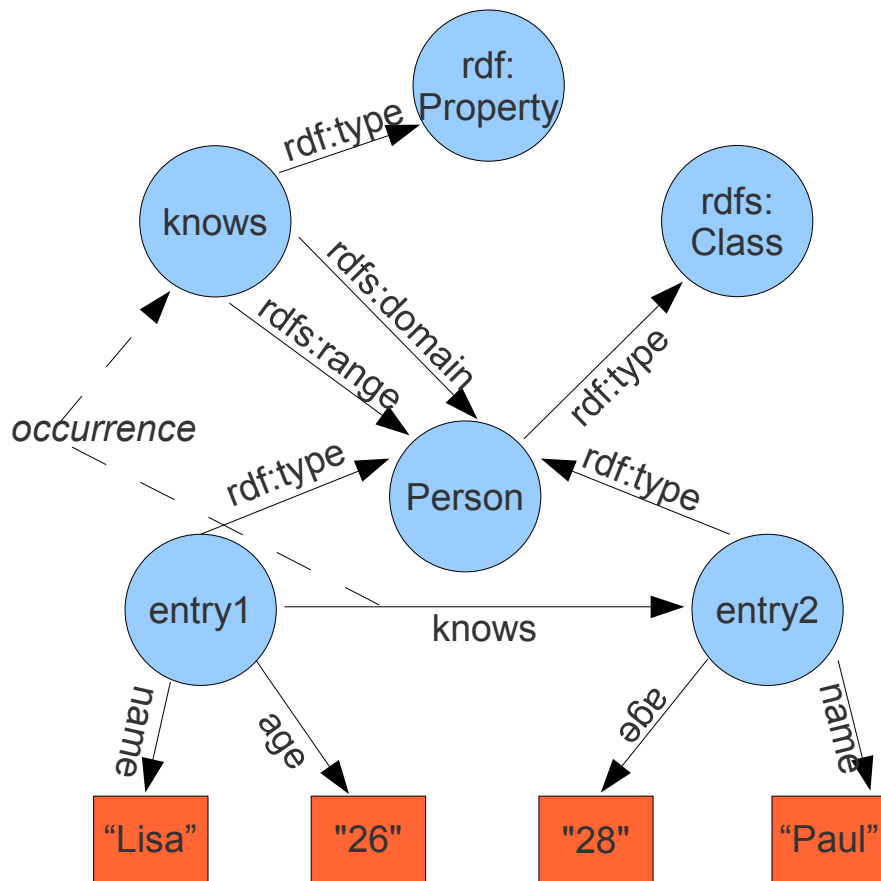
Result: {(address:entry1, □),
(address:entry2, □)}



Bridging SPARQL and GReQL

Introduction to SPARQL

Examples – Queries with Union and Filter



```

SELECT ?entry ?name
WHERE {
    {
        ?entry address:age "26" .
        ?entry address:name ?name .
    } UNION {
        ?entry2 address:knows ?entry .
        ?entry address:name ?name .
    }
}
ORDER BY ?name

Result: {(address:entry1, "Lisa"),
         (address:entry2, "Paul")}
    
```

```

ASK
WHERE {
    ?entry address:name ?name .
    FILTER regex(?name, "^L")
}

Result: yes
    
```



Bridging SPARQL and GReQL

Introduction to GReQL



General Information

- **Graph Repository Query Language**
- GReQL is a language for **querying TGraphs**, including the retrieval of **information on TGraph schemas**.
- GReQL is an **expression language**, i.e. all language elements are expressions and can be used in other expressions.
- GReQL provides a function library covering aspects such as logics, arithmetics, or graph and path analysis.



Bridging SPARQL and GReQL

Introduction to GReQL



Important GReQL Expressions

- Regular path expressions
- `from-with-report` expressions
- Quantified expressions



Bridging SPARQL and GReQL

Introduction to GReQL

Regular Path Expressions (RPEs)

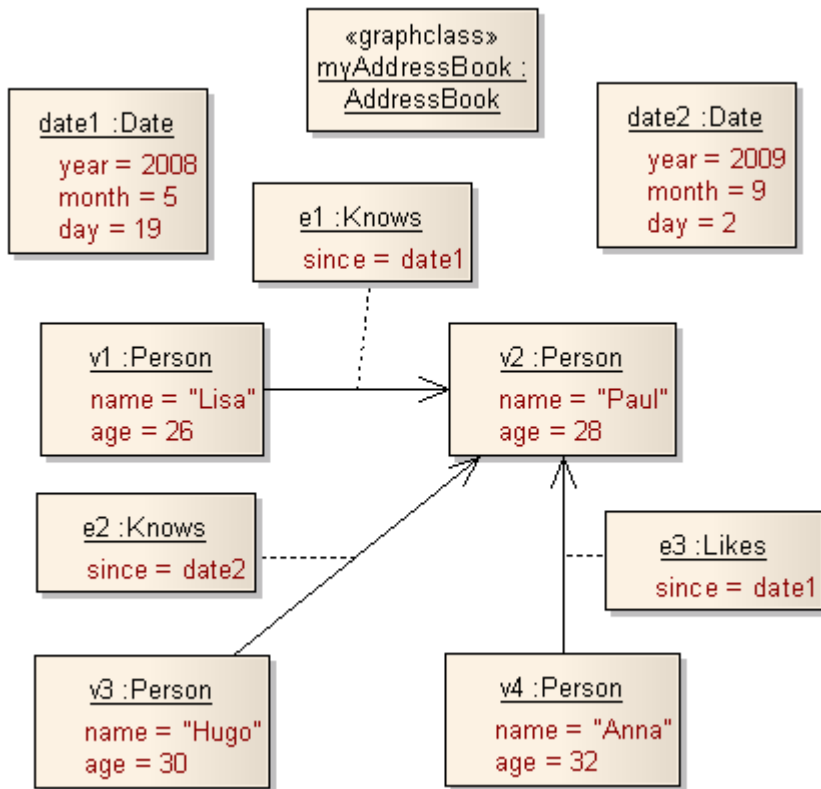
- RPEs describe the **structure of paths** in a graph.
- They are not used as stand-alone expressions, but within
 - ♦ **path existence** expressions – Check for the existence of paths between two vertices.
 - ♦ **forward vertex set** expressions – Return all vertices reachable from a given vertex.
 - ♦ **backward vertex set** expressions – Return all vertices from which a given vertex can be reached.



Bridging SPARQL and GReQL

Introduction to GReQL

Regular Path Expressions (RPEs) – Examples



- Path existence

`v1 --> v2`

Result: true

`v1 --> <-> v4`

Result: true

`v2 <-->{Knows @ year = 2007} v3`

Result: false

- Forward vertex set

`v2 <-->{Knows} | <-->{Likes}&{Person}`

Result: {v3, v4}

`v1 --e1-> [<--]`

Result: {v2, v3, v4}

- Backward vertex set

`<->* v4`

Result: {v1, v2, v3, v4}

`{Person}&<->+ v4`

Result: {v1, v2, v3}



Bridging SPARQL and GReQL

Introduction to GReQL



from-with-report (FWR) Expressions

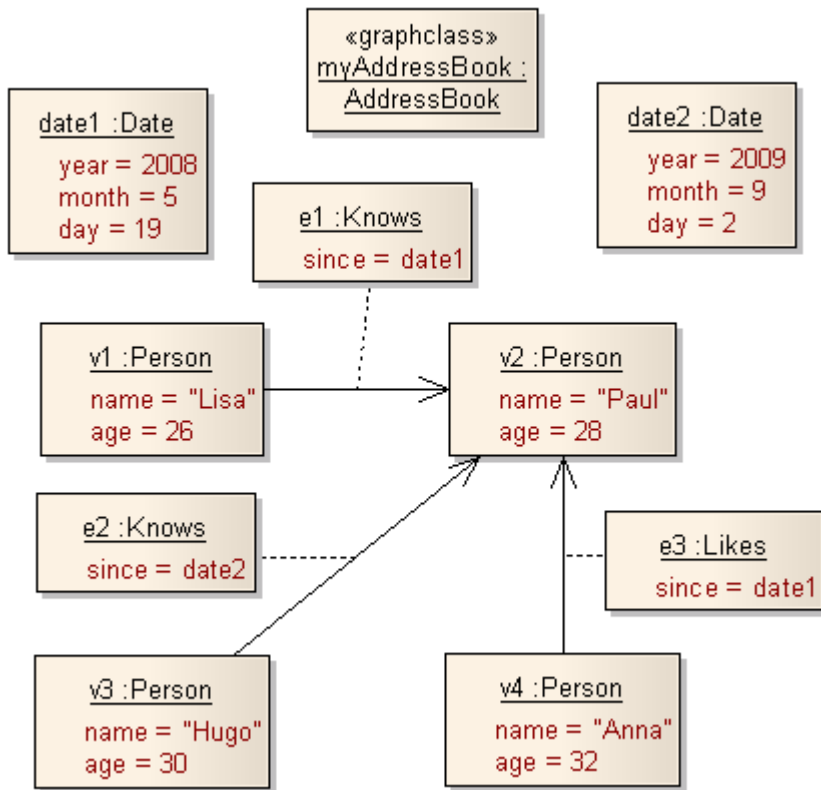
- FWR expressions return **a bag of tuples**.
- The **from part** binds variables to domains, e.g. all instances of a given vertex class.
- The **with part** specifies a boolean expression which imposes constraints on values to be included in the query result.
- The **report part** specifies the structure of the result.



Bridging SPARQL and GReQL

Introduction to GReQL

from-with-report (FWR) Expressions – Examples



```

from p:V{Person}
with p.age > 27
       and not isEmpty(p <->{Likes})
report p.name
end
    
```

Result: {"Paul"}, {"Anna"}

```

from p,q:V{Person}, k:E{Knows}
with p --k-> q and k.since.year > 2007
report p.name, q.name
end
    
```

Result: {"Lisa", "Paul"}, {"Hugo", "Paul"}



Bridging SPARQL and GReQL

Introduction to GReQL



Quantified Expressions

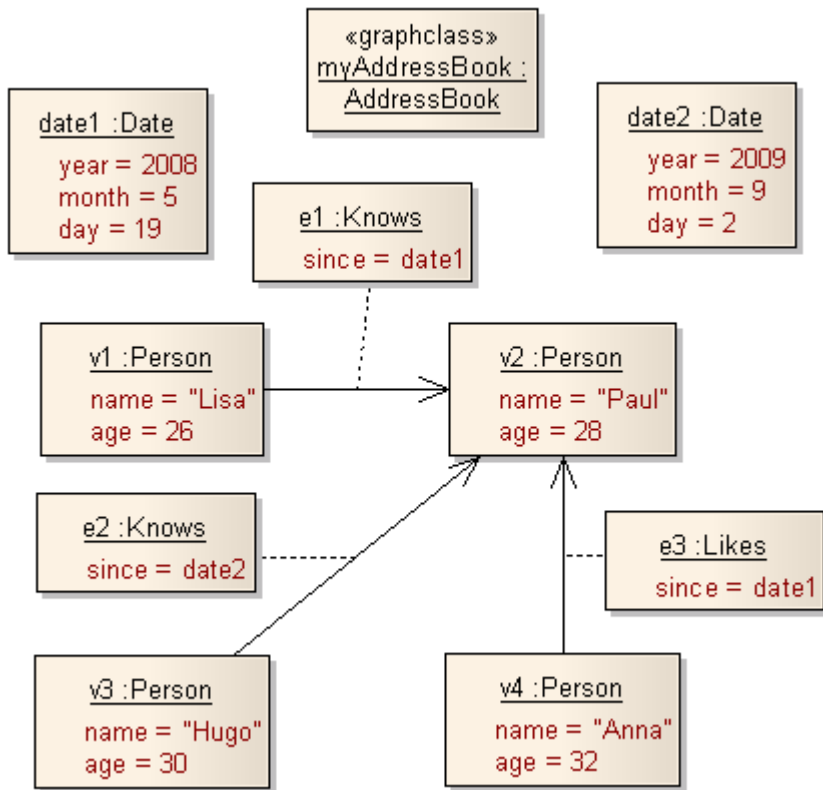
- **universally quantified expressions** – Check if all elements of a collection of elements fulfill a given boolean expression.
- **existentially quantified expressions** – Check if at least one element of a collection of elements fulfill a given boolean expression.
- **uniquely quantified expressions** – Check if exactly one element of a collection of elements fulfill a given boolean expression.



Bridging SPARQL and GReQL

Introduction to GReQL

Quantified Expressions – Examples



- Universal quantification

```
forall p:V{Person} @ p.age > 0
```

Result: true

```
forall e:E{Knows,Likes}
  @ startVertex(e) <> endVertex(e)
```

Result: true

- Existential quantification

```
exists p,q:V{Person} @ p -->{Likes} q
```

Result: {v2, v3, v4}

- Unique quantification

```
exists! p:V{Person} @ p.name = "myself"
```

Result: false



Bridging SPARQL and GReQL

Introduction to GReQL



Why GReQL?

- GReQL is in general **comparable** to the standard **Object Constraint Language** (OCL) by the OMG, but offers some **advantages**:
 - ◆ efficient handling of graph structures
 - ◆ concise description of path structures with RPEs
 - ◆ computation of transitive closure, which is problematic with OCL
- OCL features **not supported** by GReQL:
 - ◆ definition of contexts
 - ◆ generic *iterate* expression



Bridging SPARQL and GReQL

Mapping between SPARQL and GReQL

Relevant Differences (selection, see paper)

SPARQL

- SPARQL is not an expression language. It offers four query forms.
- There are four different result structures.
- The computation of transitive closure over properties is not supported.
- It is possible to construct new RDF graphs.

GReQL

- GReQL is an expression language, with every expression being a valid query.
- Query results can be of many different structures.
- GReQL allows to compute transitive closure.
- GReQL does not allow for creating TGraphs.



Bridging SPARQL and GReQL

Mapping between SPARQL and GReQL

SPARQL Query Parts – GReQL Expressions

SPARQL query part	GReQL construct	Comment
Prologue	-	not needed in GReQL
Query Form		
SELECT	report part of FWR expr.	
ASK	existentially quantified expression	
CONSTRUCT	-	no support in GReQL for building graphs
DESCRIBE	-	no support in GReQL for external descriptions
Dataset	-	GReQL only allows querying one graph
Where clause		
with SELECT	from and with parts of FWR expression(s)	variables in the where clause are mapped to the from part, triple patterns are mapped to with part
with ASK	existentially quantified expression	variables in the where clause are mapped to the declaration part, triple patterns are mapped to boolean expressions
Solution modifier	-	no support for sorting and other modifiers in GReQL



Bridging SPARQL and GReQL

Transformation from SPARQL to GReQL

Basic Approach

- There are **two transformations** from SPARQL to GReQL, depending on whether **schema-aware** or **simple transformation** was applied on the RDF document.
- For the schema-aware variant, the SPARQL query has to adhere to **restrictions** derived from the restrictions on RDF documents, e.g.:
 - ♦ A variable may be used in at most one `rdf:type` triple pattern.
 - ♦ A variable used as object in a `rdf:type` triple pattern may not be used as subject elsewhere.
- Only **SELECT** and **ASK** queries can be transformed.



Bridging SPARQL and GReQL

Transformation from SPARQL to GReQL

Examples (for schema-aware transformation)

```

SELECT ?s ?o
WHERE {
  ?s address:knows ?o .
  ?s address:age "26" .
}
  
```

Result: {(address:entry1,
address:entry2)}

```

from s, o:V
with s -->{Knows} o
      and hasAttribute(s, "age")
      and s.age = 26
report s, o
end
  
```

Result: {(v1, v2)}

```

SELECT ?p
WHERE {
  address:entry1 ?p ?o .
  ?o rdf:type address:Person .
}
  
```

Result: {(address:knows)}

```

from p:E, o:V{Person}
with not isEmpty({@ thisVertex.uriRef
                  = "address:entry1"}&--p-> o)
report p
end
  
```

Result: {(e1)}



Bridging SPARQL and GReQL

Transformation from SPARQL to GReQL

Examples (for simple transformation)

<pre> SELECT ?s ?o WHERE { ?s address:knows ?o . ?s address:age "26" . } </pre>	<pre> from s, o:V with s -->{@ thisEdge.uriRef = "address:knows"} o and not isEmpty(s -->{@ thisEdge.uriRef = "address:age"} &{Literal @ thisVertex.value = "26"}) report s, o end </pre>
---	---

Result: {(address:entry1,
address:entry2)}

Result: {(v3, v4)}

<pre> SELECT ?p WHERE { address:entry1 ?p ?o . ?o rdf:type address:Person . } </pre>	<pre> from p:E, o:V with not isEmpty({@ thisVertex.uriRef = "address:entry1"}&--p-> o) and not isEmpty(o -->{@thisEdge.uriRef = "rdf:type"} &{Resource @ thisVertex.uriRef = "address:Person"}) report p end </pre>
--	---

Result: {(address:knows)}

Result: {(e3)}



Bridging SPARQL and GReQL

Transformation from GReQL to SPARQL

Basic Approach

- Only **one transformation** exists, for there is only one transformation from TGraphs to RDF (**schema-aware**).
- Only FWR and existentially quantified expressions can be transformed.
- RPEs may not contain iterated path descriptions (Kleene-star or -plus), as this is not supported by SPARQL.



Bridging SPARQL and GReQL

Transformation from GReQL to SPARQL

Basic Approach (cont'd)

- Regular path expressions in the GReQL query must be rewritten in a **disjunctive normal form** by using distributivity rules.
- The resulting **alternatives of sequential** path descriptions can be transformed to **unions of joined** triple patterns in SPARQL.



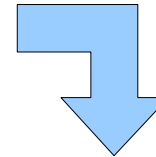
Bridging SPARQL and GReQL

Transformation from GReQL to SPARQL

Example

```

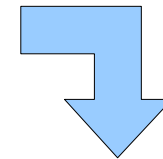
from p,q:V{Person} // (1)
with p -->{Knows} [ <--{Knows} ] q // (2)
      and p.age < 27 // (3)
report p, q.name // (4)
end
    
```



normalizing path descriptions

```

from p,q:V{Person} // (1)
with (p -->{Knows} q or p -->{Knows}<--{Knows} q) // (2)
      and p.age < 27 // (3)
report p, q.name // (4)
end
    
```



transformation to SPARQL

```

SELECT ?p ?name // (4)
WHERE {
  ?p rdf:type Person . ?q rdf:type Person . // (1)
  { { ?p knows ?q } UNION { ?p knows ?x . ?q knows ?x } } // (2)
  ?p age ?age . FILTER (?age < 27) . // (3)
  ?q name ?name . // (4)
}
    
```



Bridging SPARQL and GReQL

Exercise

GReQL Query to be transformed (schema-aware)

- Find all Requirements authored by “John Doe” together with their Risks!

```
from s:V{Stakeholder}, req:V{Requirement}, risk:V{Risk}
with s.name = "John Doe"
    and s <-->{IsAuthoredBy} req -->{HasRisk} risk
report req.id, risk.description
end
```



Bridging SPARQL and GReQL

Exercise

Resulting SPARQL Query (schema-aware)

```
from s:V{Stakeholder}, req:V{Requirement}, risk:V{Risk} // (1)
with s.name = "John Doe" // (2)
    and s <-->{IsAuthoredBy} req -->{HasRisk} risk // (3)
report req.id, risk.description // (4)
end
```

```
SELECT ?id ?desc // (4)
WHERE {
  ?s rdf:type Stakeholder . // (1)
  ?req rdf:type Requirement . // (1)
  ?risk rdf:type Risk . // (1)
  ?req name "John Doe" . // (2)
  ?req isAuthoredBy ?s . // (3)
  ?req hasRisk ?risk . // (3)
  ?req name ?id . // (4)
  ?risk description ?desc . // (4)
}
```



Exercise

Resulting SPARQL Query

- Find all Requirements authored by “John Doe” and their Risks

```
from s:V{Stakeholder}, req:V{Requirement}, risk:V{Risk} // (1)
with s.name = "John Doe" // (2)
    and s <--{IsAuthoredBy} req -->{HasRisk} risk // (3)
report req.id, risk.description // (4)
end
```

```
SELECT ?id ?desc // (4)
WHERE {
  ?s rdf:type Stakeholder . // (1)
  ?req rdf:type Requirement . // (1)
  ?risk rdf:type Risk . // (1)
  ?req name "John Doe" . // (2)
  ?req isAuthoredBy ?s . // (3)
  ?req hasRisk ?risk . // (3)
  ?req name ?id . // (4)
  ?risk description ?desc . // (4)
}
```



Contents

- Introduction and Motivation
- Basics of the Bridging Approach
- Bridging RDF and TGraphs (incl. exercise)
- Bridging SPARQL and GReQL (incl. exercise)
- **Applications**
- Summary and Outlook



Applications Overview



- In ontology-driven software engineering, to avoid formulating similar queries twice
- In any field relying on the **analysis of complex structures of relationships** between entities, GReQL with its RPEs has an advantage over SPARQL, e.g. for
 - ♦ program analysis on the basis of source code ontologies
 - ♦ querying traceability information recorded in ontologies (see following slides)

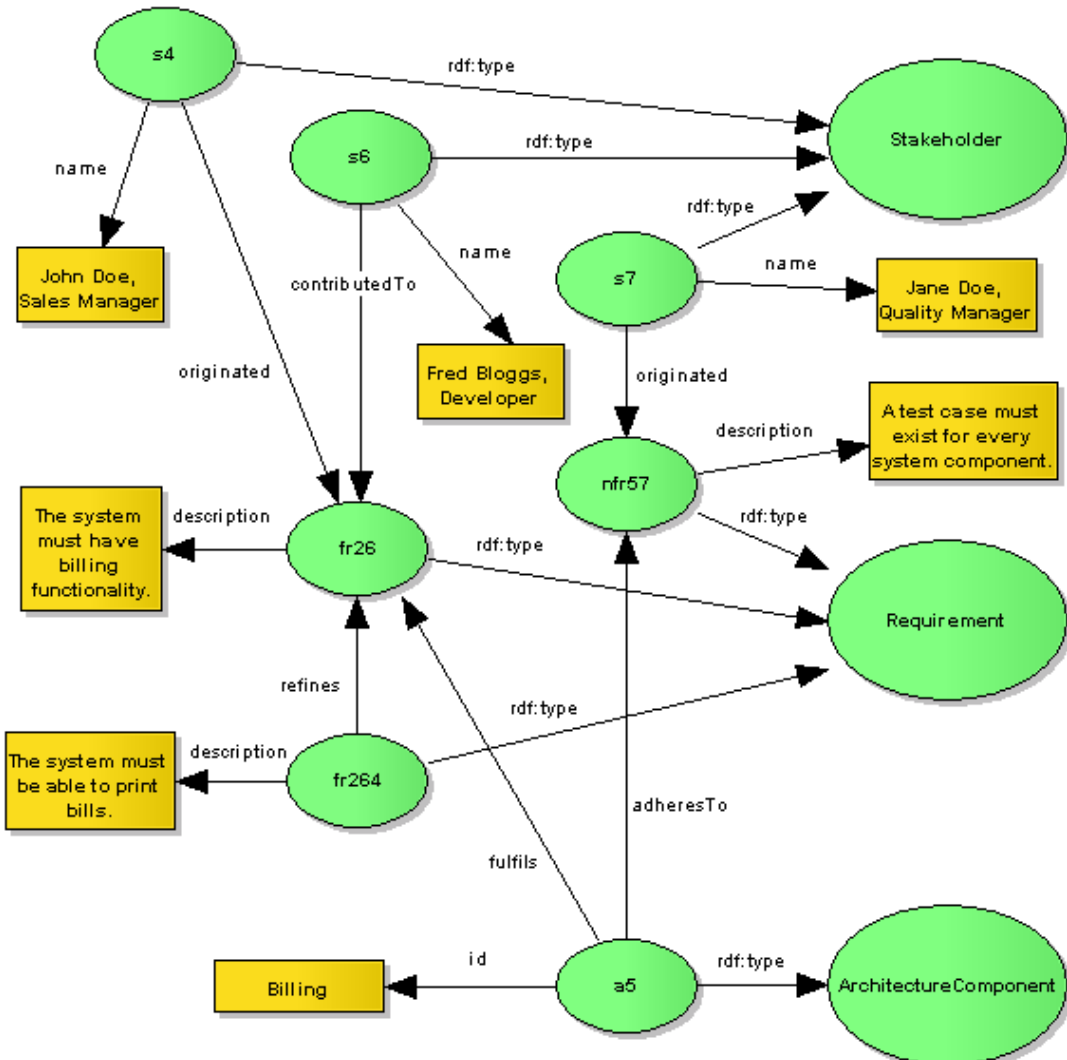


Applications

Querying Traceability Information

- Traceability Queries mostly follow one of three patterns:

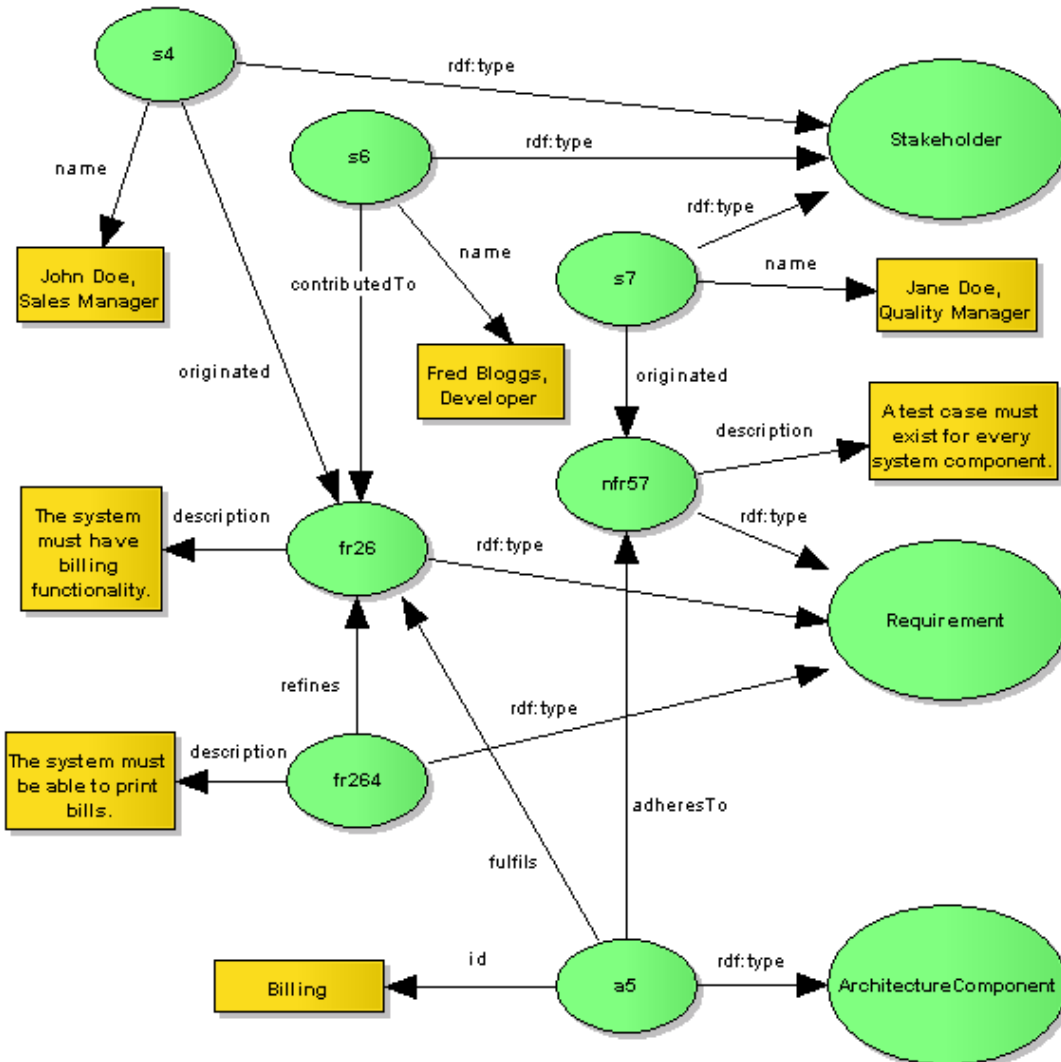
- ◆ **Existence** – Is there a path of traceability relationships between two entities?
- ◆ **Reachable Entities** – Find all entities reachable from a given entity.
- ◆ **Slice** – Find all entities reachable from a given entity and also return all intermediate entities and traceability relationships.





Applications

Querying Traceability Information



- All three patterns require the specification of path structures.
- If traceability information is recorded in ontologies, SPARQL has to be employed for querying.
- Specifying complex path structures with SPARQL results in long, uncomprehensible queries.
- Using GReQL with its RPEs would be a better solution!



Contents

- Introduction and Motivation
- Basics of the Bridging Approach
- Bridging RDF and TGraphs (incl. exercise)
- Bridging SPARQL and GReQL (incl. exercise)
- Applications
- **Summary and Outlook**



Summary

- It is possible to **bridge** the **Ontoware and Modelware** technological spaces by a transformation-based bridge, involving modeling as well as query languages.
 - ◆ Bridge between RDF (and indirectly OWL) and TGraphs
 - ◆ Bridge between SPARQL and GReQL
- For the RDF-TGraph bridge, there exist two kinds of mappings
 - ◆ **Schema-aware mapping** (for “TGraph-like” RDF documents, bidirectional).
 - ◆ **Simple mapping** (for any RDF document, from RDF to TGraphs only)



Summary (cont'd)

- Depending on the employed kind of mapping between RDF and TGraphs, the transformation between **SPARQL** and **GReQL** can also be **schema-aware** or **simple**.
- Benefits of the bridging approach:
 - ♦ **Queries** are **formulated only once** when RDF documents are transformed to TGraphs or vice versa (e.g. in Ontology-driven Software Development).
 - ♦ **Regular path expressions** can be used to represent path structures in RDF ontologies (e.g. for querying traceability information).
 - ♦ GReQL queries are **more concise** than SPARQL queries.



Outlook

- The **GReQL to SPARQL transformation** is going to be implemented.
- It is to be analysed whether transforming ontologies to TGraphs and querying them with GReQL **offers any advantages with respect to efficiency**, compared to querying with SPARQL.
- It is discussed to include **regular path expressions in SPARQL 1.1**. The developments have to be observed and compared to GReQL with respect to expressivity and efficiency.



Thank you



*Institute for
Software Technology*



MOSTPROJECT

<http://most-project.eu>