Combining Rules and Semantics in Drools

A Preliminary Study

Davide Sottara [dsotty@gmail.com]

[a]Department of Computer Science, Electronics and Systems
Faculty of Engineering, University of Bologna
Viale Risorgimento 2, 40100 Bologna (BO) Italy

Bologna/San Diego - April 19-23th, 2010

Outline

Ontologies

Ontology : *A formal specification of the terms in a domain*

- Capture knowledge about some domain of interest
- Describe the concepts in the domain
- State the relationships that hold between them
- List the individuals and their features

Outline

1. Introduction
   - Motivations
   - Modelling Knowledge
2. Integrated "Semantic Reasoning"
   - RDF
   - RDFS
   - Towards Description Logics
     - OWL-like Axioms
     - OWL-like Constructors
3. Embedding Semantics in Rules
4. Conclusions

Semantic Descriptions: Motivations

- To share common understanding of the structure of information among people or software agents
- To enable reuse of domain knowledge
- To separate domain knowledge from operational knowledge
- To analyse domain knowledge

Semantic Reasoning: Motivations

- Objects naturally fall into categories, possibly more than one...
- Categories (simple or complex) can be more general or specific than others...
- Objects have parts and relationships among them...

Semantic Reasoning: Motivations

- Objects naturally fall into categories, possibly more than one...
- Categories (simple or complex) can be more general or specific than others...
- Objects have parts and relationships among them...

So we would like...

- to define generalization relations
- to automatically infer generalization hierarchies from the provided descriptions
- to represent complex concepts by "composition" of simpler concepts
- to know if an individual belongs to some category or not

Semantic Rule-Based Reasoning : Motivations

- Descriptions still have some limitations
  - Capturing complex relations between properties
  - Capturing comples relations between individuals
- Adding Operative behaviour
  - we know that an individual belongs to some class: now what?

Outline

Formalisms

Many languages in different contexts, including:

Semantic Web

- Relational
  - RDF
  - RDF-S
- (Description) Logic-Based
  - OWL (Lite, DL, Full)
- Rule Integrations
  - SWRL

Challenges

- Languages need Reasoners to be useful
  - Complete
  - Correct
  - Efficient
  - Efficacious

Challenges

- Languages need Reasoners to be useful
  - Complete
  - Correct
  - Efficient
  - Efficacious

Drools and Semantics

Where does Drools stand?

- No support for "semantics" yet
- Some (relevant) limitations

What we want :

- Homogeneous Integration? (tightly coupled)
- Hybrid is also possible (loosely coupled)

Tight vs Loose Coupling

Hybrid

- Separated Rule and Semantic Engines
- Different languages with common points
- Rule Engine delegates the evaluation

Hybrid : Drools Example

- Custom Evaluator wrapper

Person(this isA Patient.class)

Homogeneous

- Single Rule/Semantic Engine
- Unique language with sufficient expressiveness
- Engine supports both types of reasoning

Homogeneous : Drools Example

- Native evaluation

later...

Tight vs Loose Coupling

Hybrid

- Separated Rule and Semantic Engines
- Different languages with common points
- Rule Engine delegates the evaluation

+ "Full" Expressiveness
+ Efficiency
– Interfacing
– KB alignment

Homogeneous

- Single Rule/Semantic Engine
- Unique language with sufficient expressiveness
- Engine supports both types of reasoning

+ Single component
+ Unified model
– Limited expressiveness ??
– Efficiency ??

Tight vs Loose Coupling

Homogeneous

- Many Potentialities
- Currently many Open Issues !!

Hybrid

Use the Wrapper Pattern (see my other talk...)

Tight vs Loose Coupling

Hybrid

Use the Wrapper Pattern (see my other talk...)

Homogeneous

- Many Potentialities
- Currently many Open Issues !!
- We'll see what can be (easily) done...

Tight vs Loose Coupling

Hybrid

Use the Wrapper Pattern (see my other talk...)

Homogeneous

- Many Potentialities
- Currently many Open Issues !!
- We'll see what can be (easily) done...
- And what can't be done (yet?)

Outline

RDF

Knowledge is encoded using "triples"

$$P(S,O)$$

reads (e.g.) *"S has property P with respect to O"*

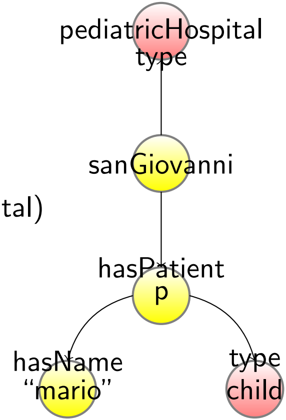
RDF

Also graphical notation

RDF Triples[a]

[a] Prolog-like, namespaces omitted

type(sanGiovanni,pediatricHospital)
hasPatient(sanGiovanni,p)
hasName(p, "mario")
type(p,child)



RDF vs Drools

Mapping triples on (dynamic) beans

- generics?
- automatic translation?

```
declare Property
@role(property)
@namespace(...)
  subject : Resource // Object
  object  : Resource // Object
end

declare PropertyValue
  pred    : Class<? extends Property>
  subject : Resource
  object  : Resource
end
```

RDF vs Drools

Equivalent representation:

```
rule "Triple 2 PropVal"
when
  $t : Property( $s: subject, $o : object)
then
  insert( new PropertyValue($t.class, $s, $o) );
end

rule "PropVal 2 Triple"
when
  PropertyValue( $p : pred, $s: subject, $o : object)
then
  insert( $p.newInstance($s, $o) );
end
```

RDF

Triples could be used in rules explicitly, possibly mixed with "usual" beans

```
rule "Visiting Parents"
when
  $c : Person( )    HasName($c,"mario")
  $h : Hospital( ) HasType($h, PediatricHospital.class)
  $p : Person(  )   HasChild($p,$c)
  $r : HasPatient($h,$p)
then
  insert( new Visits($p,$h) );
end
```

But this is just the beginning...

Outline

RDF Schema

RDF-S

Adds Schema information

- Entity/Class Relations
- Class/Class Relations

- Reason over and with *types*

RDF Schema

RDF-S

Adds Schema information

- Entity/Class Relations
- Class/Class Relations

- Reason over and with *types*
- Overcomes the `extends`/`instanceof` limitations

RDF Schema

RDF-S

Adds Schema information

- Entity/Class Relations
- Class/Class Relations

- Reason over and with *types*
- Overcomes the `extends`/`instanceof` limitations

Even in Drools:

```
// static type
when Patient( ... )

// dynamic type
when $p : Person()
     Type($r, Patient.class)
```

RDF Schema - Axioms

Provided a few relations are defined:

Schema Relations

Type	Resource $\times$ Class
Subject	Property $\times$ Resource
Object	Property $\times$ Resource
Predicate	Property $\times$ Class
Value	Resource $\times$ Resource
Domain	$\text{Class}_{\text{Property}}$ $\times$ Class
Range	$\text{Class}_{\text{Property}}$ $\times$ Class
SubClassOf	Class $\times$ Class
SubPropertyOf	$\text{Class}_{\text{Property}}$ $\times$ $\text{Class}_{\text{Property}}$
...	...

RDF Schema - Axioms

```
rule "DomainRange"
when
  $prop : SomeProperty( $subj , $obj )
  Domain( $prop.class , $dom )
  Range( $prop.class , $range )
then
  // from $prop definition:
  insert( new Type($subj ,$dom) );
  insert( new Type($obj , $range) );
end
```

RDF Schema - Axioms

```
rule "SubClassOf"
when
  Type( $x, $klass )
  SubClassOf( $klass, $super)
then
  insert( new Type($x, $super) );
end
```

$$Type(X, Patient), SubClassOf(Patient, Person) \Rightarrow Type(X, Person)$$

RDF Schema - Axioms

```
rule "SubPropertyOf"
when
  $p : Property( $s, $o )
  SubPropertyOf( $p.class, $super)
then
  insert( $super.newInstance($s, $o) );
end
```

RDF(S) : Considerations

RDFS just makes implicit type declarations explicit

- Expressiveness is limited
- So is inference

+ Simple: Drools supports it easily

Outline

1. Introduction
   - Motivations
   - Modelling Knowledge
2. Integrated "Semantic Reasoning"
   - RDF
   - RDFS
   - Towards Description Logics
     - OWL-like Axioms
     - OWL-like Constructors
3. Embedding Semantics in Rules
4. Conclusions

Description Logics

- Several logic(s) with different expressive power
  - Different features : F,E,U,C,S,H,R,O,I,N,Q, ...
- Different languages to encode them
  - OWL, KIF, ...

OWL-DL will be considered for reference

Uses of DL - Objectives

Define (complex) concepts - aka classes

Uses of DL - Objectives

Define (complex) concepts - aka classes
in terms of other classes and properties

Uses of DL - Queries

Subsumption

$C \subseteq D$?

- Is D a more general concept than C?

Satisfiability

$\exists x : x \in C$?

- Does C allow members?

Consistency

$\{...\} \Vdash \bot$?

- Does a set of facts lead to contradiction?

Instantiation

$\{...\} \Vdash x \in C$?

- Is x member of C given the available knowledge?

Description Logics

OWL defines axioms and class constructors:

Axioms

- subClassOf
- equivalentClass
- subPropertyOf
- equivalentProperty
- disjointWith
- sameAs
- differentFrom
- transitiveProperty
- inversefunctionalProperty
- symmetricProperty
- inverseOf

Constructors

- intersectionOf
- unionOf
- complementOf
- oneOf
- allValuesFrom
- someValuesFrom
- hasValue
- minCardinality
- maxCardinality

Drools Integration

Drools works with instances.

Drools Integration

Drools works with instances.

- *Instantiation* is (almost) immediate

Drools Integration

Drools works with instances.

- *Instantiation* is (almost) immediate

- *Subsumption* can be reduced to *Satisfiability*
- *Satisfiability* is still an open issue

Drools Integration

Drools works with instances.

- *Instantiation* is (almost) immediate

- *Subsumption* can be reduced to *Satisfiability*
- *Satisfiability* is still an open issue

Preliminary analysis:

- Tableau algorithms seem the most likely candidates
  - generative
- Still need some features (e.g. backtracking, *false* relations)
- We'll start from what can be done already

Outline

1. Introduction
   - Motivations
   - Modelling Knowledge
2. Integrated "Semantic Reasoning"
   - RDF
   - RDFS
   - Towards Description Logics
     - OWL-like Axioms
     - OWL-like Constructors
3. Embedding Semantics in Rules
4. Conclusions

Axioms: General Principles

Most axioms define features of Properties

- Meta-data specified using attributes
- Engine automatically inserts meta-facts
- Rule Bases automatically include meta-rules

subClassOf

Same as in RDFS, but...

```
declare Patient
@role(entity)
@subclass(Person)
@subclass(...)
end
```

Attribute `@subclass` inserts
`SubclassOf(Patient.class,Person.class)`

subClassOf

Same as in RDFS, but...

```
declare Patient
@role(entity)
@subclass(Person)
@subclass(...)
end
```

Attribute `@subclass` inserts
`SubclassOf(Patient.class,Person.class)`
Here hierarchy is declared, but not inferred

subPropertyOf

As for `SubClassOf`:

```
declare HasSon
@role(property)
@subproperty(HasChild)
end
```

Attribute `@subproperty` inserts
`SubPropertyOf(HasSon.class,HasChild.class)`
same as before - but DL do not entail subproperty relations!

Class/Property Equivalence

Two more Attributes:

- `@equivalentClass()`
- `@equivalentProperty()`

Syntactic sugar: $C \equiv D \Leftrightarrow (C \rightarrow D \wedge D \rightarrow C)$

Class/Property Equivalence

Two more Attributes:

- `@equivalentClass()`
- `@equivalentProperty()`

Syntactic sugar: $C \equiv D \Leftrightarrow (C \rightarrow D \wedge D \rightarrow C)$

- ... but also $(C \rightarrow D \wedge \neg C \rightarrow \neg D)$
  remeber/see the imperfect case?

Disjoint

```
declare Male
@role(entity)
@disjointWith(Female)
end
```

The attribute controls the insertion of an instance of the relation:

```
declare DisjointWith
@role(property)
@symmetric
subject : Class
object : Class
end
```

Disjoint

```
rule "Disjoint" // not in standard Drools...
when
  Type($x, $klass)
  DisjointWith($klass, $anotherKlass)
then
  insert( new Type($x,$anotherKlass, FALSE));
end
```

$Type(X, Male) \Rightarrow \neg Type(X, Female)$

(In)Equalities

Two relations, to be specified on an individual basis

```
declare Equals
@role(property)
@symmetric
@transitive
  subject : Resource
  object  : Resource
end

declare DifferentFrom
@role(property)
@symmetric
  subject : Resource
  object  : Resource
end
```

Transitivity

The relation attribute `@transitive` allows to compute closures:

```
declare Transitive
@role(property)
  subject : Class<? extends Property>
  object  : boolean
end

rule "Closure"
when
  PropertyValue( $p, $x, $y )
  PropertyValue( $p, $y, $z )
  Transitive($p, true)
then
  insert( $p.newInstance($x,$z) );
end
```

$$Relative(X, Y), Relative(Y, Z) \Rightarrow Relative(X, Z)$$

Symmetry

The relation attribute `@symmetric` inverts roles:

```
declare Symmetric
@role(property)
  subject : Class<? extends Property>
  object  : boolean
end

rule "Symmetry"
when
  $prop : PropertyValue( $p, $x, $y )
  Symmetric($p, true)
then
  insert( $p.newInstance($y,$x) );
end
```

$Relative(X, Y) \Rightarrow Relative(Y, X)$

Functionality

Functionality (resp. inverse-functional) properties are decorated using the attributes `@functional` and `@invFunctional`

```
rule "Functionality" //resp. inverse
when
  PropertyValue( $p, $x, $y )
  // as per object identity
  PropertyValue( $p, $x, $z != $y )
  Functional($p, true)
then
  insert( new SameAs($y,$z) );
end
```

HasFather(X, "john"), HasFather(X, "mrWhite") $\Rightarrow$
SameAs("john", "mrWhite")

Inverse

The relation attribute `@inverse` allows:

```
declare Inverse
@role(property)
@symmetric
  subject : Class<? extends Property>
  object  : Class<? extends Property>
end

rule "Inverse"
when
  PropertyValue( $p, $x, $y )
  Inverse($p, $q)
then
  insert( $q.newInstance($y,$x) );
end
```

$HasFather(X, Y) \Rightarrow FatherOf(Y, X)$

Outline

Constructors : General Principles

Constructors become specialized rule-like patterns

```
declare Klass
@restriction($x)(      // target variable
  // Patterns here
  $x : Resource( ... )    // binding
  ...                     // definition
)
end
```

Automatically inserts `Type(x,Klass.class)`

Intersection

$$C_1 \wedge \cdots \wedge C_n \longrightarrow K$$

```
rule "Intersect"
when
  $x : Resource()
  Type($x,C1.class)
  ...
  Type($x,Cn.class)
then
  insert( new Type($x, K.class) );
end
```

Union

$$C_1 \vee \cdots \vee C_n \longrightarrow K$$

```
rule "Union"
when
 $x : Resource()
  Type($x,C1.class)
  ...
  or Type($x,Cn.class)
then
  insert( new Type($x, K.class) );
end
```

Complement

$C \rightarrow \neg K$

See @disjointWith

```
rule "Complement"
when
  $x : Resource()
   Type($x, $c : C.class)
then
   insert( new Type($x, K.class, FALSE) );
end
```

OneOf

$\{e_1, \ldots, e_n\} \subseteq K$

```
rule "One of Many"
when //one rule for each individual
  $x : Resource(...) // e_x
then
  insert( new Type($x, K.class) );
end
```

All Values from

$$\forall(P(X,Y) \wedge C(Y)) \rightarrow K(X)$$

```
rule "AllValues"
when
  $x : Resource()
  $k : Class(...) //$k may be a "literal"
  forall ( SomeProperty($x,$y)
           Type($y,$k) )
then
  insert( new Type($x,K.class) );
end
```

Some Values from

$$\exists(P(X, Y) \land C(Y)) \rightarrow K(X)$$

```
rule "SomeValues"
when
  $x : Resource()
  $k : Class(...) //$k may be a "literal"
  exists ( SomeProperty($x,$y)
           Type($y,$k) )
then
  insert( new Type($x,K.class) );
end
```

Cardinality of Values

$$(|P(X,Y)| \lesseqgtr n) \rightarrow K(X)$$

```
rule "Cardinality"
when
  $x : Resource()
  Collection( size == N ) // also > or <
    from collect ( SomeProperty($x,$y) )
then
  insert( new Type($x,K.class) );
end
```

On Constructors

- So far, more like class constraints
    - Still useful in practice!

– Not quite like DL reasoners

- Necessary (but not sufficient) : reverse constructors

Intersection (reverse)

$$C_1 \wedge \cdots \wedge C_n \leftarrow K$$

```
rule "IntersectRev"
when
   $t : Type($x,K.class)
        not (Type($x, C1.class)
         Type($x, C2.class))
then
   insert( new Type($x, C1.class) );
   ...
   insert( new Type($x, Cn.class) );
end
```

Union (reverse)

Non-deterministic : requires new features!

$$C_1 \vee \cdots \vee C_n \leftarrow K$$

```
rule "UnionRev"
when
  $t : Type($x,K.class)
  not (Type($x, C1.class))
  not (Type($x, C2.class))
then
  insertBackTrack(
    new Type($x, C1.class),
    new Type($x, Cn.class) );
end
```

All Values from (reverse)

$$\forall(P(X,Y) \wedge C(Y)) \leftarrow K(X)$$

```
rule "AllValues"
when
  Type($x,K.class)
  $p : SomeProperty($x,$y)
  not ( Type($y,C.class) )
then
  insert( new Type($y,C.class) );
end
```

Some Values from (reverse)

$$\exists(P(X,Y) \wedge C(Y)) \leftarrow K(X)$$

```
rule "SomeValues"
when
   $t : Type($x,K.class)
   not ( SomeProperty($x,$y)
         Type($y,C.class) )
then
   Resource o = new Blank();
   insert( new SomeProperty($x,o) );
   insert( new Type(o,C.class) );
end
```

Cardinality (reverse)

$$(|P(X,Y)| \lesseqgtr n) \leftarrow K(X)$$

```
rule "Prop Cardinality = N"
// assuming should be = N
when
  $t : Type($x,K.class)
  $c : Collection( $s : size < N )
    from collect ( SomeProperty($x,$y) )
then
  for (int j : 0..(N-$s)) {
    Resource y = new Blank();
    insert( new SomeProperty($x,y) );
  }
end
```

Conclusions (so far)

- \+ A subset of DL can be built on top of Drools natively
- \+ More features will be added
- − Notation is still verbose

Ideas...

"On the fly" class declaration + rule

```
rule "No Fever"
when
  $p : Patient()
  forall ( HasRecord($p,$r)
           HasTemperature($r,$t)
           LessOrEqual($t,37) // celsius
         )
then
  // ...
end
```

Ideas...

Goal:

```
rule "No Fever"
when
  Patient( hasRecord[].hasTemp all lessOrEqual 37 )
then
  // ...
end
```

Tighter Integration - Proposals

Property role /1

- Properties as "virtual fields"

Patient(type Senior.class,
hasRecord[].hasTemp all lessOrEqual 37)

$P(S, O) \Leftrightarrow S.P \ni O$

- Query mode : $\exists X : p(s, X)$?
- "Fields" are set-valued unless properties are functional

Tighter Integration - Proposals

Property role /1

- Properties as "virtual fields"
- Properties can be navigated

Patient(type Senior.class,
hasRecord[].hasTemp all lessOrEqual 37)

$P(S, O) \Leftrightarrow S.P \ni O$

- Query mode : $\exists X : p(s, X)$?
- "Fields" are set-valued unless properties are functional

Tighter Integration - Proposals

Property role /1

- Properties as "virtual fields"
- Properties can be navigated
- "Fields" need not be declared at compile time

Patient(type Senior.class,
hasRecord[].hasTemp all lessOrEqual 37)

$P(S, O) \Leftrightarrow S.P \ni O$

- Query mode : $\exists X : p(s, X)$?
- "Fields" are set-valued unless properties are functional

Tighter Integration - Proposals

Property role /2

- Properties as restrictions

Patient(type Senior.class,
hasRecord[].hasTemp all lessOrEqual 37)

- Evaluation mode : $p(s, o)$?

  iterates over all records

Tighter Integration - Proposals

Quantifier role

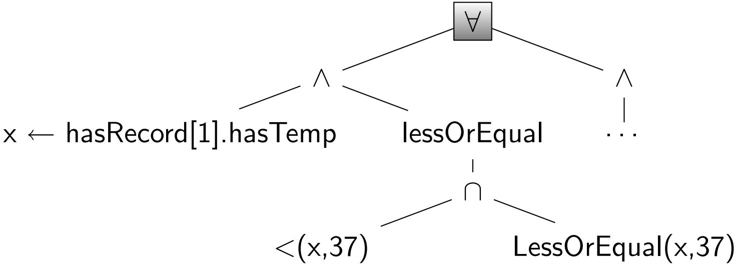
- Need quantifiers in constraints

Patient(type Senior.class,
hasRecord[].hasTemp all lessOrEqual 37)

Patterns:

- getProperty **all** evalProperty object
- getProperty **only** evalProperty object
  - implicit: maxCard=1, minCard=1
- getProperty **some** evalProperty object
  - implicit: minCard=1
- getProperty **some** @[max="", min=""] evalProperty object
  - explicit maxCard and/or minCard

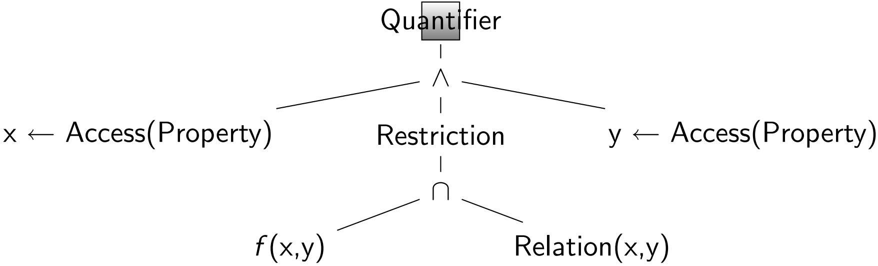
Tighter Integration - Logic Structure



Tighter Integration - Logic Structure

In general:

- Left and right operands are accessed (recursively)
- Every possible pair is tested
  - Using a direct evaluator
  - Using asserted relations
- Behaviour is conditioned by quantifier
- Natural extension for uncertainty



On Implementation

Two main points:

- Dynamic fields
- Node behaviour

And questions (just to cite some):

- Field mapping
  - what if $P(S, O)$ is in the WM, but S is not?
- Should triples always be kept explicitly in WM ?

Conclusions

- Compact syntax is more Drools-like
- Comparable expressiveness with explicit triples
- Dynamic types and fields overcome the problem of static declarations
- Need improvements on language and engine
- Implementation and Efficiency to be tested
- Better architecture for uncertain reasoning...