Logic and Constraint Programming

5- Rule-based systems

A.A. 2021/2022

Lorenzo Rossi

lorenzo.rossi@unicam.it

University of Camerino

Rule-based systems

RULE-BASED SYSTEMS

- Rules are the **main way to express knowledge** in many fields of I.A.
- Most common rules are:
  - production rules (eg.: Drools)
  - logic programs (eg.: Prolog)
- They are similar, but **realized in a dual way**

Rule-based systems

- Modus Ponens:

$$\frac{\langle p(x), p(X) \rightarrow q(Y) \rangle}{q(y)}$$

if it holds that p(X) implies q(Y) and p(x) holds, then q(y) holds

Es.: If it rains, then the street is wet.
Here it rains.
Then, here the street is wet.

Rule-based systems

- Modus Ponens:

$$\frac{\langle p(x), p(X) \rightarrow q(Y) \rangle}{q(y)}$$

if it holds that p(X) implies q(Y) and p(x) holds, then q(y) holds

Es.: If it rains, then the street is wet.	implication
Here it rains.	premise
Then, here the street is wet.	conclusion

Rule-based systems

Production rules

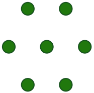
- Forward-chaining
- The facts activate rules that generate new facts
- Pattern matching
- Parallelism

Logic programs

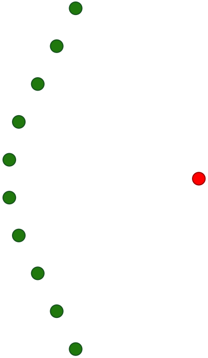
- Backward-chaining
- From goal to facts, applying rules in a backward way
- Unification
- Backtracking

RULE-BASED SYSTEMS

Production rules

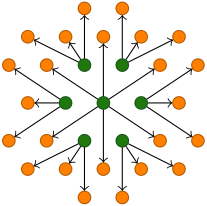


Logic programs

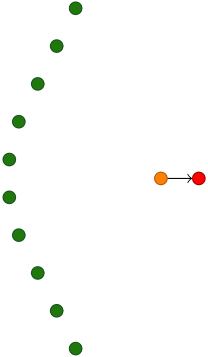


RULE-BASED SYSTEMS

Production rules

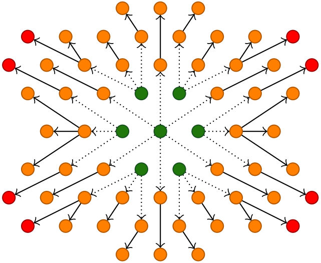


Logic programs

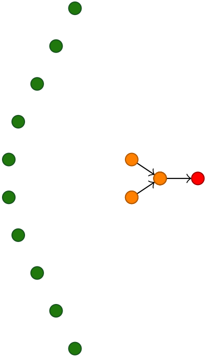


Rule-based systems

Production rules

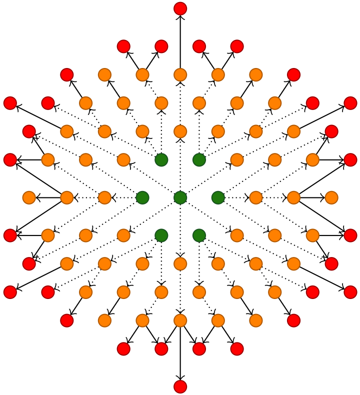


Logic programs

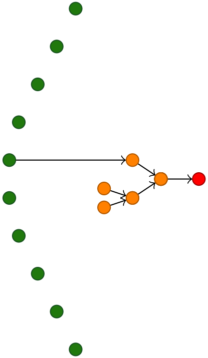


Rule-based systems

Production rules

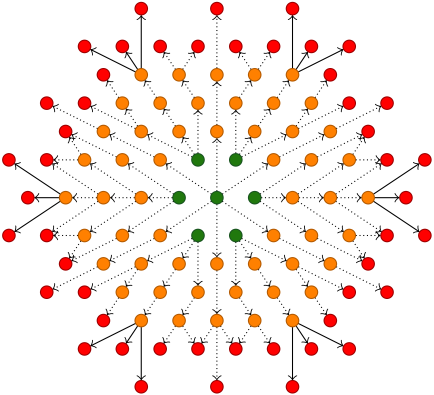


Logic programs

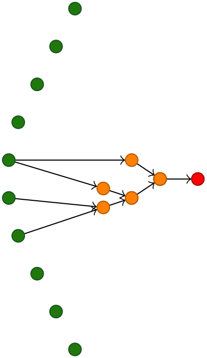


Rule-based systems

Production rules

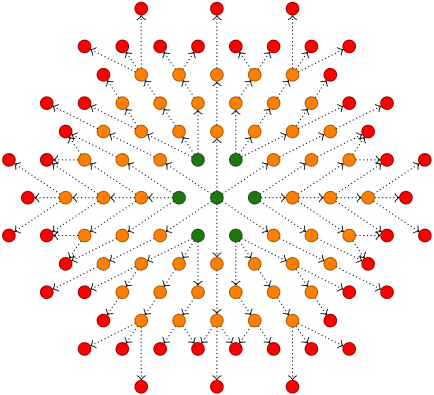


Logic programs

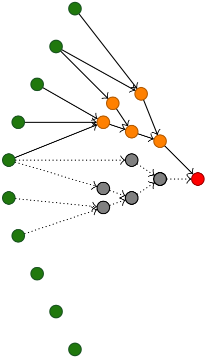


RULE-BASED SYSTEMS

Production rules

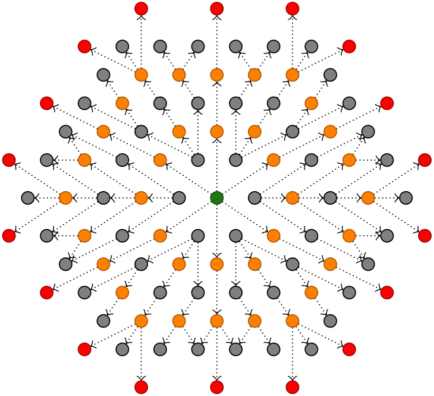


Logic programs

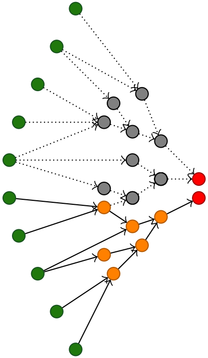


RULE-BASED SYSTEMS

Production rules



Logic programs



PRODUCTION RULE SYSTEMS

- Production Rule Systems (PRS):
  - are Rule Based Systems (RBS),
  - are based on the Modus Ponens principle,
  - rely on a reactive/generative approach

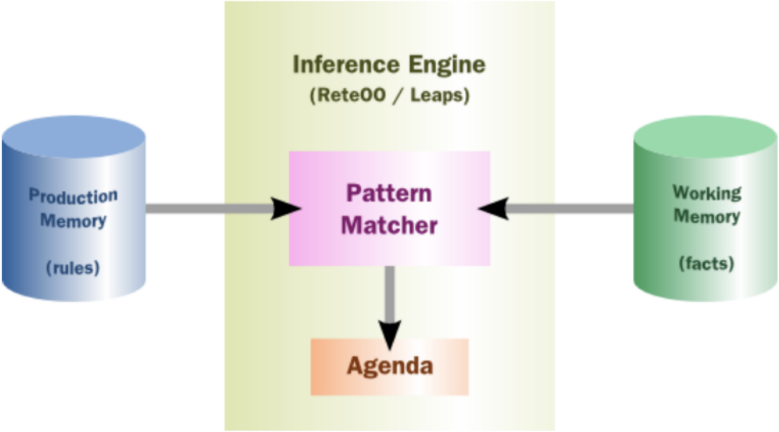
When a PRS is a right choice?

- The problem is too complex for traditional coding approaches: rules provide a **more abstract view**, preventing fragile implementations
- The problem is **not fully known**
- **Flexibility**, when system logic changes often over time
- Domain knowledge readily available

Example scenario

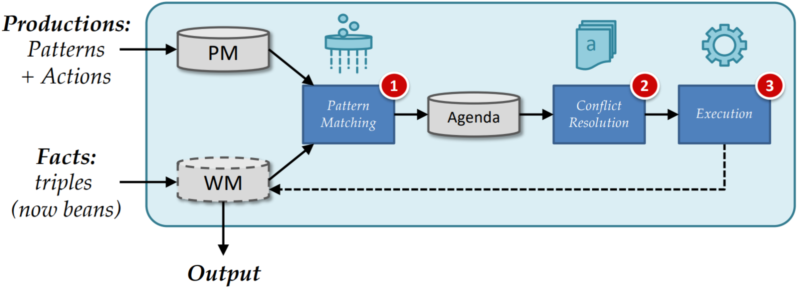
PRODUCTION RULE SYSTEMS

Architecture and working schema



Production Rule Systems

Architecture and working schema

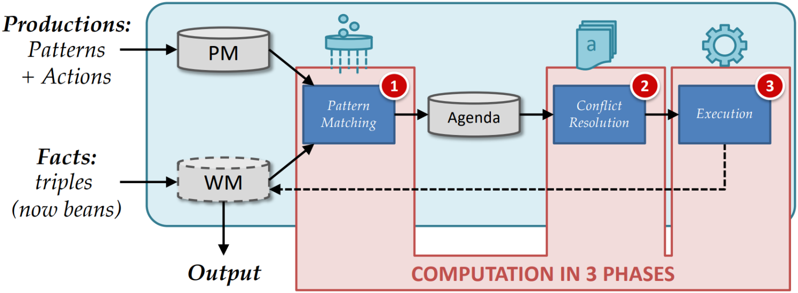


Production Rule Systems

- Rules are stored in the **Production Memory** (PM)
- Facts are stored in the **Working Memory** (WM), where they can be changed or retracted
- **Inference engine** applies to data in the WM the rules in in the PM to deduce new information
- The **Agenda** deals with the execution order in case of conflicts, using conflict resolution strategies

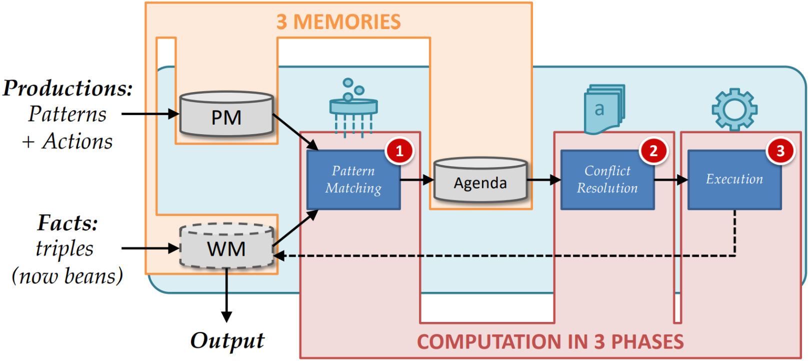
Production Rule Systems

Architecture and working schema



Production Rule Systems

Architecture and working schema

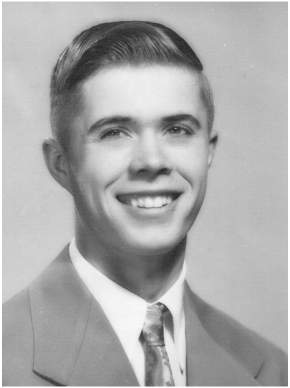


Pattern Matching: RETE algorithm

RETE ALGORITHM

RETE is a **pattern matching** algorithm for implementing **rule-based systems**.

The Rete algorithm was designed by *Charles L. Forgy* of Carnegie Mellon University, first published in a working paper in 1974, and later elaborated in his 1979 Ph.D. thesis and a 1982 paper.



RETE NETWORK

The **Rete network** is the *brain* behinf the Rete algorithm

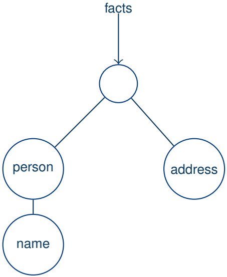
It is made of **nodes** that each hold a list of objects that **satisfy some associated condition**

The original Rete algorithm worked out of **facts**, while commercial engines have evolved to be object-oriented nowadays

Rete network

≫Alfa nodes

The discrimination tree starts with **Alfa nodes**



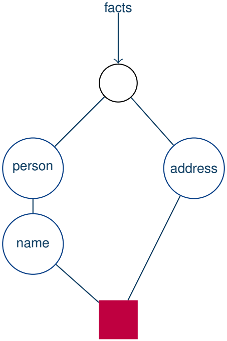
Alfa nodes are created for each fact, then **attributes** are appended

Each node represents an additional test to the series of **conditions** applied upstream

Rete network

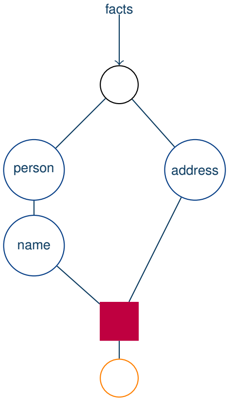
≫Beta nodes

Nodes are then connected across facts into **Beta nodes**



Those nodes combine the list of facts that verify conditions on one branch with the list of facts that verify the conditions on another branch

RETE NETWORK

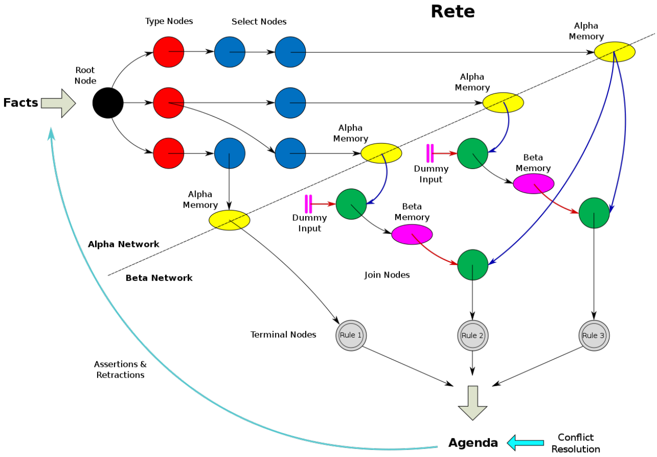


The path eventually ends with the **action part** of the rule

The content of the actions is irrelevant for the Rete network

RETE ALGORITHM

≫RETE NETWORK



Rete00

- The Rete implementation used in Drools is called ReteOO
- It is an enhanced and optimized implementation of the Rete algorithm specifically for **object-oriented systems**

RETE00
≫OBJECT TYPE NODE

When using ReteOO, the root node is where all objects (facts) enter the network. From there, it immediately goes to the **ObjectTypeNode**.

The ObjectTypeNode helps to reduce the workload of the rules engine. To make things efficient, the ObjectTypeNode is used so that the engine only passes objects to the nodes that **match the object's type**

An inserted object retrieves a list of valid ObjectTypesNodes through a lookup in a HashMap from the object's class. If this list does not exist, it scans all the ObjectTypeNodes to find valid matches

RETE00

≫ALFANODES

AlfaNodes are used to evaluate literal conditions. When a rule has multiple literal conditions for a single object type, they are linked together. E.g., if an application asserts an object, it must first satisfy the first literal condition before it can proceed to the next AlfaNode

AlfaNodes are propagated using ObjectTypeNodes. Each time an AlfaNode is added to an ObjectTypeNode, it adds the literal value as a key to the **HashMap** with the AlfaNode as the value.

RETE00

≫ALFANODES

When a new instance enters the ObjectType node, rather than propagating to each AlfaNode, **it retrieves the correct AlfaNode from the HashMap**. This avoids unnecessary literal checks.

When facts enter from one side, you may do a **hash lookup** returning potentially valid candidates (referred to as indexing). At any point a valid join is found, the Tuple joins with the Object (referred to as a partial match) and then propagates to the next node.

Rete00

≫BetaNodes

BetaNodes are used to compare two objects and their fields. The objects may be of the same or different types

Alfa memory refers to the left input on a BetaNode. Beta memory is the term used to refer to the right input of a BetaNode

When facts enter from one side, if a valid join is found, the object (referred to as a partial match) and then propagates to the next node

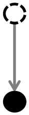
RETE00

≫TERMINALNODES

Terminal nodes are used to indicate when a single rule matches all its conditions (that is, the rule has a full match). A rule with an OR conditional disjunctive connective results in a sub-rule generation for each possible logical branch. Because of this, one rule can have **multiple terminal nodes**

Rete00 Examples

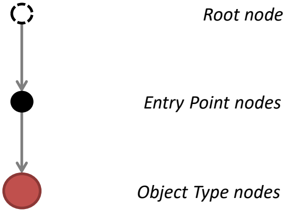
```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

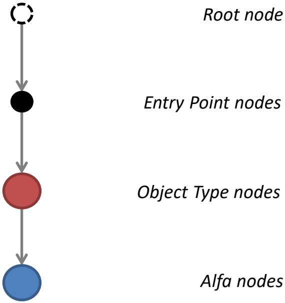


Root node

Entry Point nodes

Object Type nodes

```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```



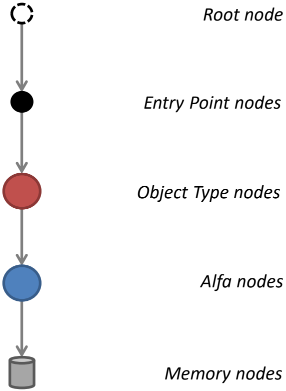
Root node

Entry Point nodes

Object Type nodes

Alfa nodes

```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```



Root node

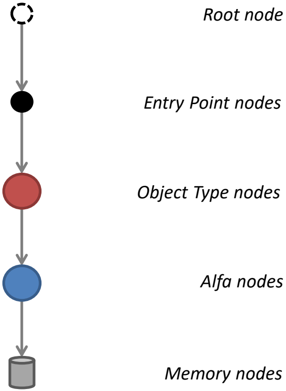
Entry Point nodes

Object Type nodes

Alfa nodes

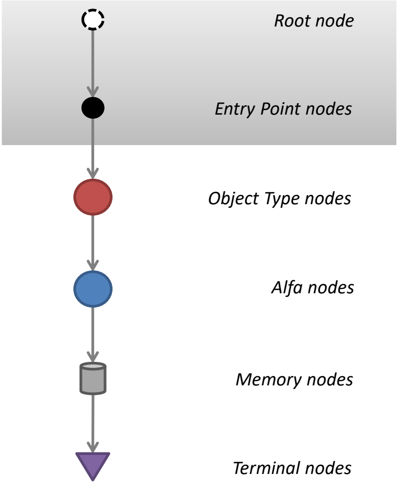
Memory nodes

```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```



NB: facts in a (Alfa) Memory Node match with a simple pattern!

```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

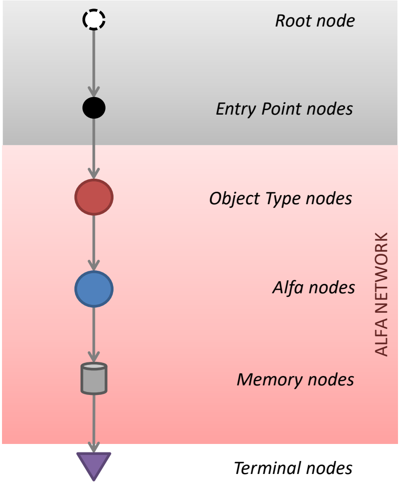
Object Type nodes

Alfa nodes

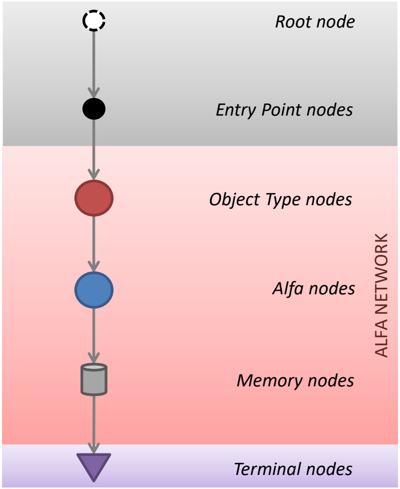
Memory nodes

Terminal nodes

```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

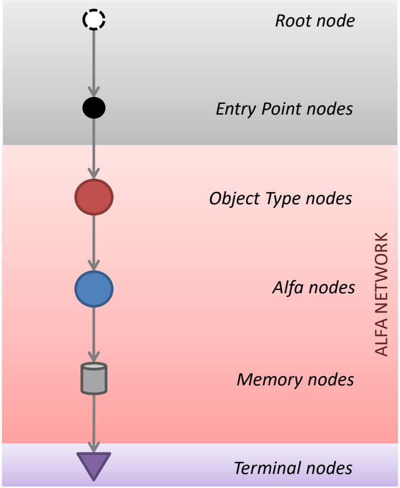


```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```



```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

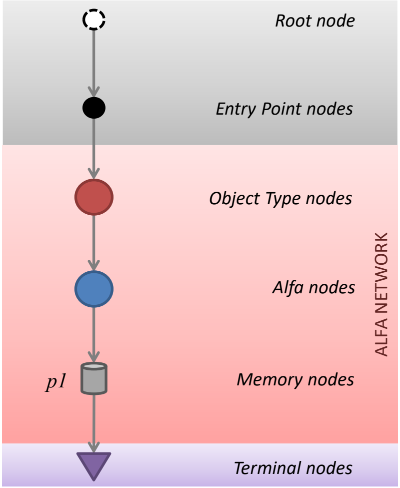
```
_
```



```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

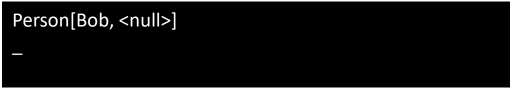
p1: Person("Bob", null)

```
Person[Bob, <null>]
_
```

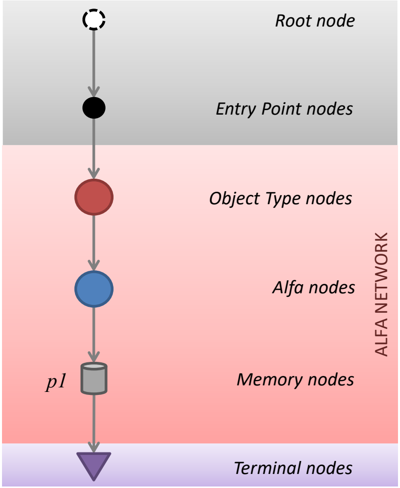


```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

p1: Person("Bob", null)
a1: Address("Via Po 2", 40068,
"San Lazzaro")



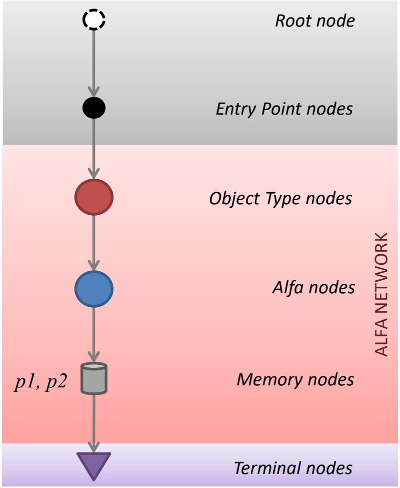
```
Person[Bob, <null>]
_
```



```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

p1: Person("Bob", null)
a1: Address("Via Po 2", 40068,
"San Lazzaro")
p2: Person("Bob", a1)

```
Person[Bob, <null>]
Person[Bob, Address[Via Po 2, 40068, San Lazzaro]]
_
```

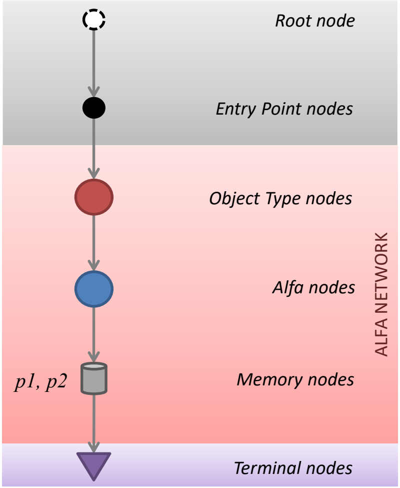


```
rule "Find Bobs"
when
  $p: Person( name=="Bob" )
then
  System.out.println($p);
end
```

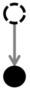
WM

p1: Person("Bob", null)
a1: Address("Via Po 2", 40068,
"San Lazzaro")
p2: Person("Bob", a1)
p3: Person("Frank", a1)

```
Person[Bob, <null>]
Person[Bob, Address[Via Po 2, 40068, San Lazzaro]]
_
```



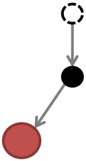
```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

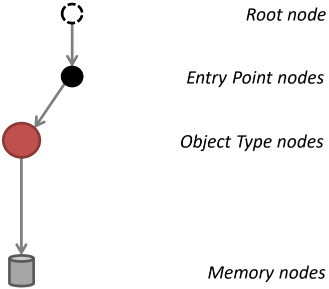


Root node

Entry Point nodes

Object Type nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



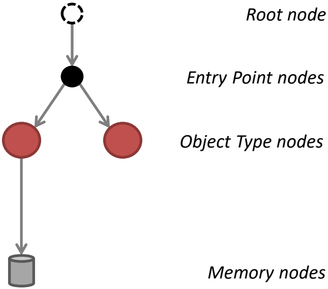
Root node

Entry Point nodes

Object Type nodes

Memory nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



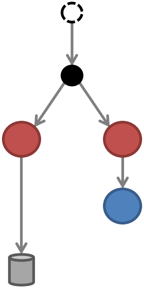
Root node

Entry Point nodes

Object Type nodes

Memory nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

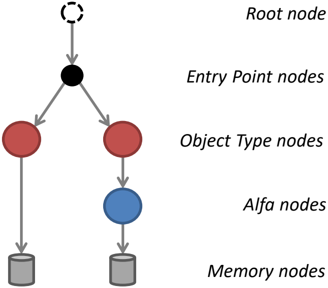
Entry Point nodes

Object Type nodes

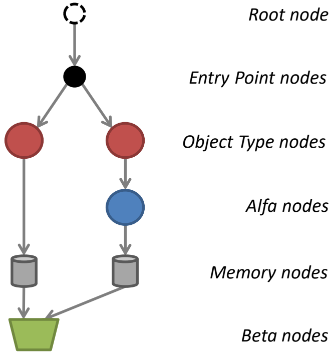
Alfa nodes

Memory nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

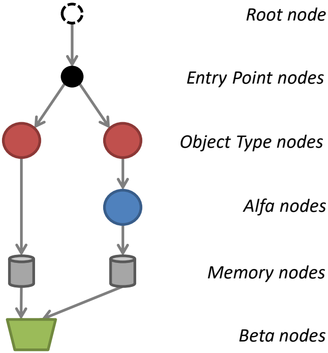
Object Type nodes

Alfa nodes

Memory nodes

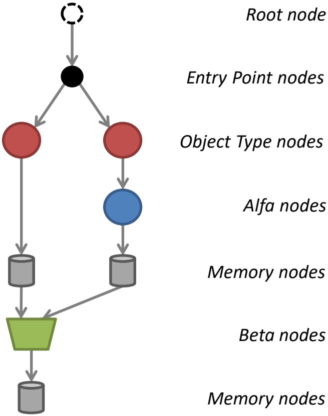
Beta nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



NB: Beta Nodes make cartesian product of objects filtered by Alfa father!

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

Object Type nodes

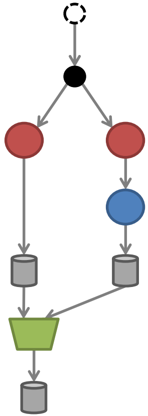
Alfa nodes

Memory nodes

Beta nodes

Memory nodes

```
rule "Find Bobs and addresses"
when
   $a: Address()
   $p: Person( name=="Bob" )
then
   System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

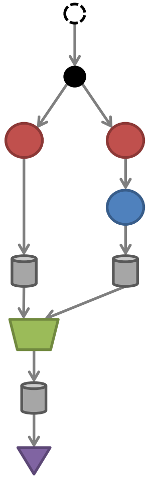
Memory nodes

Beta nodes

Memory nodes

NB: tuple in a(Beta) Memory Node match with a composite pattern!

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

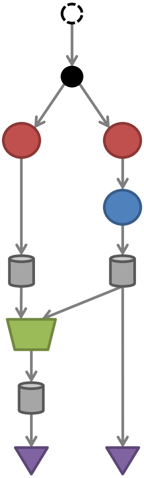
Memory nodes

Beta nodes

Memory nodes

Terminal nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

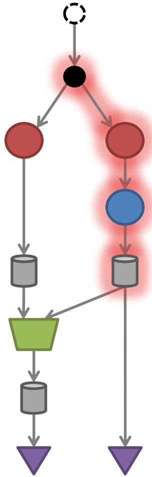
Memory nodes

Beta nodes

Memory nodes

Terminal nodes

```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

Memory nodes

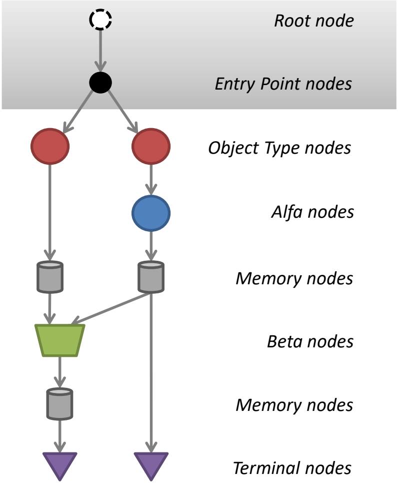
Beta nodes

Memory nodes

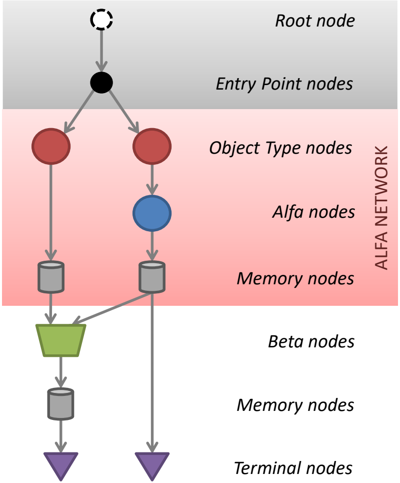
Terminal nodes

NB: nodes of the previous rules are shared!

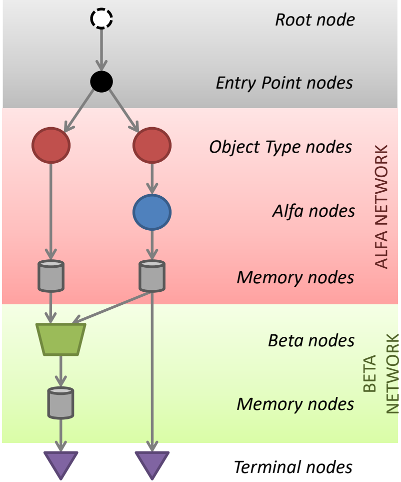
```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



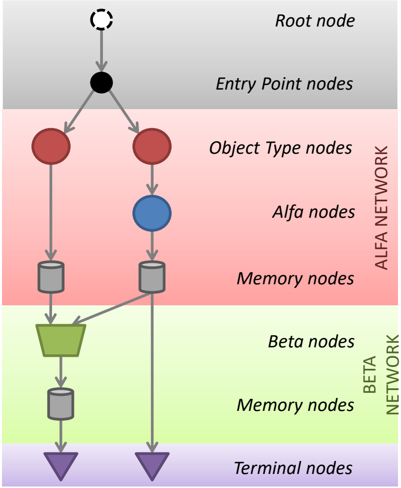
```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

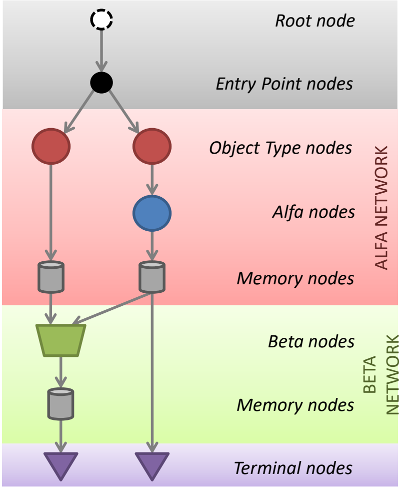


```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```



```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

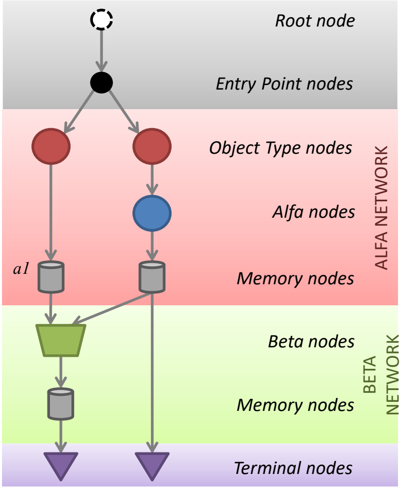
```
_
```



```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

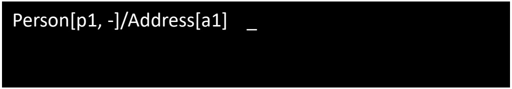
a1: Address("Via Po 2", 40068,
"San Lazzaro")

```
_
```

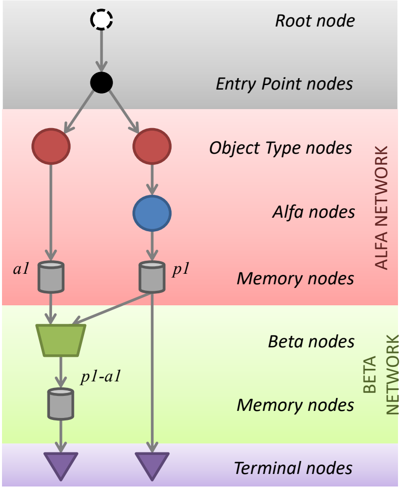


```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)



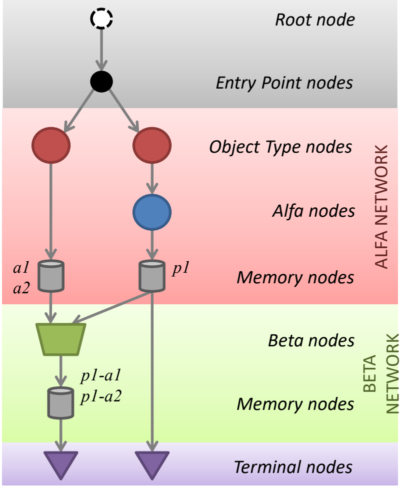
```
Person[p1, -]/Address[a1]  _
```



```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")

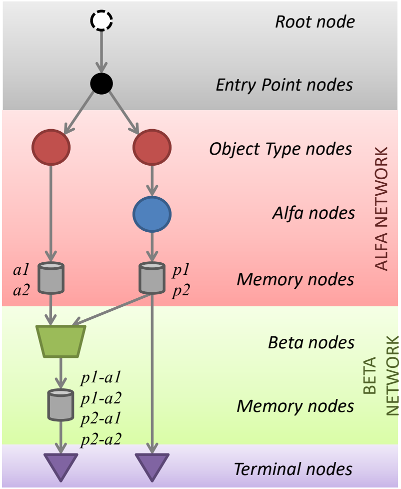
```
Person[p1, -]/Address[a1]   Person[p1, -]/Address[a2]
_
```



```
rule "Find Bobs and addresses"
when
   $a: Address()
   $p: Person( name=="Bob" )
then
   System.out.println($p+"/"+$a+" ");
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")
p2: Person("Bob", a1)

```
Person[p1, -]/Address[a1]   Person[p1, -]/Address[a2]
Person[p2, -]/Address[a1]   Person[p2, a1]/Address[a2]
_
```

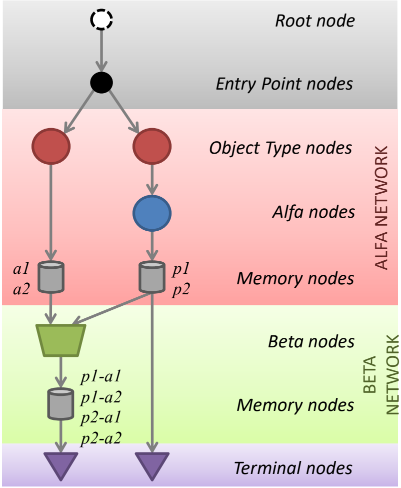


```
rule "Find Bobs and addresses"
when
   $a: Address()
   $p: Person( name=="Bob" )
then
   System.out.println($p+"/"+$a+" ");
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")
p2: Person("Bob", a1)
p3: Person("Giacomo", a1)

```
Person[p1, -]/Address[a1]   Person[p1, -]/Address[a2]
Person[p2, -]/Address[a1]   Person[p2, a1]/Address[a2]

_
```

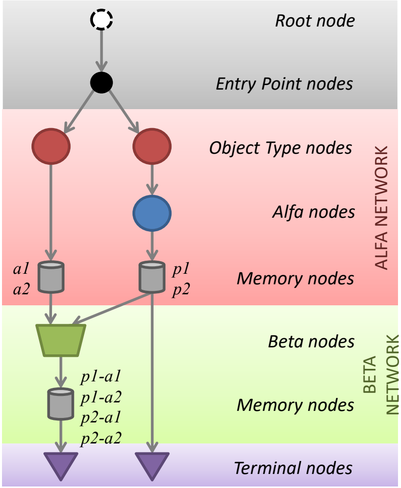


```
rule "Find Bobs and addresses"
when
  $a: Address()
  $p: Person( name=="Bob" )
then
  System.out.println($p+"/"+$a+" ");
end
```

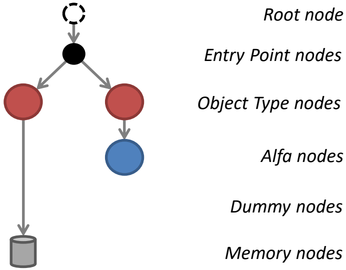
a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")
p2: Person("Bob", a1)
p3: Person("Giacomo", a1)

```
Person[p1, -]/Address[a1]   Person[p1, -]/Address[a2]
Person[p2, -]/Address[a1]   Person[p2, a1]/Address[a2]

_
```



```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

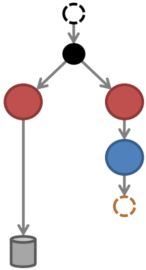
Object Type nodes

Alfa nodes

Dummy nodes

Memory nodes

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

Object Type nodes

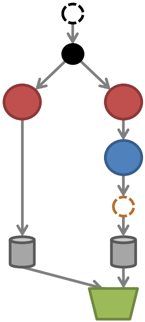
Alfa nodes

Dummy nodes

Memory nodes

NB: this Alfa node contains a cross reference that cannot be resolved.

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

Object Type nodes

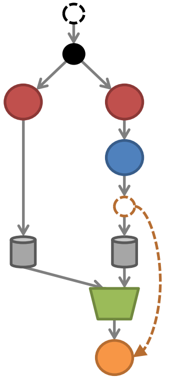
Alfa nodes

Dummy nodes

Memory nodes

Beta nodes

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

Dummy nodes

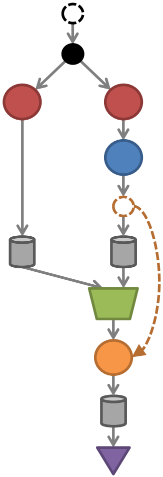
Memory nodes

Beta nodes

Delayed nodes

NB: the previous Alfa node is inserted here because it can resolve the cross reference.

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

Dummy nodes

Memory nodes

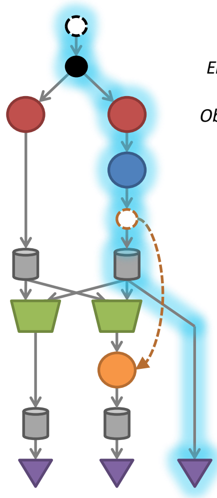
Beta nodes

Delayed nodes

Memory nodes

Terminal nodes

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

Dummy nodes

Memory nodes

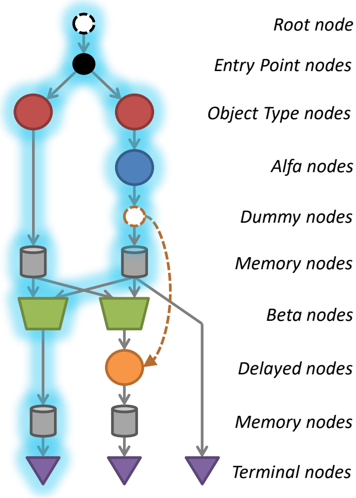
Beta nodes

Delayed nodes

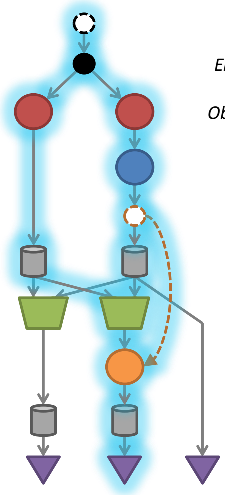
Memory nodes

Terminal nodes

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



Root node

Entry Point nodes

Object Type nodes

Alfa nodes

Dummy nodes

Memory nodes

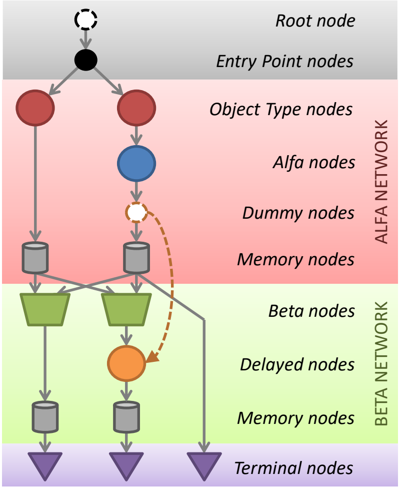
Beta nodes

Delayed nodes

Memory nodes

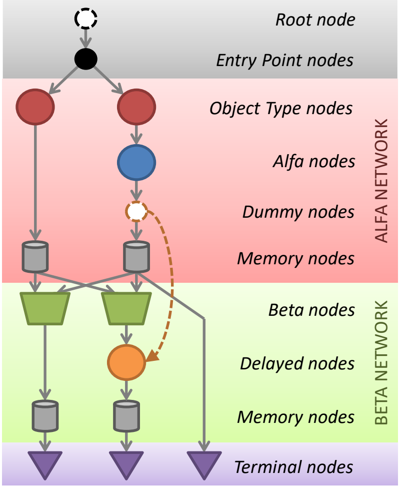
Terminal nodes

```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```



```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```

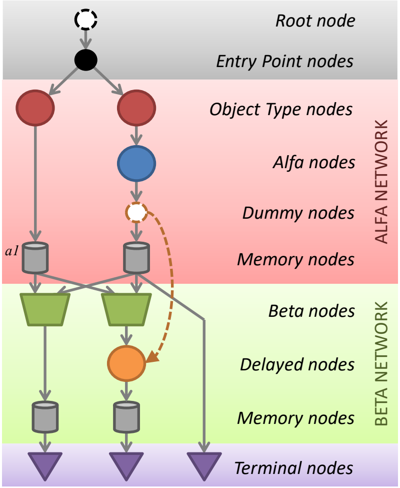
```
_
```



```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")

```
_
```

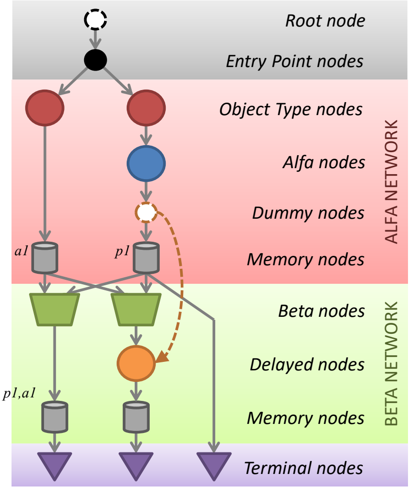


```
rule "Find Bob with its address"
when
   $a: Address()
   $p: Person( name=="Bob",
               address == $a )
then
   System.out.println($p);
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")

p1: Person("Bob", null)

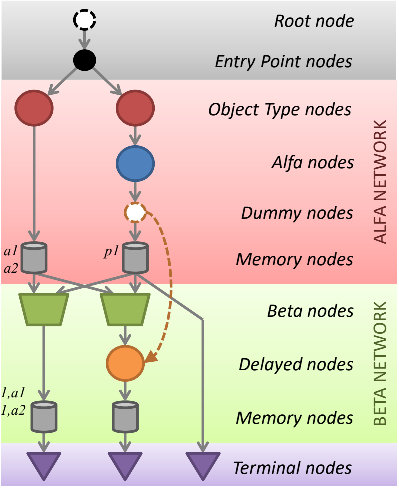
```
_
```



```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")

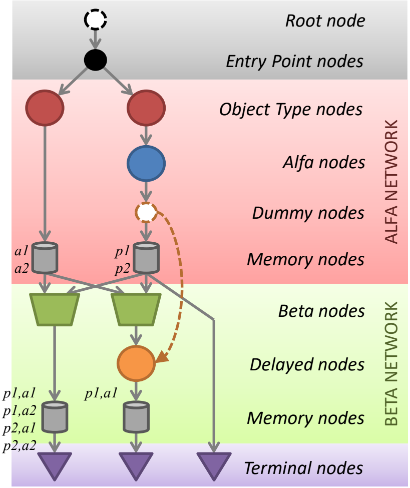
```
_
```



```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")
p2: Person("Bob", a1)

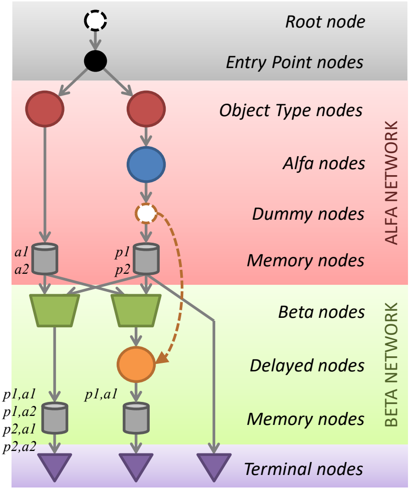
```
Person[Francesco, Address[Via Po 2, 40068, San Lazzaro]]
_
```



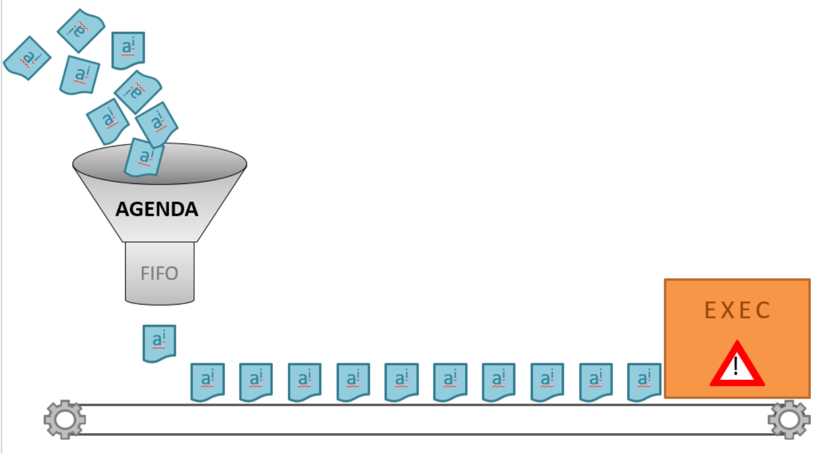
```
rule "Find Bob with its address"
when
  $a: Address()
  $p: Person( name=="Bob",
              address == $a )
then
  System.out.println($p);
end
```

a1: Address("Via Po 2", 40068,
"San Lazzaro")
p1: Person("Bob", null)
a2: Address("Via Roma 5",
40128, "Bologna")
p2: Person("Bob", a1)
p3: Person("Frank", a1)

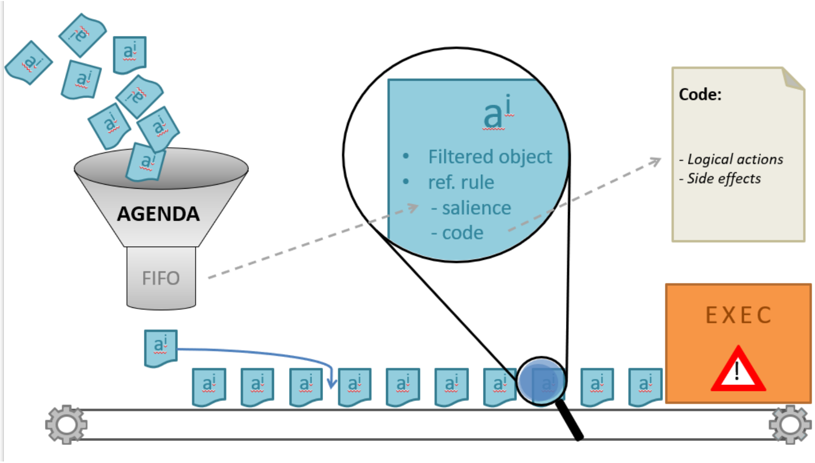
```
Person[Francesco, Address[Via Po 2, 40068, San Lazzaro]]
_
```



Conflict resolution and Execution

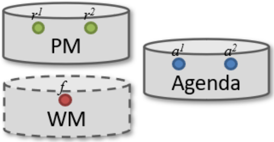


Conflict resolution and Execution



Conflict resolution and Execution

Conflict resolution and Execution

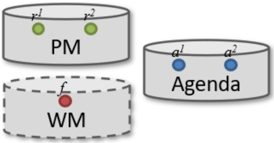


```
rule "r1"
when
  F()
then
  assert(new G());
end
```

```
rule "r2"
when
  $f: F()
then
  retract($f);
end
```

?!?

Conflict resolution and Execution

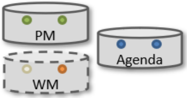


```
rule "r1"
when
  F()
then
  assert(new G());
end
```

```
rule "r2"
when
  $f: F()
then
  retract($f);
end
```

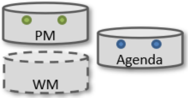
?!?

$a^1 < a^2$



First insert G,
then retract F.

$a^2 < a^1$



First retract F,
a1 cannot be applied,
G never inserted.

$r^1 < r^2$

```
rule "r1"
salience 10
...
```

```
rule "r2"
salience 5
...
```

Estabilish a precedence
fixed order between
r1 and r2.

REFERENCES

- Charles L. Forgy, “RETE: A Fast Algorithm for the Many Patter/Many Object Match Problem”, Artificial Intelligence, 19, pp. 17-37, 1982
- R.B. Doorenbos, “Production Matching for Large Learning Systems”, Ph.D. Thesis, 1995
- Schmit, Struhmer and Stojanovic, “Blending Complex Event Processing with the RETE algorithm”, in Proceedings of iCEP, 2008
- http://en.wikipedia.org/wiki/Rete_algorithm
- http://en.wikipedia.org/wiki/Complex_event_processing