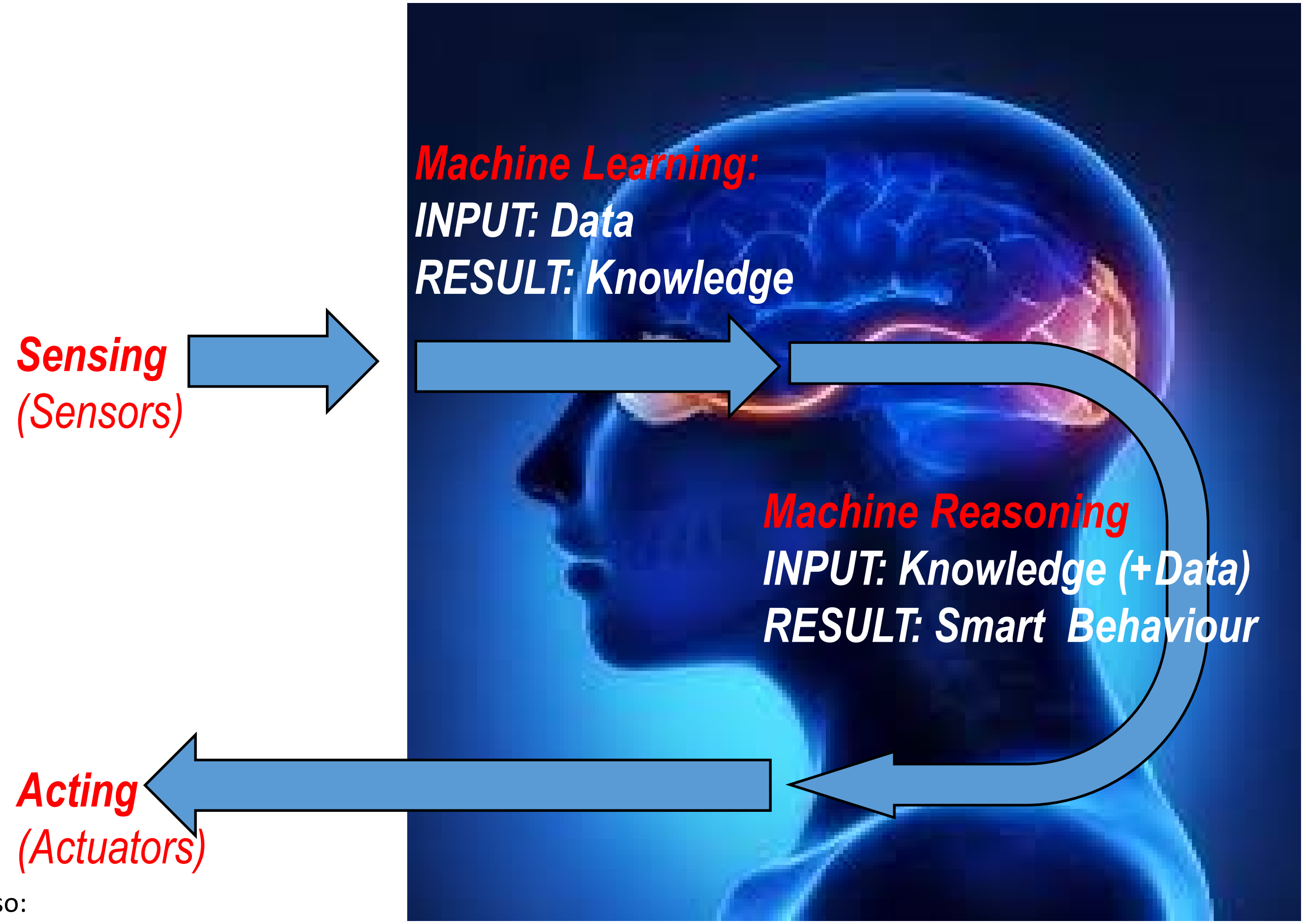
Machine Reasoning

Knowledge Engineering SS24
MSc Computer Science
Camerino, 06/05/2024
Dr. Emanuele Laurenzi

The two sides of A.I.: Machine Learning and Machine Reasoning



Two Sides of A. I.

Machine Learning =
Non-symbolic AI
(Neural Networks,
Deep Learning,
Knowledge Discovery)
+
Machine Reasoning =
Symbolic AI
(Semantic Technology,
Knowledge Representation,
Knowledge Engineering)

See also:
https://www.ipsoft.com/2018/06/22/dws-2018-inside-amelia-brain/

From lecture of Prof. Dr. Robert Buchmann

Languages for Machine Reasoning

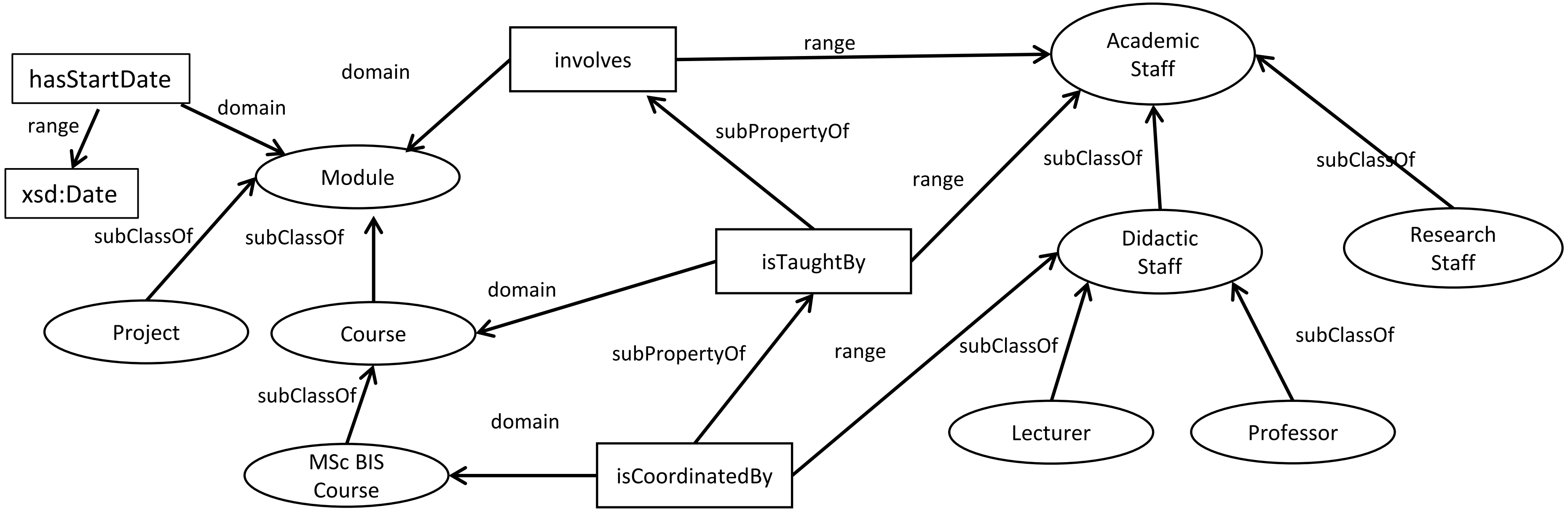
- There exist languages maintained by the W3C that allow machine reasoning.
- Machine reasoning means applying reasoning services on knowledge graphs or ontologies that are expressed in some ontology language.
- The focus in this lecture is on the following languages:
  - SPARQL CONSTRUCT/INSERT
  - SWRL
  - SHACL

SPARQL for Machine Reasoning

SPARQL for Machine Reasoning

- CONSTRUCT vs. INSERT
  - "Reasoning on the fly" vs. "Reasoning with writing“
- CONSTRUCT delivers the transformed/extracted subgraph to the client, without storing it (the client app can later choose to save it or just display it);
- INSERT stores the generated graph, without returning it (the client app must perform a SELECT to retrieve what was generated).

Solution for the Ontology Development 101 exercise

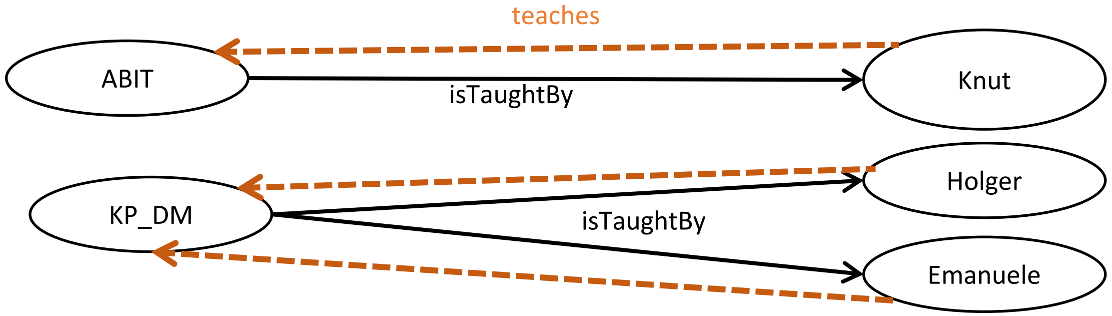


A possible rule

– The following rule derive the inverse property is_taught_by

```
CONSTRUCT {?y :teaches ?x}
WHERE {
  ?x :isTaughtBy ?y .
}
```

If a course ?x is taught by teacher ?y, then teacher ?y teaches course ?x.



Properties derived by the rule.

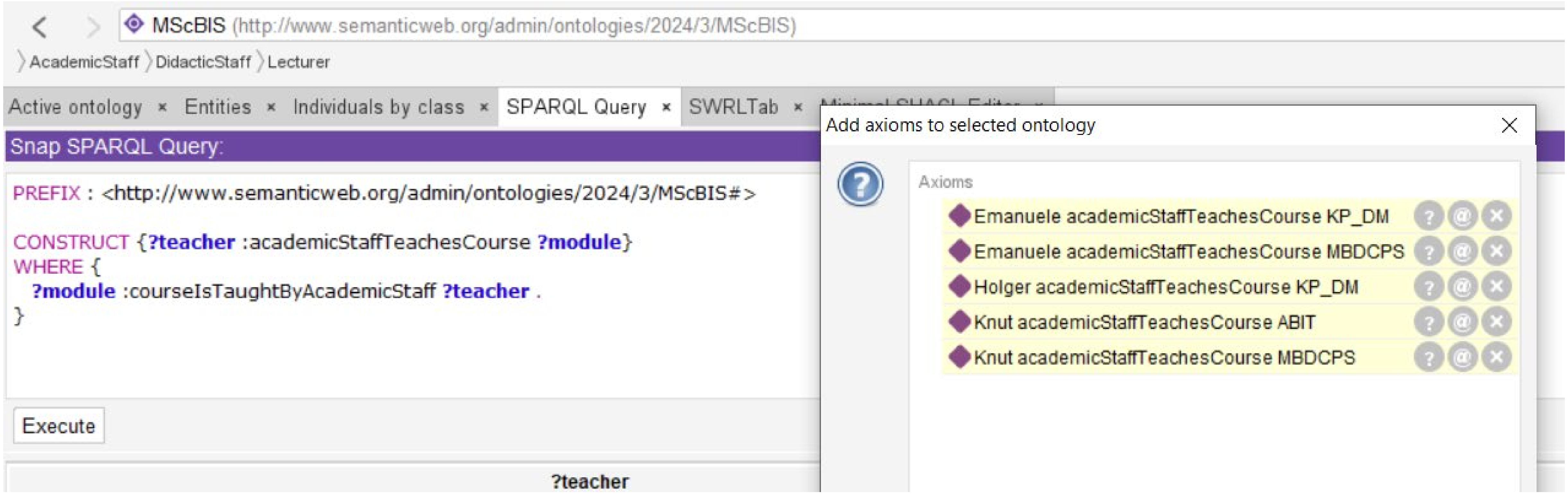
Query Editor | Query Library

```
CONSTRUCT {?teacher teaching:teaches ?module}
WHERE {
  ?module teaching:isTaughtBy ?teacher .
}
```

[Subject]	Predicate	Object
teaching:Emanuele123	teaching:teaches	teaching:KP_DM
teaching:Holger123	teaching:teaches	teaching:KP_DM
teaching:Knut123	teaching:teaches	teaching:ABIT

Test in Protégé

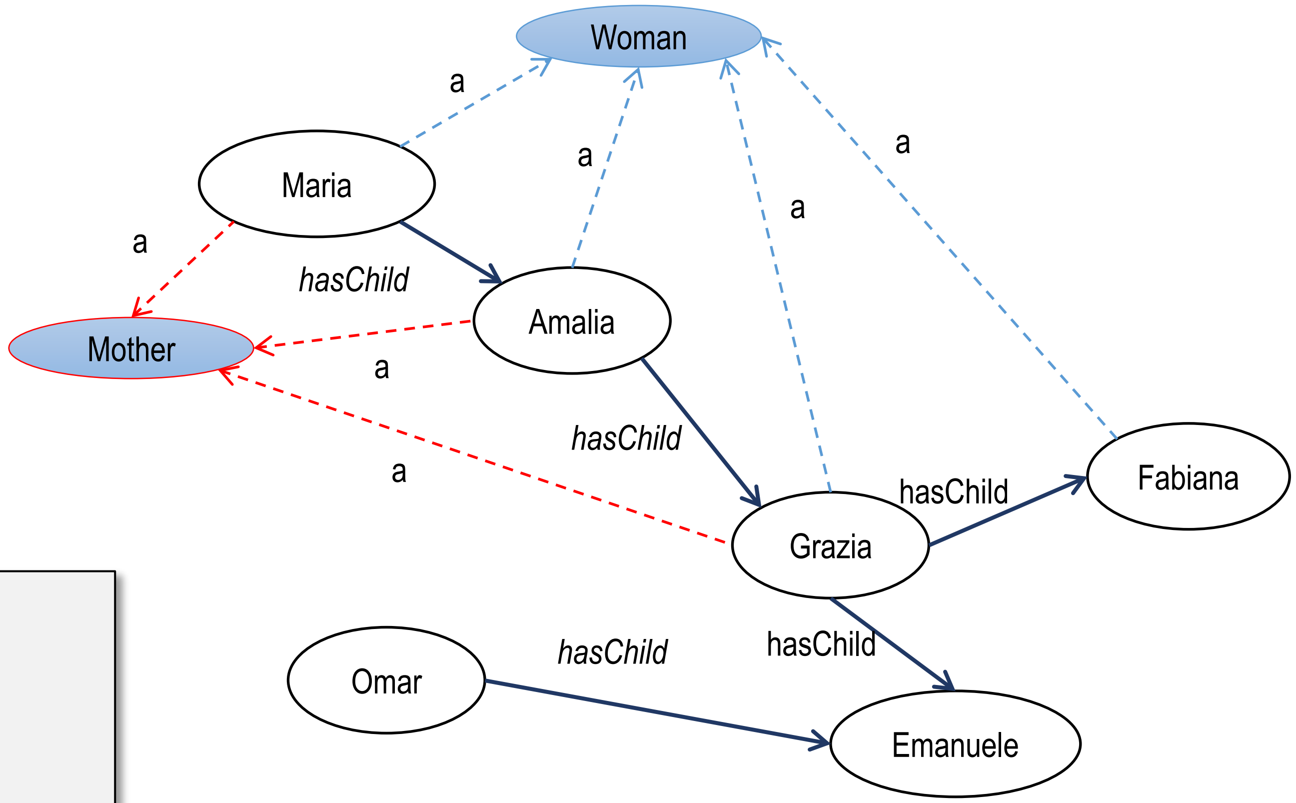
– The reasoner should be started before executing the CONSTRUCT.



Use of some operators of SPARQL for machine reasoning

Conjunction (AND)

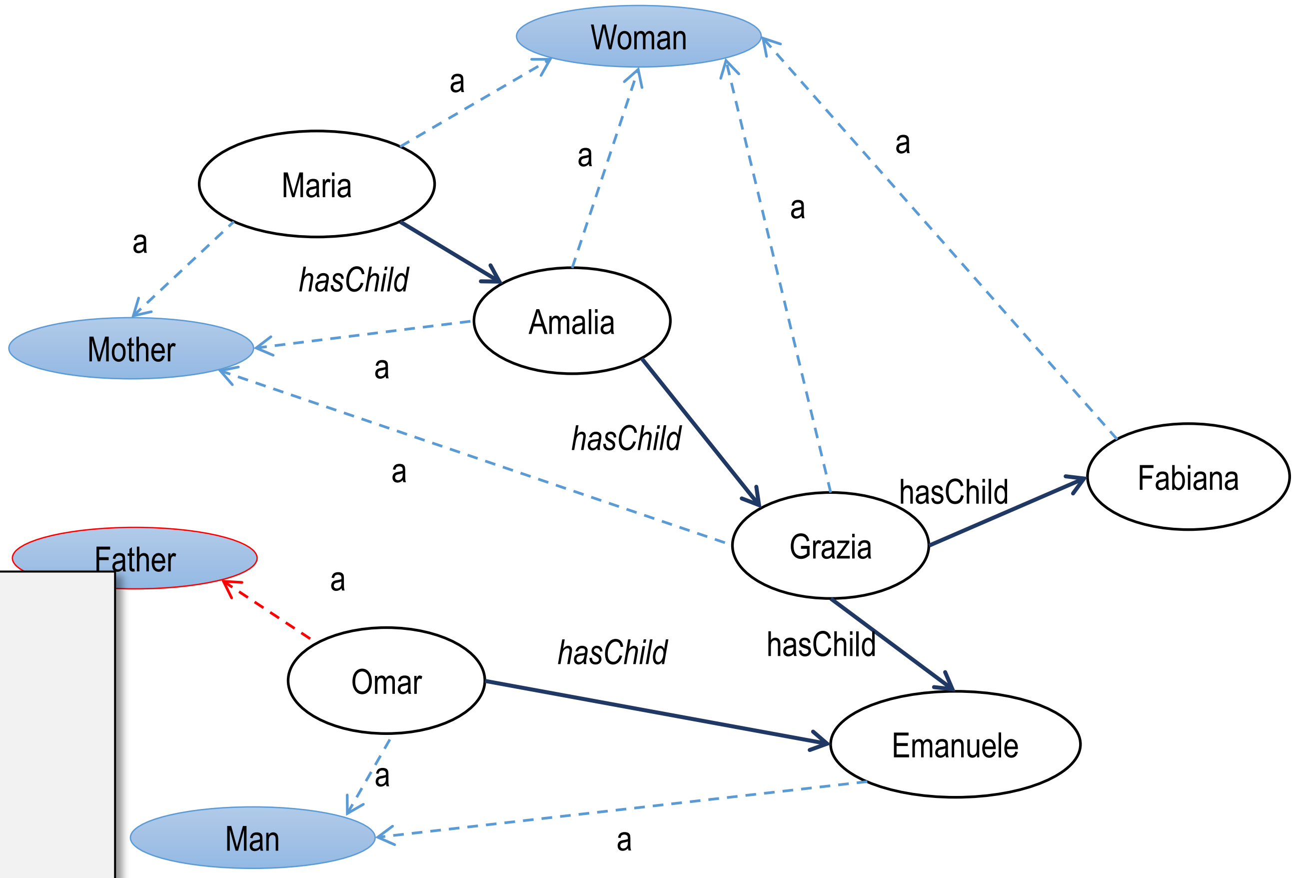
- Use SPARQL for
  - Adding those Women to Mother which have a child




```
PREFIX : <http://laurenzi.ch#>
CONSTRUCT { ?s a :Mother }
WHERE {
    ?s :hasChild ?o .
    ?s a :Woman
}
```

Conjunction (AND)

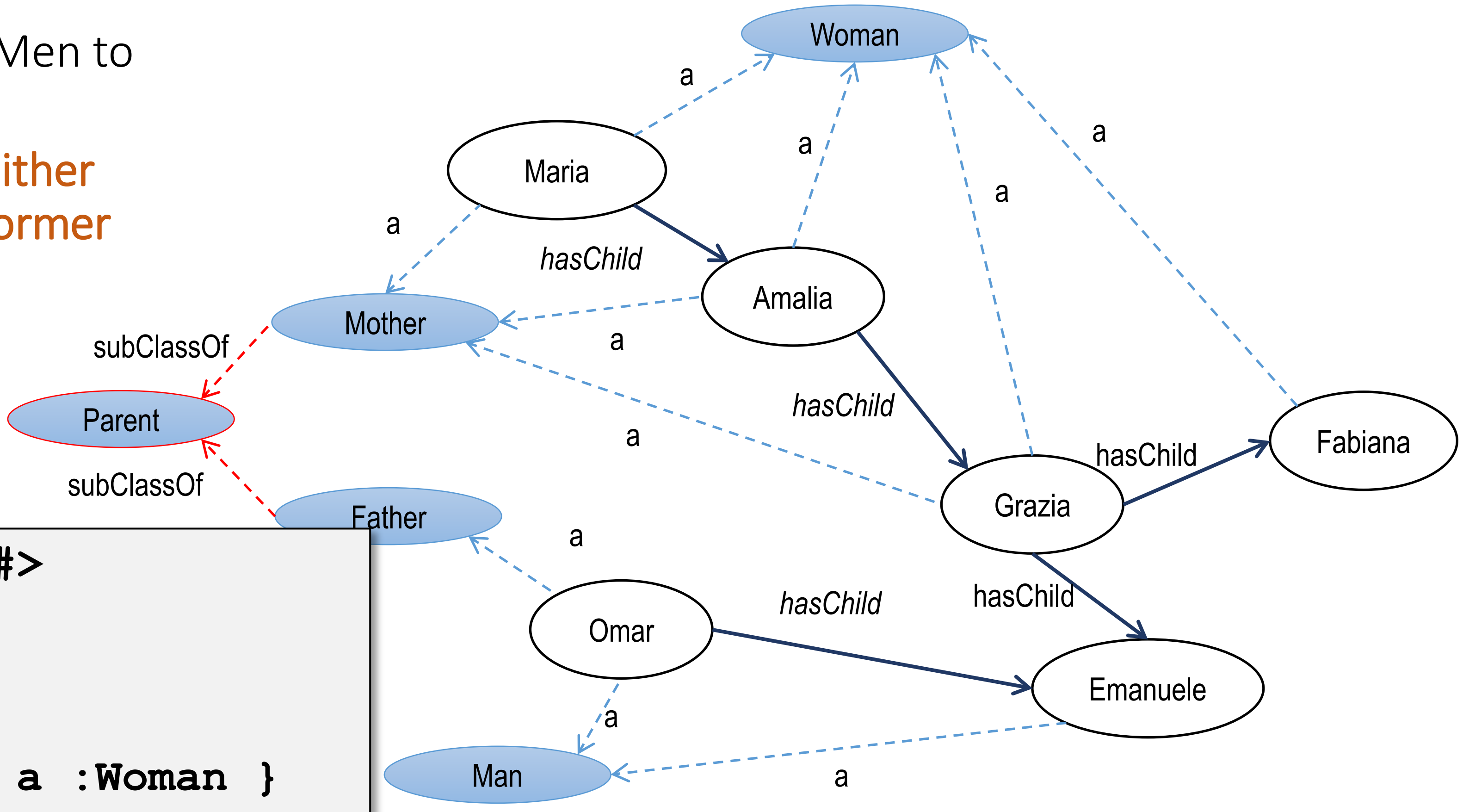
– Use SPARQL for
  – Adding those Men to Father which have a child




```
PREFIX : <http://laurenzi.ch#>
CONSTRUCT { ?s a :Father }
WHERE {
    ?s :hasChild ?o .
    ?s a :Man
}
```

Disjunction (OR)

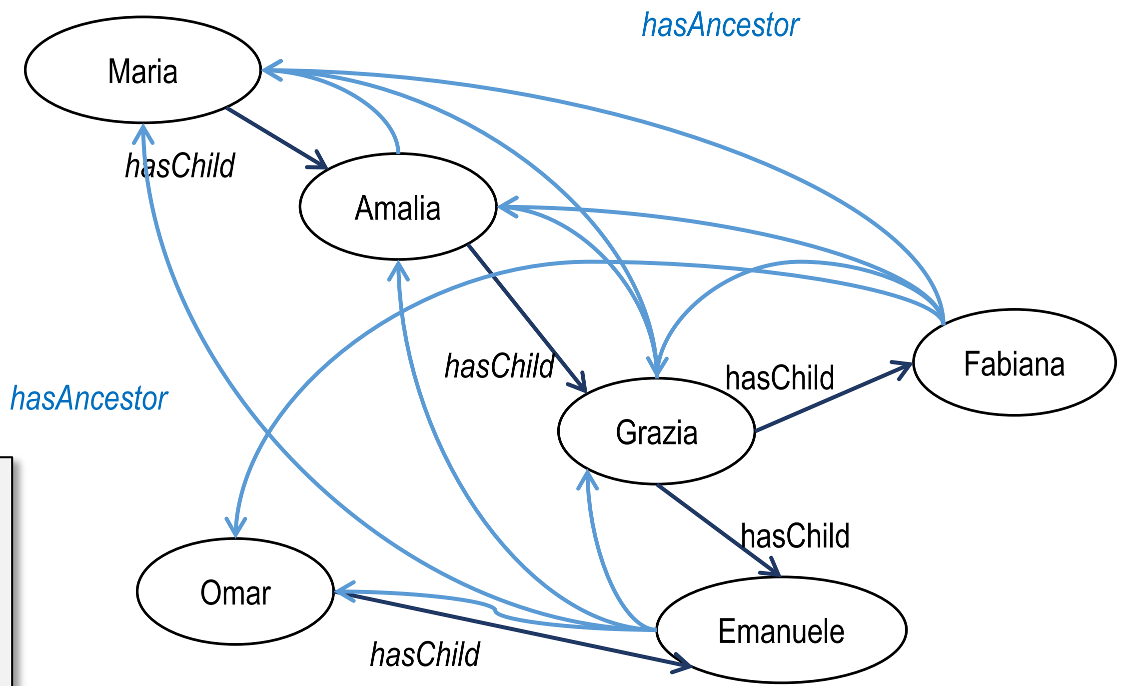
– Use SPARQL for
  – Adding those Women and Men to Parent who have a child
  – Warning: You don't have neither RDFS nor the results from former queries




```
PREFIX : <http://laurenzi.ch#>
CONSTRUCT { ?s a :Parent }
WHERE {
     ?s :hasChild ?o .
     { ?s a :Man } UNION { ?s a :Woman }
}
```

Recursion

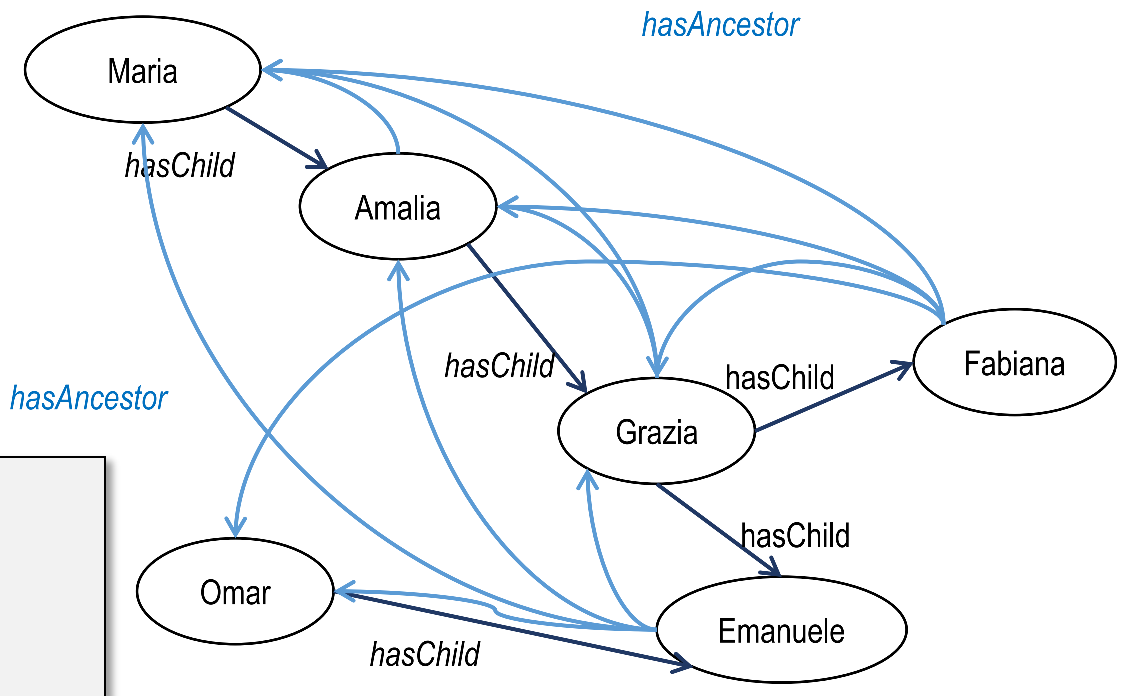
– Use SPARQL for
  – Adding the relationships hasAncestor which is either someone which has a child or someone which has a child whose is already an ancestor




```
PREFIX : <http://laurenzi.ch#>
INSERT { ?d :hasAncestor ?a }
WHERE {
    { ?a :hasChild ?d } UNION
    { ?a :hasChild ?x . ?d :hasAncestor ?x }
}
```

Recursion (Another Solution)

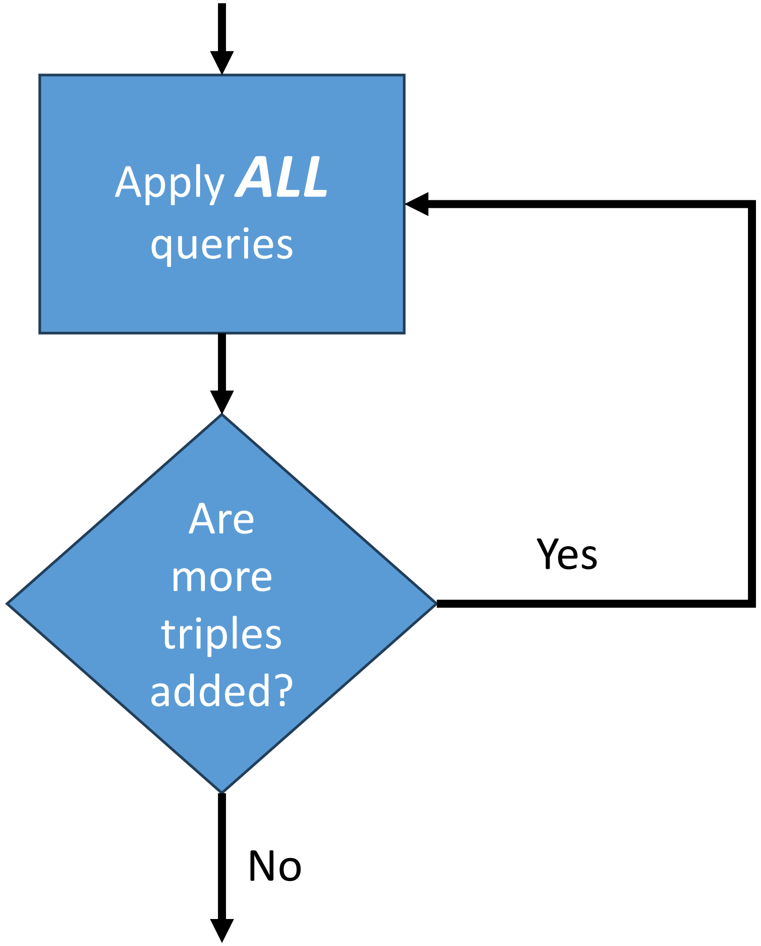
- Use SPARQL for
  - Adding the relationships hasAncestor which is either someone which has a child or someone who has a child that is already an ancestor




```
PREFIX : <http://laurenzi.ch#>
INSERT { ?d :hasAncestor ?a }
WHERE {
    { ?a :hasChild ?d } UNION
    { ?d :hasAncestor ?x . ?x :hasAncestor ?a }
}
```

Learnings from the Recursion

- Queries are NOT applied until no further triples are added.
- In order to ensure the deduction of ALL results you need to implement the loop
  - In a step ALL queries needs to be applied…
  - … until no further triples are added
- Infinite Loops?



Repetition: Inference Procedure for Logic Programming

Rule Engines already include the loop

Let *resolvent* be the query ?- Q_1, ..., Q_m

While *resolvent* is not empty do

1. **Choose** a query literal Q_i from *resolvent*.
2. **Choose** a renamed[1] clause H :- B_1, ..., B_n from P such that Q_i and H unify with an most general **unifier** σ , i.e. $Q_i\sigma = H\sigma$
3. **If** no such Q_i and clause exist, then **backtrack**
4. Remove Q_i from the resolvent
5. **Add** B_1, ..., B_n to the resolvent
6. Add σ to σ_{all}
7. Apply substitution σ to the *resolvent* and go to 1.

If *resolvent* is empty, **return** σ_{all}, else **return** *failure*.

Negation in SPARQL: FILTER NOT EXISTS

- Negation as NOT EXISTS
  - True if a specific graph does not exists
- Opposite of Negation: EXISTS ☺

```
@prefix : <http://example/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .

:alice rdf:type foaf:Person .
:alice foaf:name "Alice" .
:bob rdf:type foaf:Person .
```

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?person
WHERE {
   ?person rdf:type foaf:Person .
   FILTER NOT EXISTS { ?person foaf:name ?name }
}
```

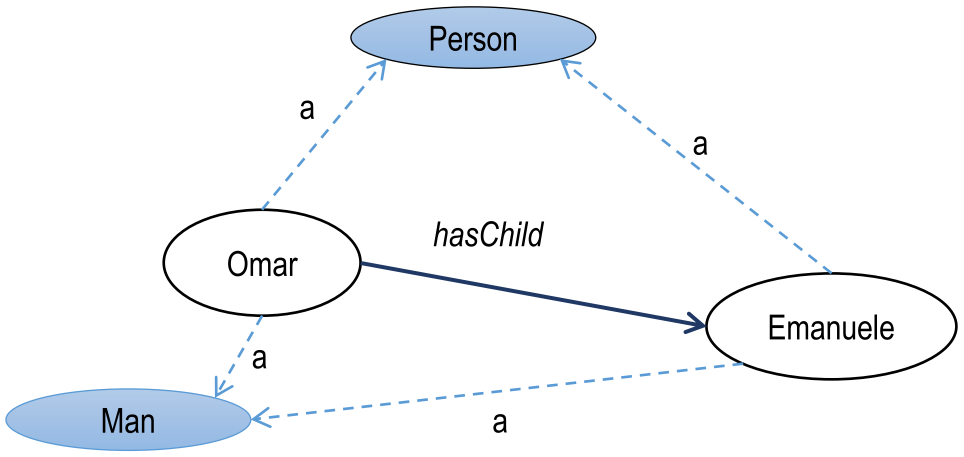
© https://www.w3.org/TR/sparql11-query/#negation [28.05.2024]

Negation

```
PREFIX : <http://laurenzi.ch#>
CONSTRUCT { ?s a :UnhappyMan }
WHERE {
    ?s a :Man .
    FILTER NOT EXISTS {?s a :Father}
}
```

- Use SPARQL for
  - Finding out who is an unhappy man. An unhappy man is a man who is NOT a father

- Is Omar a Father?
- Is Omar an UnhappyMan?
- Is Emanuele a Father?
- Is Emanuele an UnhappyMan?

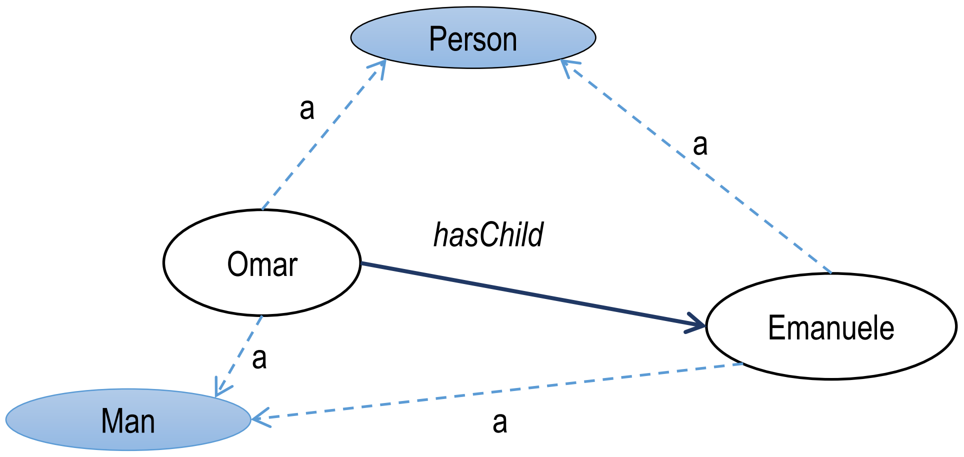


Negation

```
PREFIX : <http://laurenzi.ch#>
CONSTRUCT { ?s a :UnhappyMan }
WHERE {
    ?s a :Man .
    FILTER NOT EXISTS {?s :hasChild ?c}
}
```

- Use SPARQL for
  - Finding out who is an unhappy man. An unhappy man is a man who is NOT a father

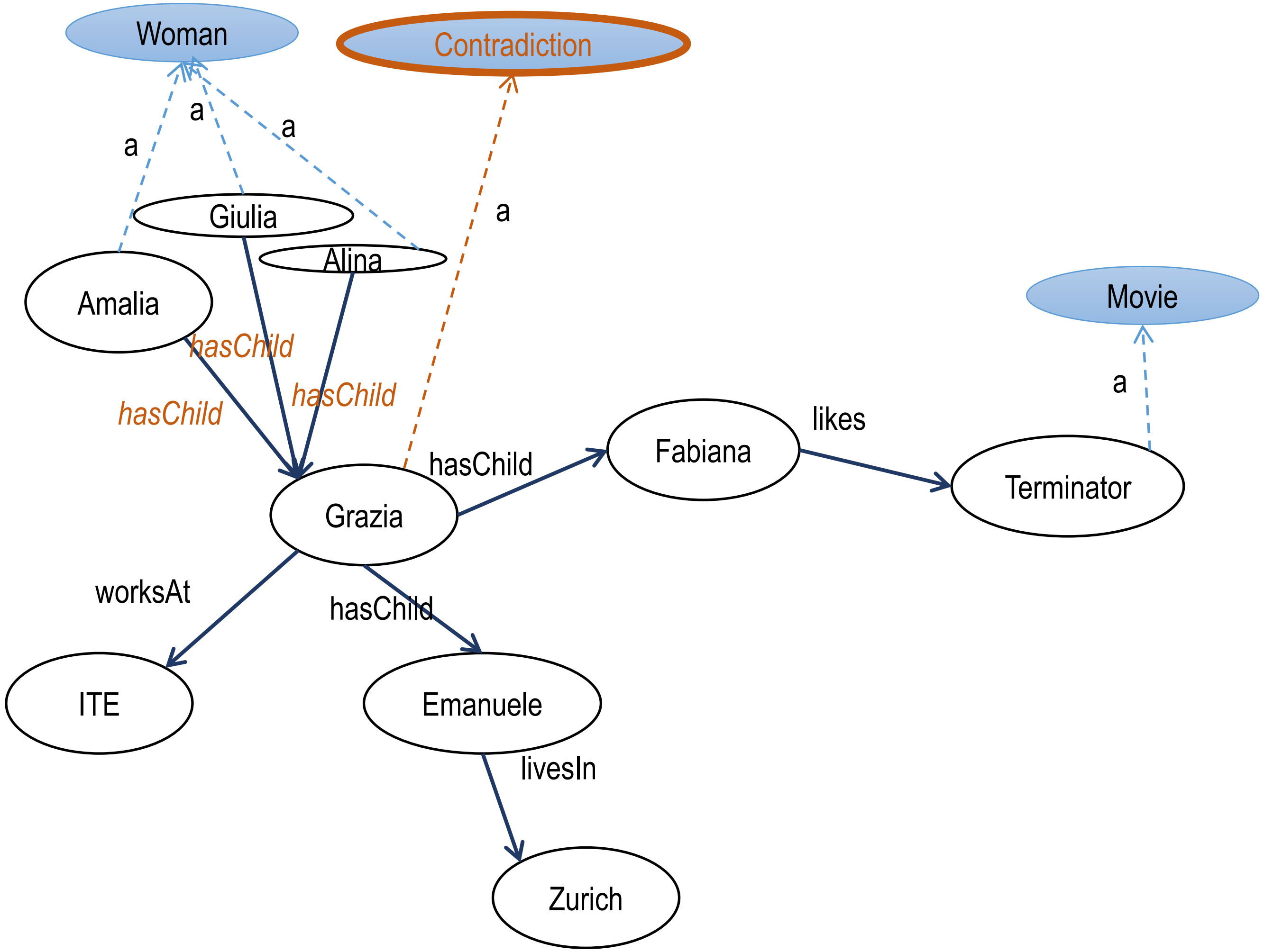
- Is Omar a Father?
- Is Omar an UnhappyMan?
- Is Emanuele a Father?
- Is Emanuele an UnhappyMan?



Integrity Constraint

- Use SPARQL for
  - Detecting contradictions

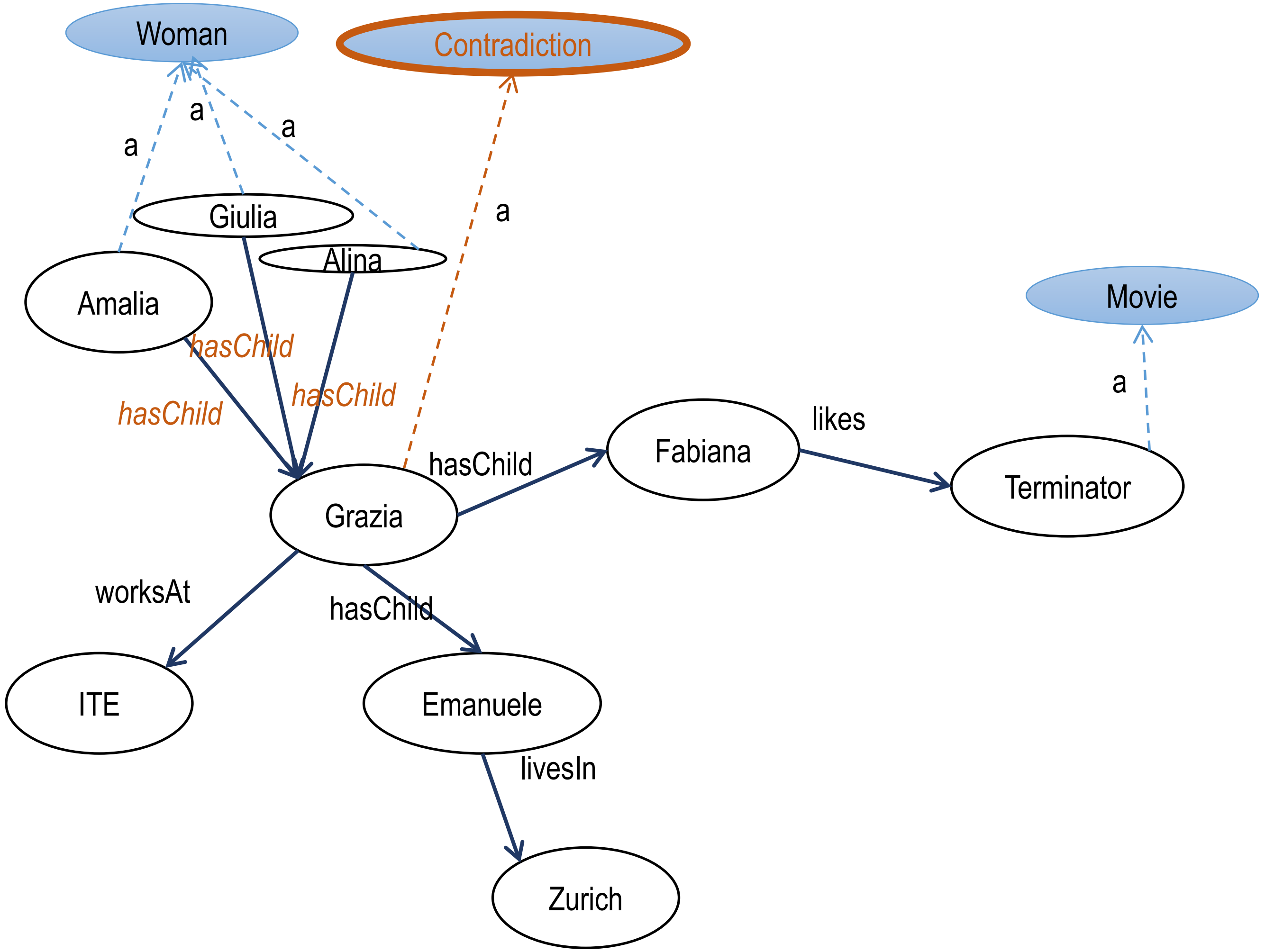
Contradiction pattern (a person with 3 mothers!)



Class exercise

– Use SPARQL for
  – Detecting contradictions
– Find the Turtle file with contradictions on Wiki.
– Create and test the rule with GraphBD
– Be ready to present.

Contradiction pattern (a person with 3 mothers!)



SWRL

A semantic rule language for rule-based reasoning

SWRL

– SWRL = "Semantic Web Rule Language"
– Example

```
hasParent(?C,?P) ∧ hasBrother(?P,?U) -> hasUncle(?C,?U)
```

– SWRL-Rule are similar to PROLOG rules:
  – Several conditions (in predicate notation), separated by “^” (AND)
  – A consequence (also in predicate notation)
  – Variables starts with “?” (like in SPARQL)
– Conditions and consequence are divided by “->” (minus and greater sign)

http://www.w3.org/Submission/SWRL/

Examples for Rules

– Classification

```
Man(?m) -> Person(?m)
```

– Inverse relationships

```
hasChild(?p,?c) -> hasParent(?c,?p)
```

Examples for Rules

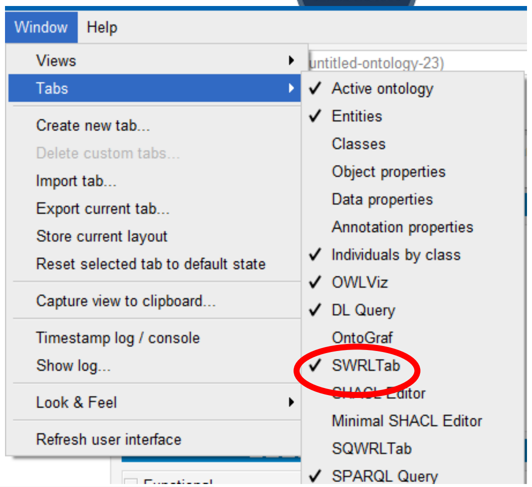
– Assigning values to properties

```
hasParent(?C,?P) ∧ hasBrother(?P,?U) -> hasUncle(?C,?U)
hasParent(?C,?P) ∧ hasSister(?P,?A) -> hasAunt(?C,?A)
```

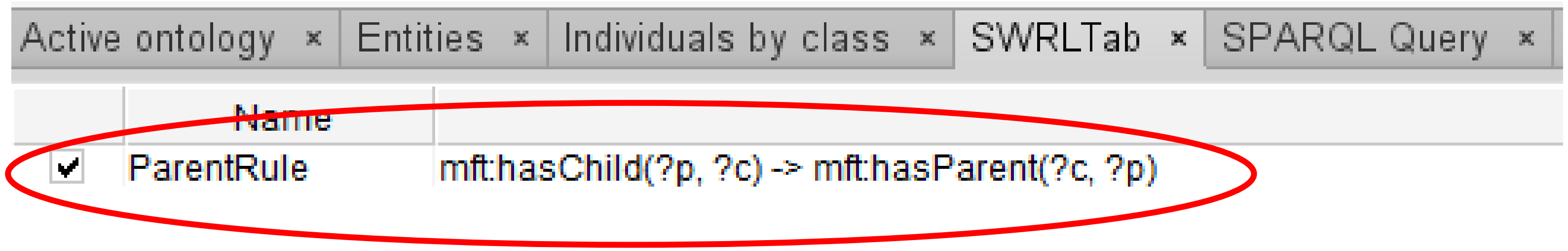
– Rules with Literals and operators

```
Person(?p) ^ hasAge(?p,?age) ^ swrlb:greaterThan(?age,17) ->
   Adult(?p)
Person(?p) ^ hasNumber(?p,?n) ^ swrlb:startsWith(?n, "+") ->
   hasInternationalNumber(?p, true)
```

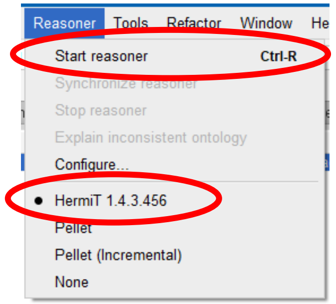
Rules in Protege



- In Protege there is the SWRLTab
- In this tab one can write rules



- to execute rules, one has to start a reasoner
  - select the reasoner (e.g. HermiT or Pellet) and
  - click on "Start reasoner"

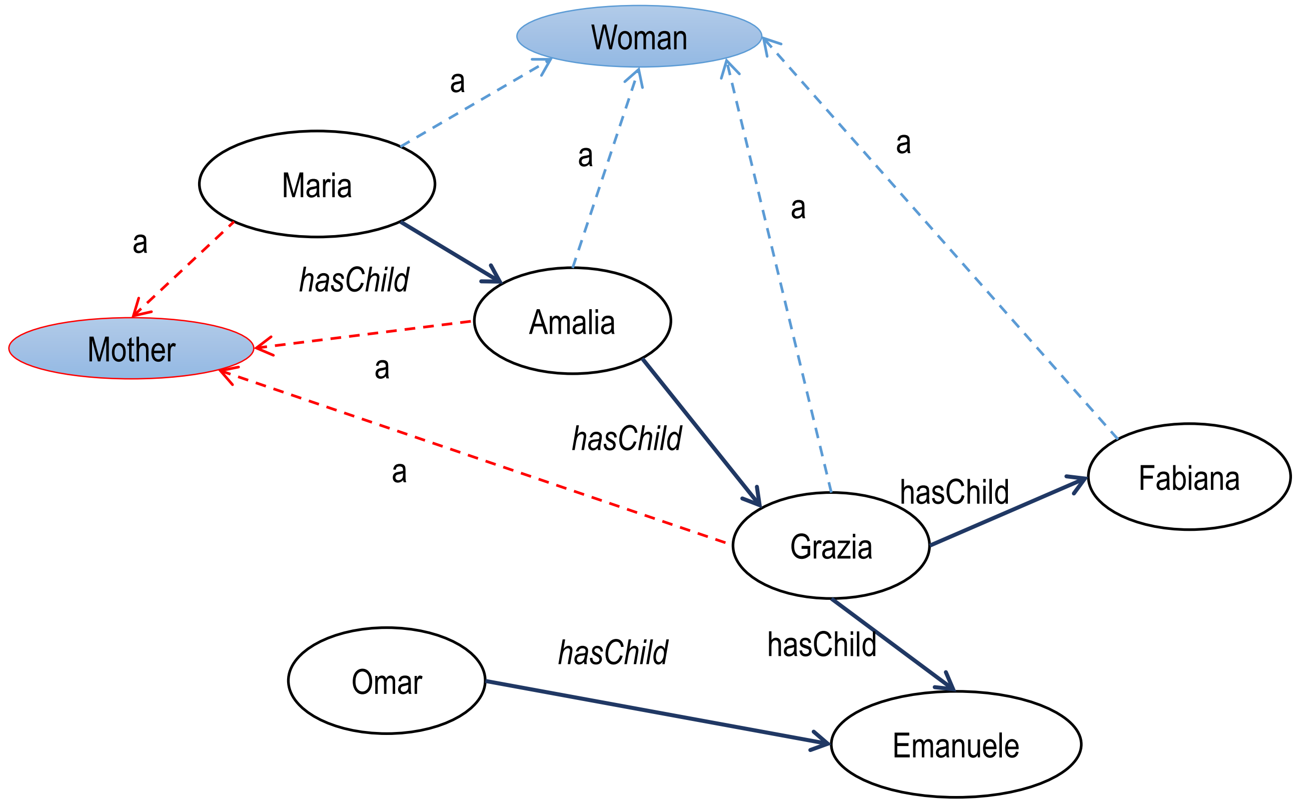


Use of some operators of SWRL for machine reasoning

```
Woman(?W) ^ hasChild(?W, ?C) -> Mother(?W)
```

Conjunction (AND)

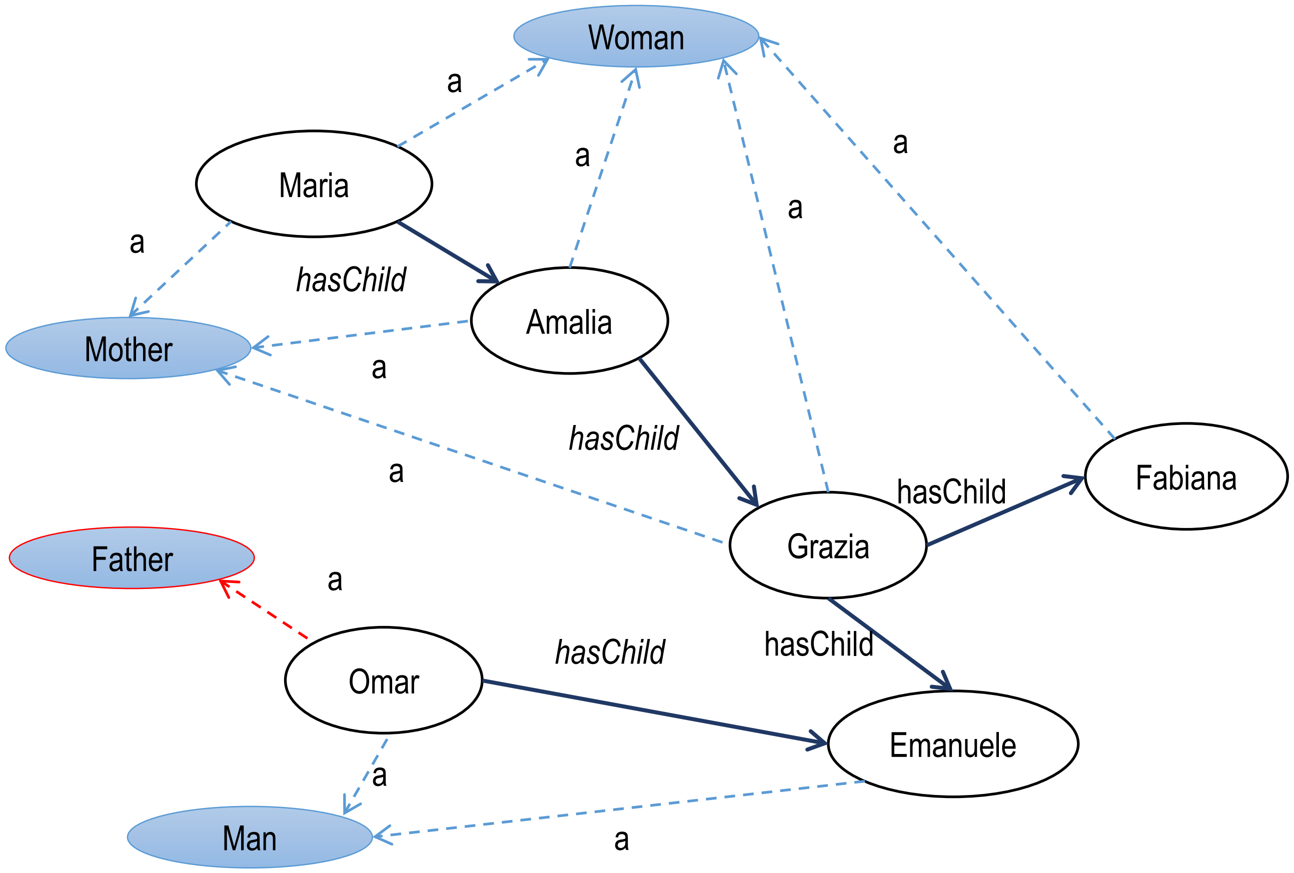
– Use SWRL for
  – Adding those Women to Mother which have a child



```
Man(?X) ^ hasChild(?X, ?C) -> Father(?X)
```

Conjunction (AND)

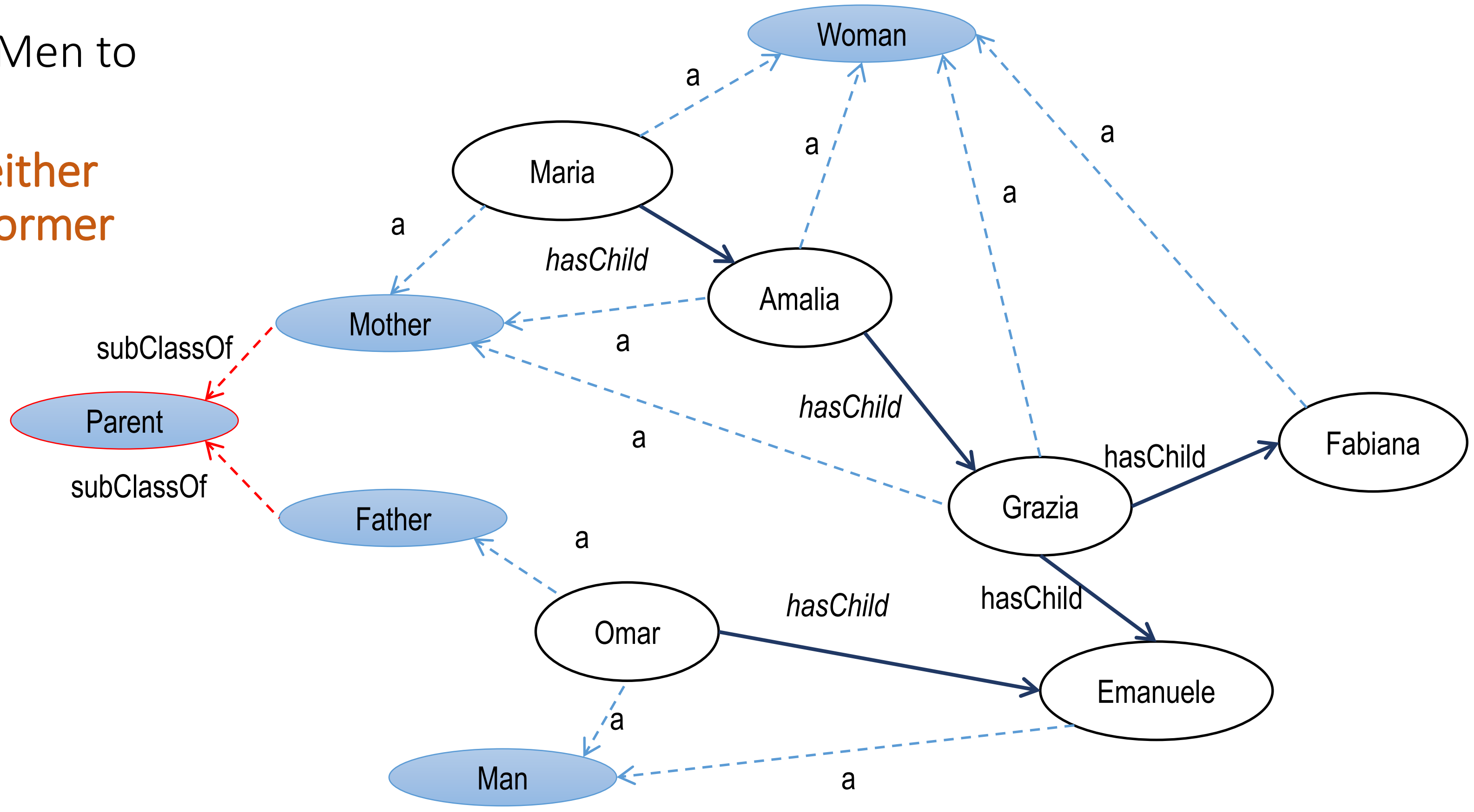
- Use SWRL for
  - Adding those Men to Father which have a child



```
Father(?F) -> Parent(?F)
Mother(?M) -> Parent(?M)
```

Disjunction (OR)

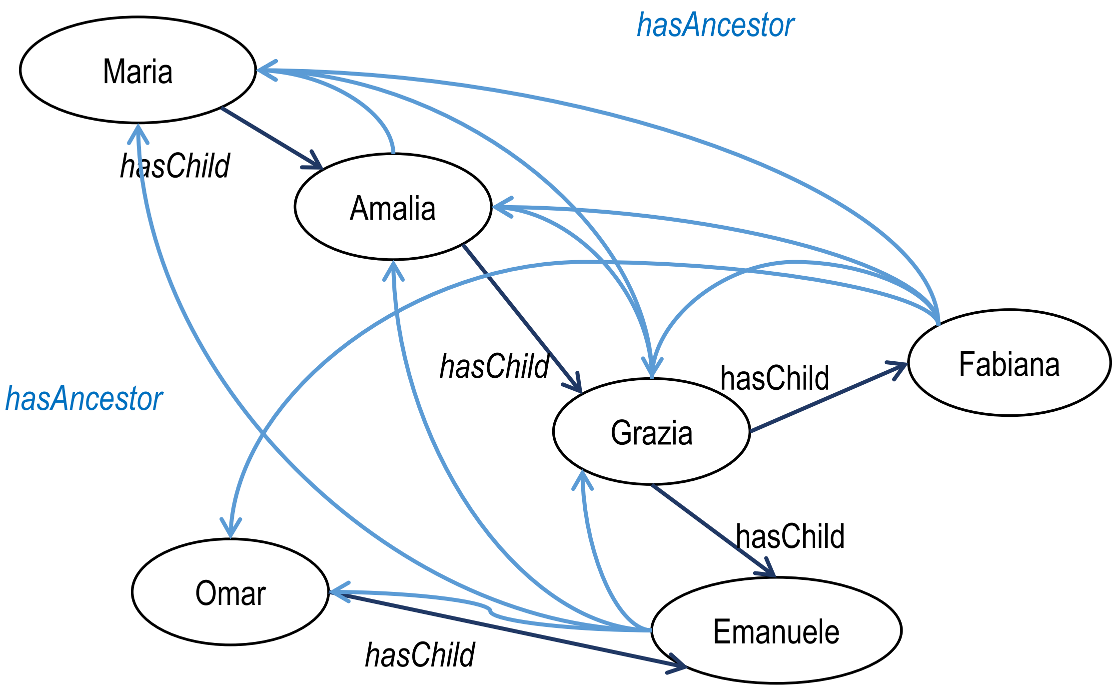
- Use SWRL for
  - Adding those Women and Men to Parent which have a child
  - Warning: You don't have neither RDFS nor the results from former queries



```
hasChild(?P, ?C) -> hasAncestor(?C, ?P)
hasChild(?A, ?X) ^ hasAncestor(?D, ?X) -> hasAncestor(?D, ?A)
```

Recursion

- Use SWRL for
  - Adding the relationships hasAncestor which is either someone which has a child or someone which has a child whose is already an ancestor



Negation in SWRL = Negation in OWL (extended Version of RDFS)

- SWRL does NOT support negation …
- … but OWL (underlying language under SWRL)
  - OWL is an extension of RDFS
  - OWL allows negation
- Two examples for OWL axioms

```
Father ≡ Man and (hasChild some Person)
```

```
UnhappyMan ≡ Man and (not Father)
```

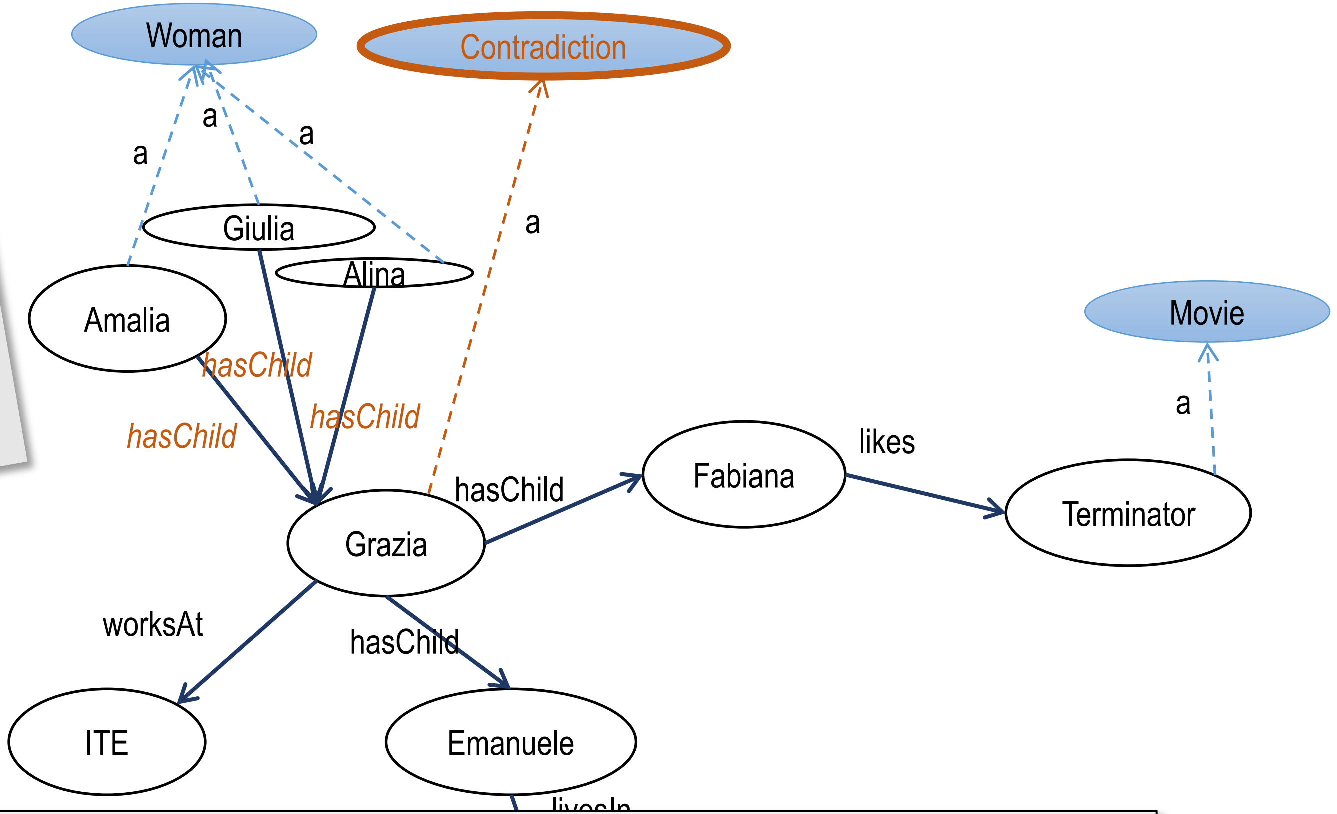
Integrity Constraint

– Use SWRL for
  – Detecting contradictions

However, is Amalia different from Giulia?
And Guilia different from Alina?
Amalia different from Alina?

Do you have PROOF of it?

Contradiction pattern (a person with 3 mothers!)

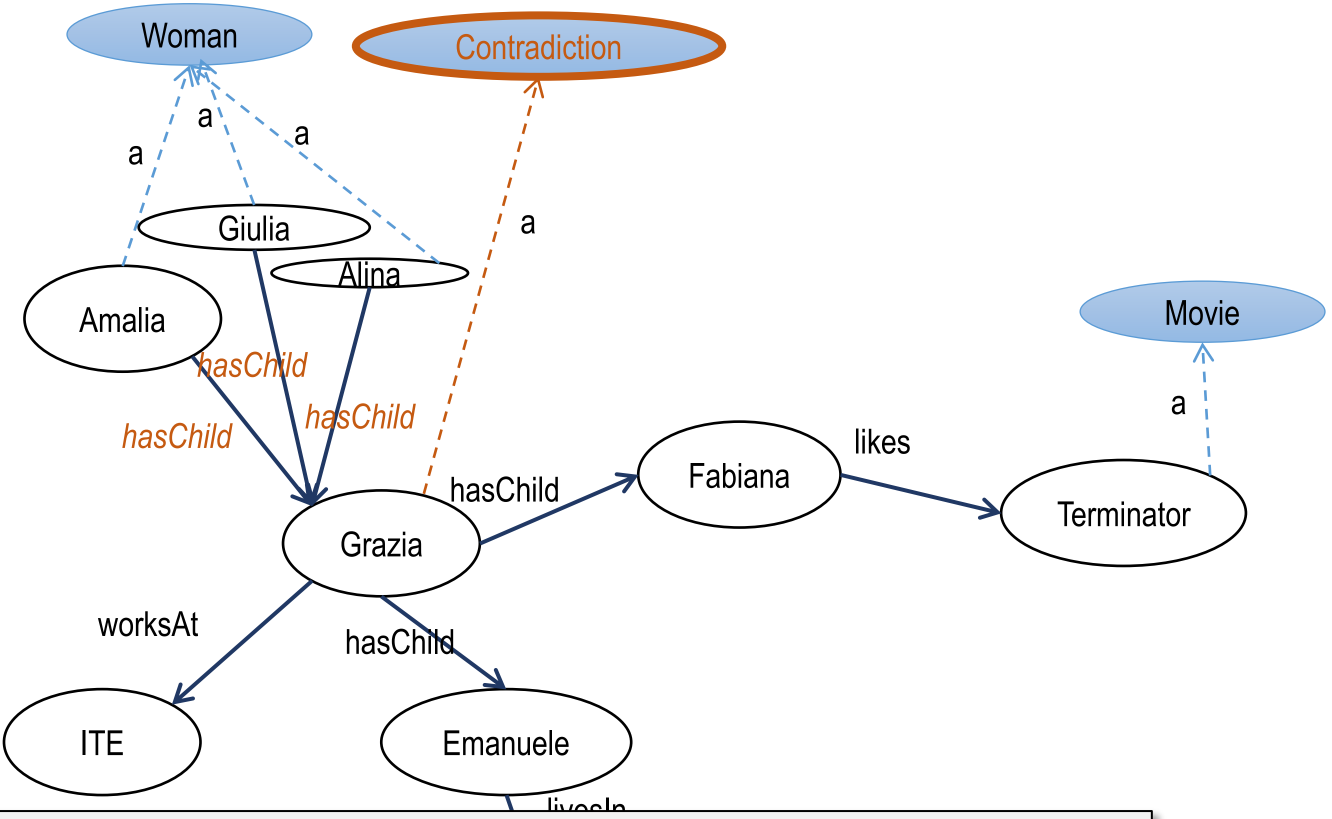



```
hasChild(?B, ?A) ^ hasChild(?C, ?A) ^ hasChild(?D, ?A) ^
differentFrom(?B, ?C) ^ differentFrom(?B, ?D) ^ differentFrom(?C, ?D) ->
Contradiction(?A)
```

No Unique Name Assumption

- Usually a Open World Assumption excludes also a Unique Name Assumption
- Amalia and Alina might be the same "individual" represented by two different names
- You need to specify explicitly that they are different!

Contradiction pattern (a person with 3 mothers!)




```
hasChild(?B, ?A) ^ hasChild(?C, ?A) ^ hasChild(?D, ?A) ^
differentFrom(?B, ?C) ^ differentFrom(?B, ?D) ^ differentFrom(?C, ?D) ->
Contradiction(?A)
```

```
DifferentIndividuals: Amalia, Giulia, Alina
```

SHACL

A W3C standard to validate RDF graphs

Shape Constraint Language - SHACL

- W3C recommendation since 20 July 2017 - https://www.w3.org/TR/shacl/
- RDF language
- Created to allow validation of RDF
- A “schema” language for RDF
- SHACL defines a “Shapes Graph” that is used to validate the “Data Graph”.

Example

Shapes Graph

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .

schema:Person
  a rdfs:Class, sh:NodeShape ;
  sh:property
    [
      sh:path schema:name ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
    ],
    [
      sh:path schema:age ;
      sh:minCount 1 ;
      sh:minInclusive 18 ;
    ] ;
.
```

Data Graph

```
{
  "@context": {
    "@base": "https://example.com/",
    "@vocab": "http://schema.org/"
  },
  "@id": "John-Doe",
  "@type": "Person",
  "name": [
    "John",
    "Johnny"
  ],
  "age": 18
}
```

Validation Report

Success

No

Errors found

- https://example.com/John-Doe:
  - schema:name:
    - More than 1 values

https://shacl-playground.zazuko.com/

University of Applied Sciences and Arts Northwestern Switzerland
School of Business

Example

Shapes Graph

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix schema: <http://schema.org/> .

schema:Person
  a rdfs:Class, sh:NodeShape ;
  sh:property
    [
      sh:path schema:name ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
    ],
    [
      sh:path schema:age ;
      sh:minCount 1 ;
      sh:minInclusive 18 ;
    ] ;
.
```

Data Graph

```
{
  "@context": {
    "@base": "https://example.com/",
    "@vocab": "http://schema.org/"
  },
  "@id": "John-Doe",
  "@type": "Person",
  "name": [
    "John",
    "Johnny"
  ],
  "age": 18
}
```

Validation Report

Success

No

Errors found

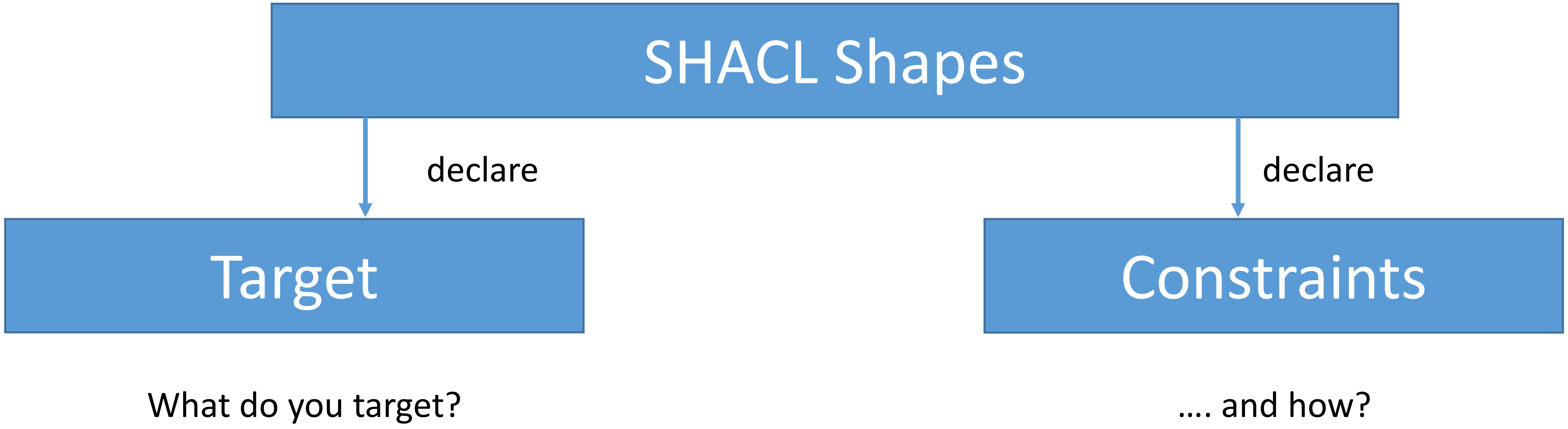
- https://example.com/John-Doe:
  - schema:name:
    - More than 1 values

https://shacl-playground.zazuko.com/

SHACL Processor

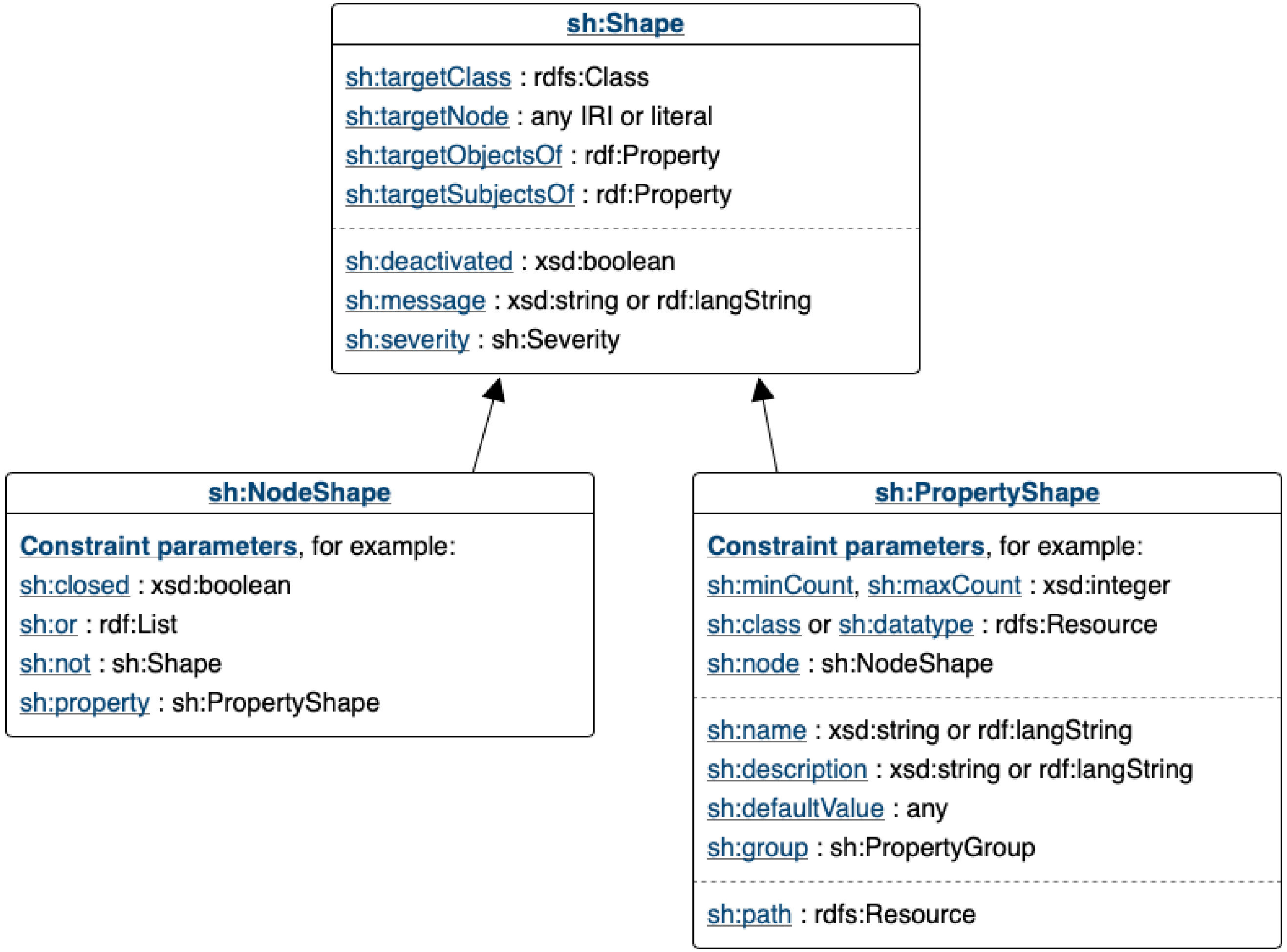
- Two inputs:
  - a **data graph** (validation target)
  - a **shapes graph** (how to validate);
- SHACL processors must not change the graphs, i.e., both data and shapes graphs at the end of the validation must be identical to the graph at the beginning of validation
- Generates a results graph
- There are: SHACL Core processor and SHACL SPARQL processor.
  - SHACL Core processors support validation with the SHACL Core Language
  - SHACL-SPARQL processors support validation with the SHACL-SPARQL Language

SHACL Core Language – A Semplified View



SHAPE

- A shape is a collection of target and constraints
  - **Targets:** define which nodes in the data graph must conform to the shape.
  - **Constraint:** define how to validate a node.
- Sh:Shape
  - Sh:NodeShape
    - Specify constraints on the target nodes (classes)
  - Sh:PopertyShape
    - Specify constraints on target properties and their values



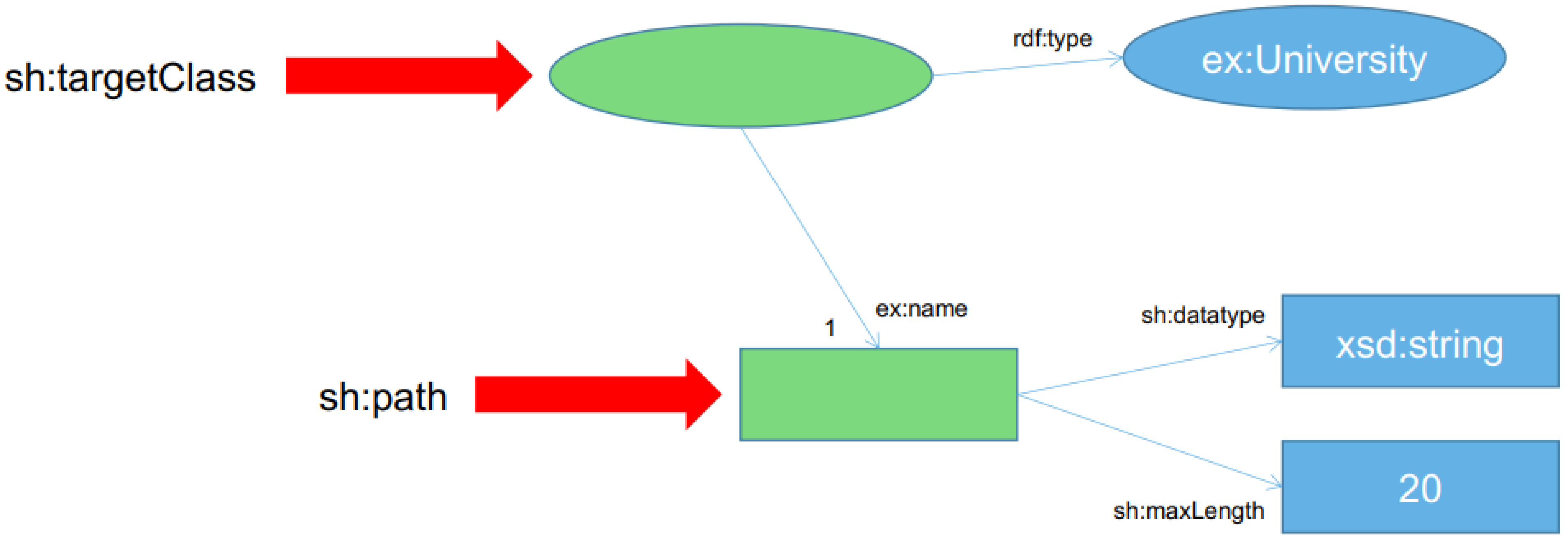
Example

Targets

Constraints

```
ex:UniversityShape a sh:NodeShape ;
    sh:targetClass ex:University ;
    sh:property [
        sh:path ex:name ;
        sh:minCount 1 ;
        sh:maxCount 1 ;
        sh:datatype xsd:string ;
        sh:maxLength 20 ;
    ] ;
```

Example

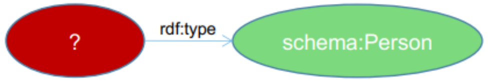

sh:targetClass
rdf:type
ex:University
ex:name
1
sh:datatype
xsd:string
sh:path
20
sh:maxLength

Targets

– Target declarations:

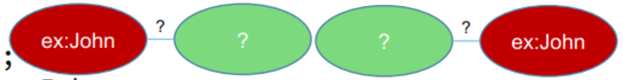
- sh:targetClass: targets all resources that are instances of a given class

```
:MyClassShape  a sh:NodeShape ;
               sh:targetClass schema:Person;
```



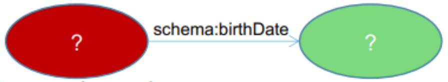
- sh:targetNode: targets a specific resource, e.g., a given instance

```
:MyNodeShape   a sh:NodeShape ;
               sh:targetNode ex:John;
```



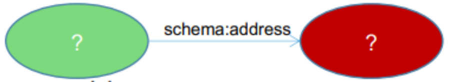
- sh:targetSubjectOf: targets all the subjects of a given predicate

```
:MySubjShape   a sh:NodeShape ;
               sh:targetSubjectsOf schema:birthDate;
```



- sh:targetObjectOf: targets all the objects of a given predicate

```
:MyObjShape    a sh:NodeShape ;
               sh:targetObjectsOf schema:address;
```



Core Constraints Components

Type	Constraints
Cardinality	minCount, maxCount
Types of values	class, datatype, nodeKind
Values	node, in, hasValue
Range of values	minInclusive, maxInclusive minExclusive, maxExclusive
String based	minLength, maxLength, pattern, languageIn, uniqueLang
Logical constraints	not, and, or, xone
Closed shapes	closed, ignoredProperties
Property pair constraints	equals, disjoint, lessThan, lessThanOrEquals
Non-validating constraints	name, description, group, order, defaultValue
Qualified shapes	qualifiedValueShape, qualifiedMinCount, qualifiedMaxCount

Cardinality Constraints

Constraint	Description
minCount	Restricts the minimun amount of occurences of a given property. Default value: 0
maxCount	Restricts the maximum amount of occurences of a given property. Default value: 0

```
:Person a sh:NodeShape, rdfs:Class ;
  sh:property [
    sh:path schema:knows ;
    sh:minCount 1;
    sh:maxCount 2;
  ] .
```

```
:john a schema:Person ;
      schema:knows :mary .

:mary a schema:Person ;
      schema:knows :peter ,
                   :anna ,
                   :fred .

:peter a schema:Person ;
      schema:givenName “Peter” .
```

Datatype of Values Constraints

Constraint	Description
datatype	Restricts the datatype of all value nodes to a given value

```
:Person a sh:NodeShape, rdfs:Class ;
  sh:property [
    sh:path schema:birthDate ;
    sh:datatype xsd:date;
  ] .
```

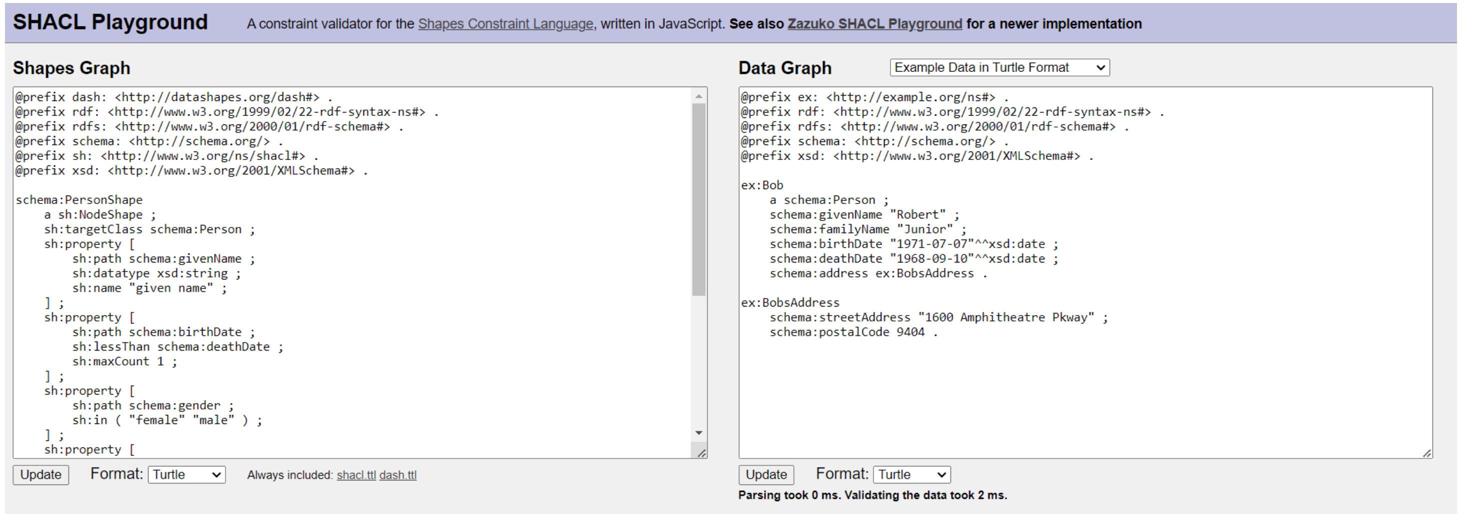
```
:john a schema:Person ;
     schema:birthDate “1990-05-01”^^xsd:date .

:mary a schema:Person ;
     schema:birthDate “Unknown”^^xsd:date .

:peter a schema:Person ;
     schema:birthDate 1995 .
```

Other Applications for SHACL

- Interface building,
  - e.g., using with DASH, a [Python framework for interactive wen applications];
- Data structure and semantics declaration (semantic data model specification)
- Code generation
- Data integration
- Rule-based inferencing

Useful resources for SHACL

- Online SHACL Validators
  - https://shacl.org/playground/
  - https://www.ida.liu.se/~robke04/SHACLTutorial/
- https://archive.topquadrant.com/technology/shacl/



Exercise

- Given the below shape, find a solution for the below Turtle file.
- Test it at https://www.ida.liu.se/~robke04/SHACLTutorial/

Data Graph Data 1

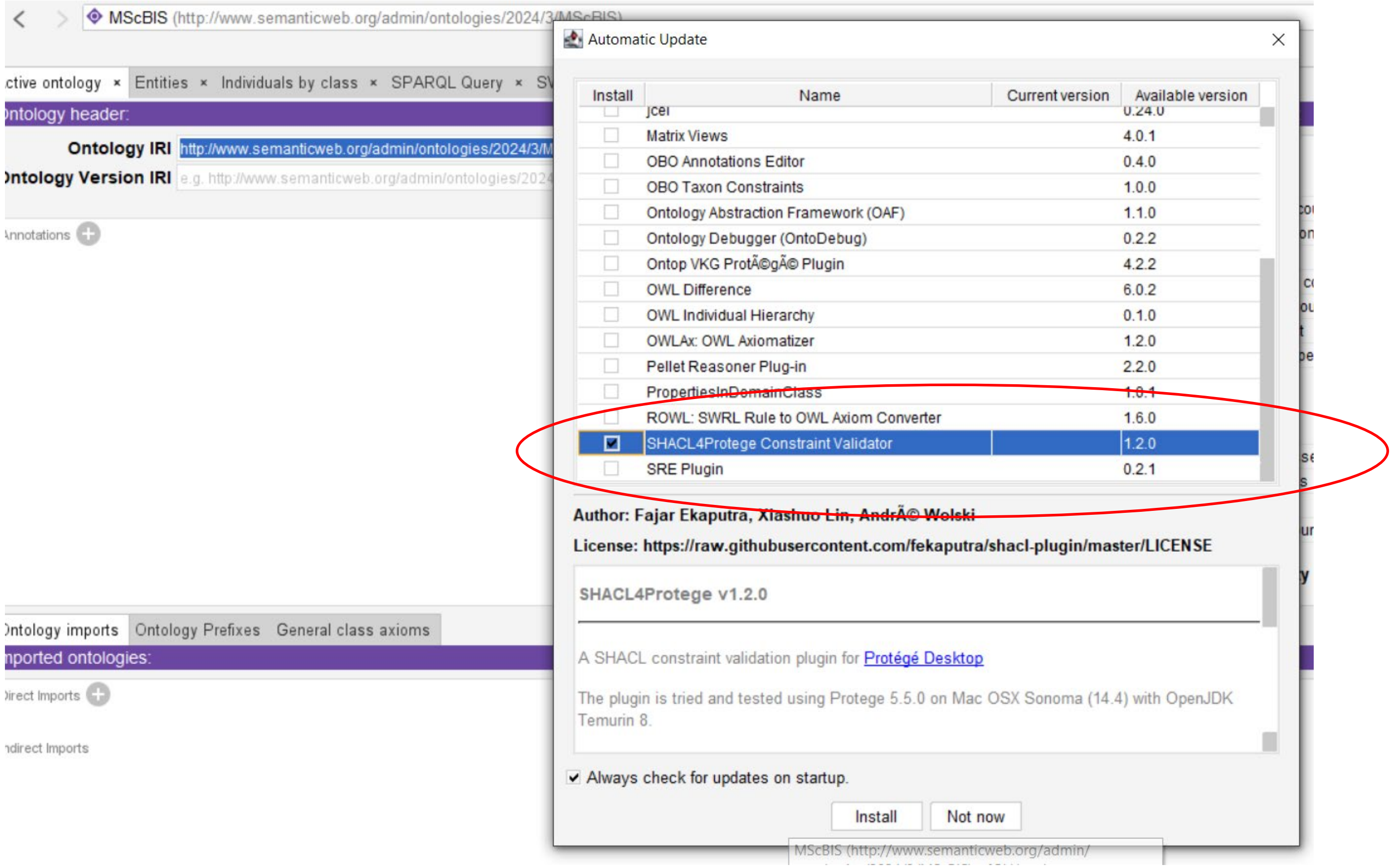
```
@prefix laureate: <http://data.nobelprize.org/resource/laureate/> .
@prefix xsd:   <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

laureate:935
    a                   foaf:Person ;
    foaf:birthday       "1948-10-09"^^xsd:date ;
    foaf:familyName     "Hart" ;
    foaf:givenName      "Oliver" ;
    foaf:name           "Oliver Hart" ;
    foaf:gender         "male" .
```

Shape Graph Shapes 1

```
@prefix ex: <http://example.org#> .
@prefix dash: <http://datashapes.org/dash#> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .

ex:PersonShape a sh:NodeShape ;
    sh:targetClass foaf:Person ;
    sh:property [
       sh:path foaf:birthday ;
       sh:datatype xsd:string ;
    ] .
```

SHACL in Protégé

- File>Check for plugins…
- Select and install SHACL4Protege Constraint Validator
- Window>Tabs>Minimal SHACL editor
- Now the new tab “Minimal SHACL editor” is visualized in Protégé



Homework

– Create a SHACL shape for your ontology and test it over Protégé.

Takeaways

- There exist languages maintained by the W3C that allow machine reasoning.
- Machine reasoning means applying reasoning services on knowledge graphs/ontologies that are expressed in some ontology language.
- The focus of this lecture:
  - SPARQL CONSTRUCT/INSERT
    - It enables deductive reasoning but does not allow for recursion. For this, an algorithm for the loop shall be implemented. Limitations with respect to a declarative rule-based approach.
  - SWRL
    - It enables deductive reasoning as a declarative knowledge base (rule-based system). Infinite loops are possible. Negation is not supported. Properties must be declared in advance for the respective values to be inferred, which is a behavior implemented in ontology editors like Protégé.
  - SHACL
    - It enables the validation of RDF graphs. It is coupled with RDF/RDF(S) and adds expressivity to lightweight ontology languages like RDF and RDF(S).