BRMS: Best (and worst) Practices and Real World Examples

Edson Tirelli

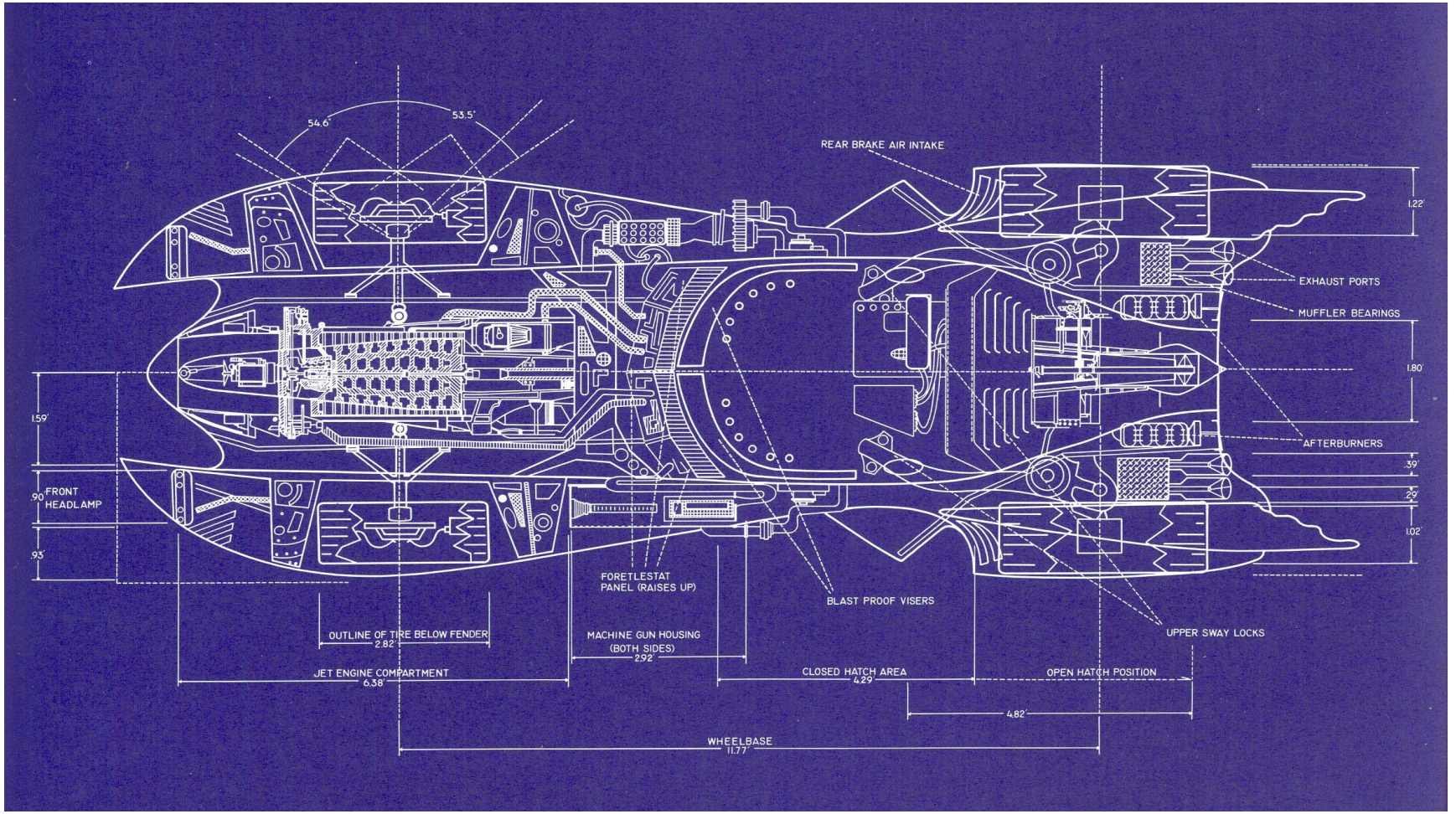
Senior Software Engineer

JBoss, by Red Hat

It all starts with a question...

What is the performance of <BRMS product name>?

Followed by...



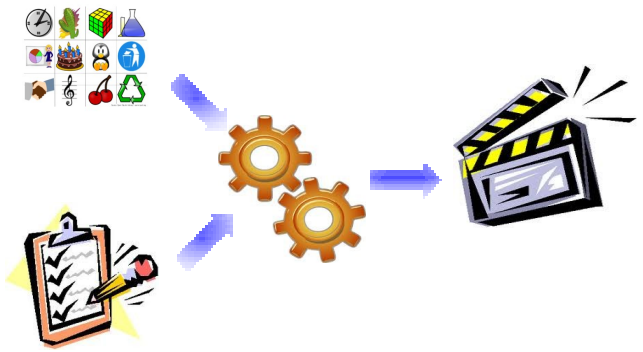
Are there any best practices (a.k.a blue prints) to achieve performance <X>?

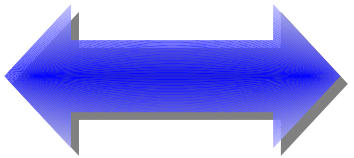
Lets talk about blue prints...

... but first, lets open our eyes to everything rules engines have to offer us!

BRMS adoption goals

A simple analogy



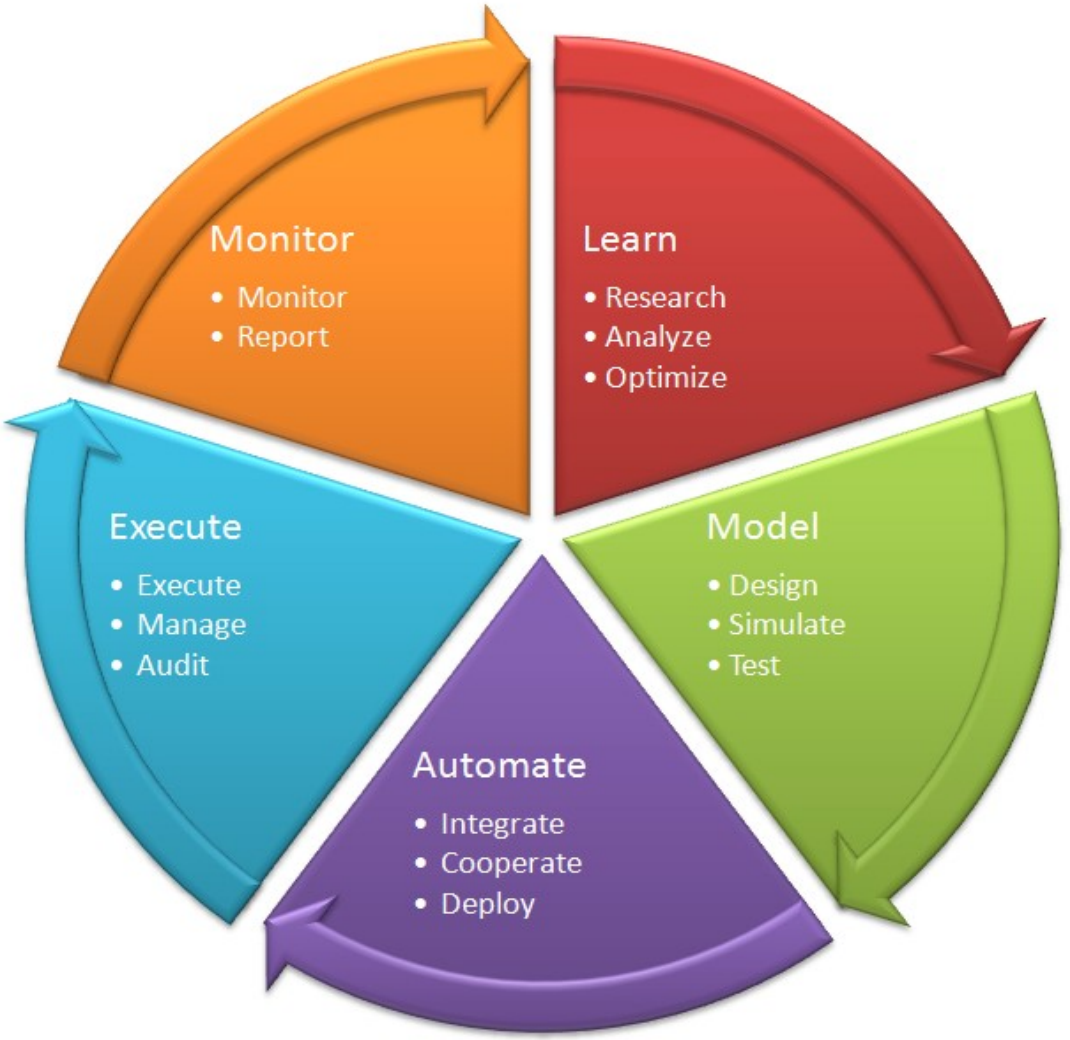


Business Rules Engines (Rules)

Databases (Data)

BRMS: goals

- **Independent Lifecycle Management of Business Rules**



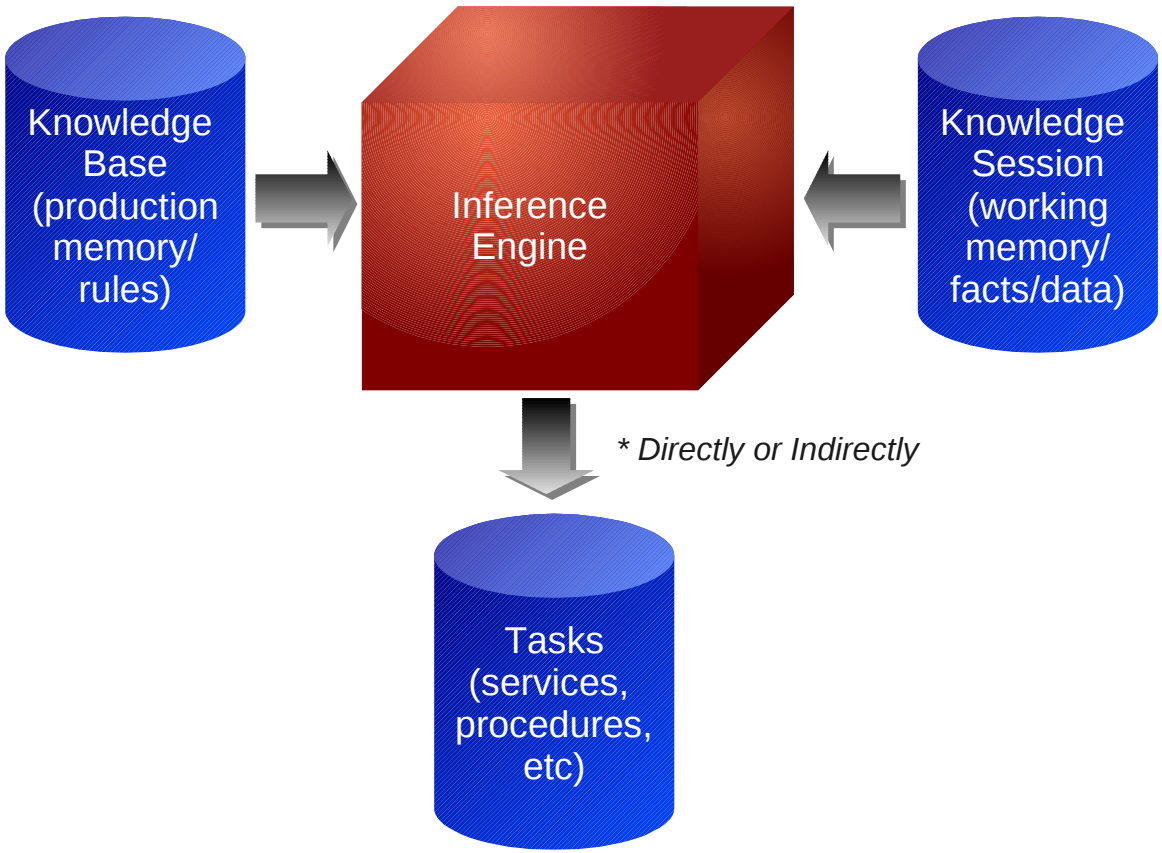
BRMS: goals

- **Declarative language for rules and queries**
  - Focus on “what to do”, not “how to do it”

```
rule “Send shipment pick-up SMS alert”
when
    There is a shipment order
    There is a route assigned to the order
    There is a truck GPS reading and the truck is 15m from the pick-up location
then
    Send SMS to customer: ARRIVING, “15 minutes”
end
```

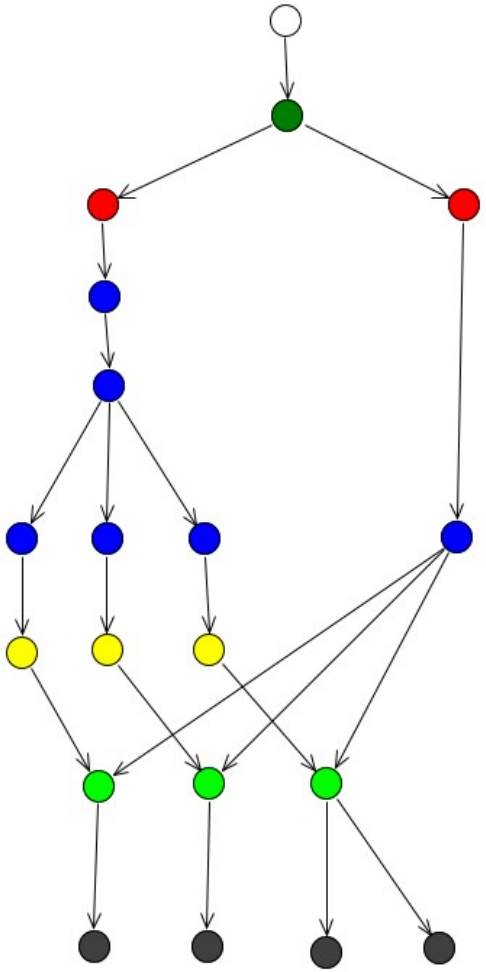
BRMS: goals

- **Logic, Data and Tasks split**
- **Centralization of Knowledge**
  - Consistency
  - Testing / Simulation
  - Auditing
- **Explanation Facility**



BRMS: goals

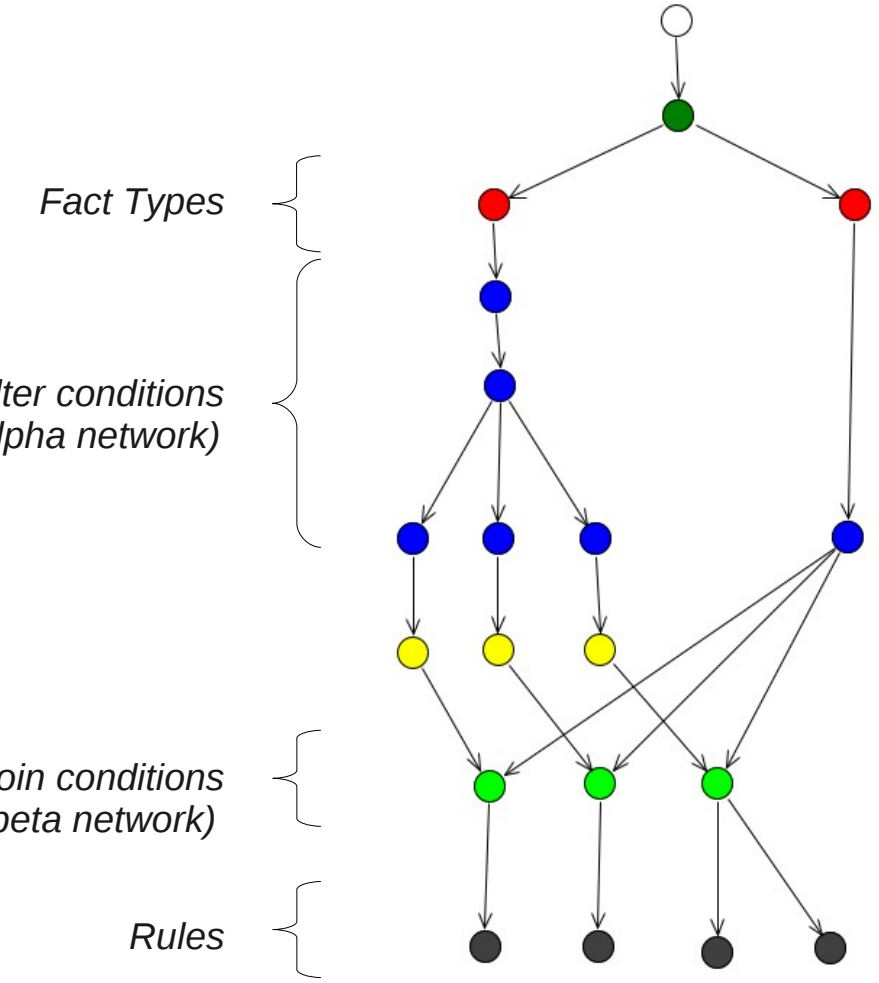
- **Execution Performance and Scalability**
  - Many-to-many pattern matching algorithm
  - Optimizes the whole set of rules
  - Performance is linear on the number of rules “touched” by each fact
  - Easily scales to tens of thousands of rules
  - Eager evaluation with incremental changes re-evaluation



Under the hood

Rete in 30 seconds

- **Classic Rete algorithm “idea”**
  - Discrimination network
  - Facts arrive at the top
  - Propagate down
  - If they reach the bottom, corresponding rule activates
- **+ Enables optimizations**
- **+ Enables additional feature set**



Typical Rete Optimizations

- **Support to POJOs as facts**
  - no mapping/data copy necessary
- **Full split between KBase and Working Memory**
  - lightweight session creation
  - knowledge base sharing
- **Completely Dynamic KBase management**
  - Hot addition/removal/updates of rules/queries/processes
- **Full support to First Order Logic and Set operations**
- **JIT compilation for constraints and data access**

Typical Rete Optimizations

- **Node Sharing (constraint sharing)**

- **Alpha Hashing:**
  - Provides O(1) performance for mutually exclusive constraints

- Rule 1: Customer is GOLD
- Rule 2: Customer is SILVER
- Rule 3: Customer is BRONZE
- Rule x: Customer is XYZ

Uses a hash function instead of checking each constraint

Typical Rete Optimizations

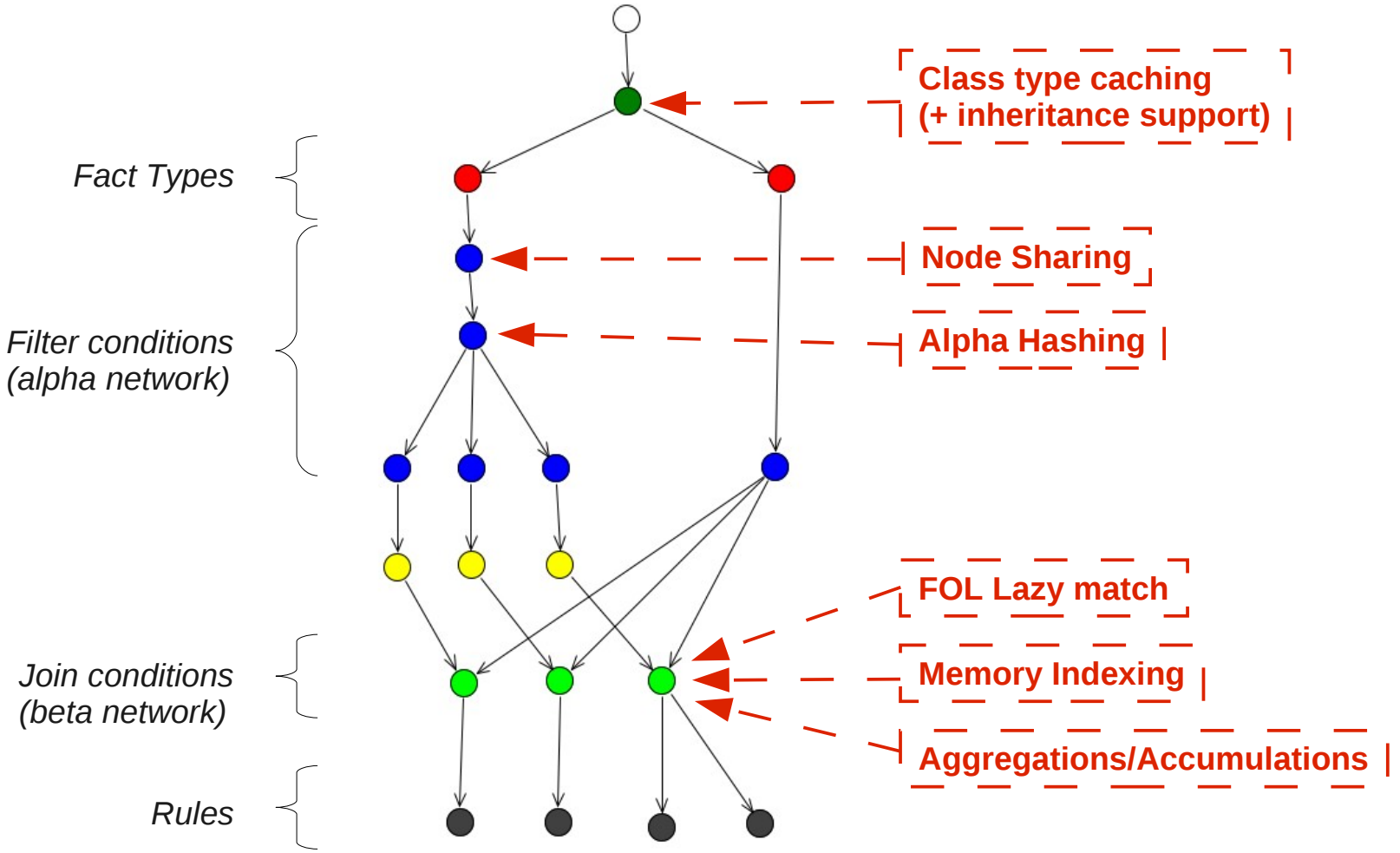
- **Beta Memory Indexing:**
  - Provides O(1) performance for joins

    Order(customerId == $someCustomer.id)

- **FOL lazy match:**
  - Avoids wasted matching on FOL conditional elements

    exists(Customer(age > 50))

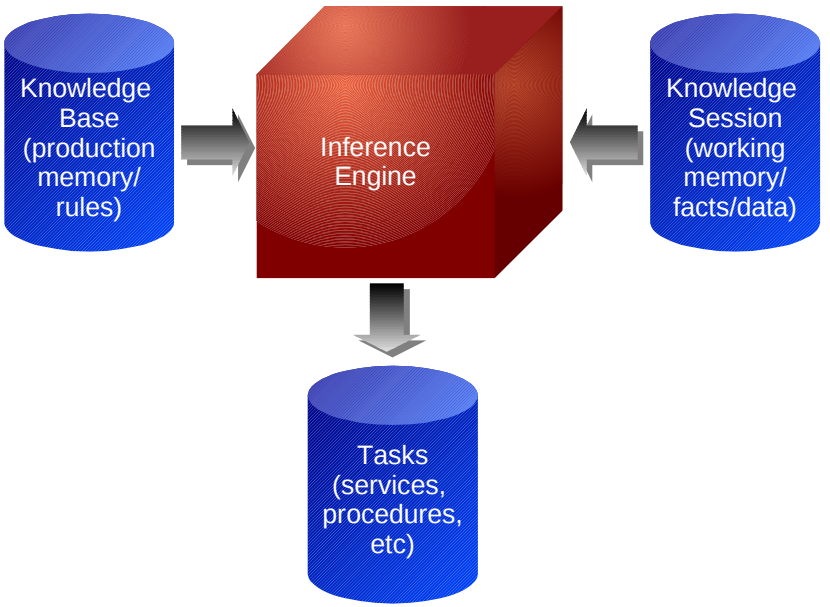
Typical Rete Optimizations: Visual Index



Best Practices

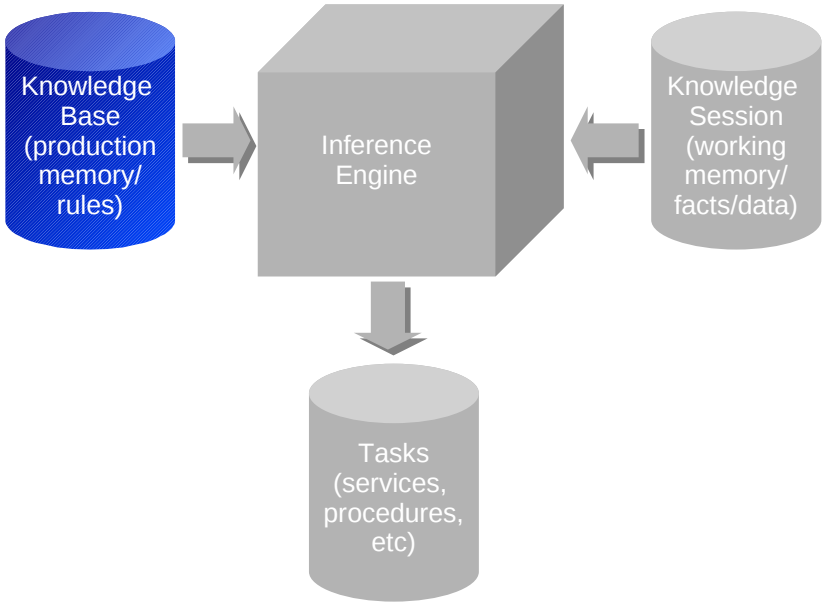
BRMS Best Practices

- **Don't try to micro-control rules execution**
  - **Use the Conflict Resolution Strategies instead**
    - **Salience**
    - **Agenda groups**
    - **Ruleflow**
    - **Dynamic enablement**



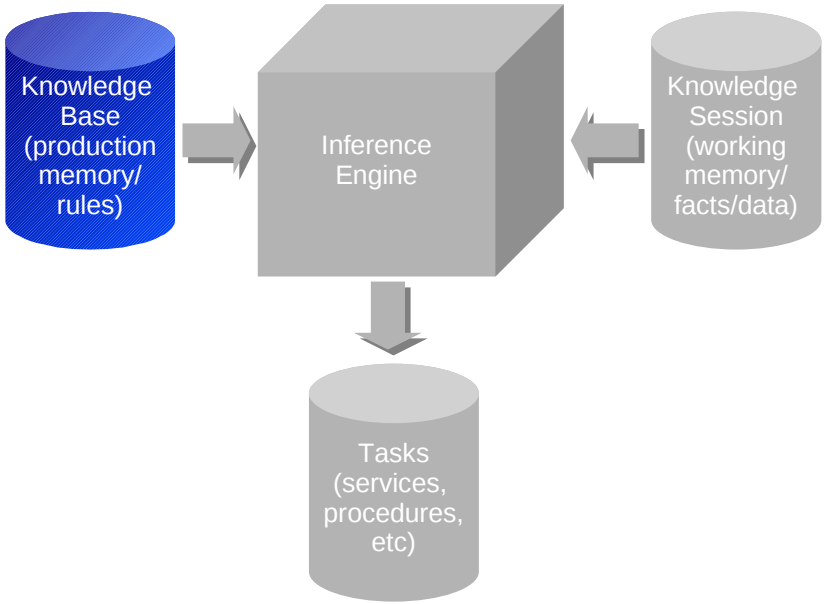
BRMS Best Practices

- **Partition your Knowledge Bases properly**
  - **Subject matter**
  - **Transaction / Service / Unit of Work**
  - **Business Entity**
- **Avoid monolithic kbases**
- **Avoid fine grained kbases**



BRMS Best Practices

- **Cache the Knowledge Base, share the sessions**
  - **Rules are usually JIT compiled at load time**
  - **Can be compiled at build time for non-dynamic use cases**



BRMS Best Practices

- **Don't overload rules**
  - **Each rule should describe one and only one scenario → action mapping**
  - **The engine will optimize shared conditions**
  - **The engine supports inference**

```
rule “Short sale”
when
    $ss : SecuritySaleOrder()
    not( Security( this == $ss.security ) )
then
    // identify as a short sale
    insert( new ShortSale( $ss ) );
end
```

```
rule “Refuse short sales”
when
    $ss : SecuritySaleOrder()
    exists( ShortSale( this.ss == $ss ) )
then
    // refuse order
    $ss.refuse();
end
```

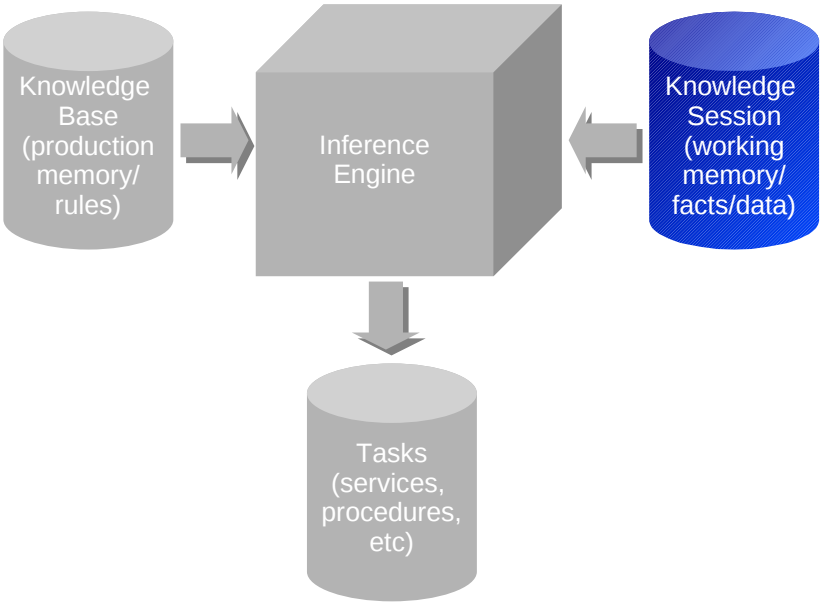
BRMS Best Practices

- Avoid the use of “evals”
  - Harder for the engine to optimize
  - Harder for rule authors to maintain
- Use engine features:
  - Pluggable operators
  - Annotations
  - ...

```
rule “Simultaneous calls from different places”
    @system( “Fraud Detection” ) @requirementId( “FD001” )
when
    $c1 : VoiceCall()
    $c2 : VoiceCall( cust == $c1.cust, base != $c1.base,
                     base notAdjancent $c1.base )
then
    // possible fraud detected
end
```

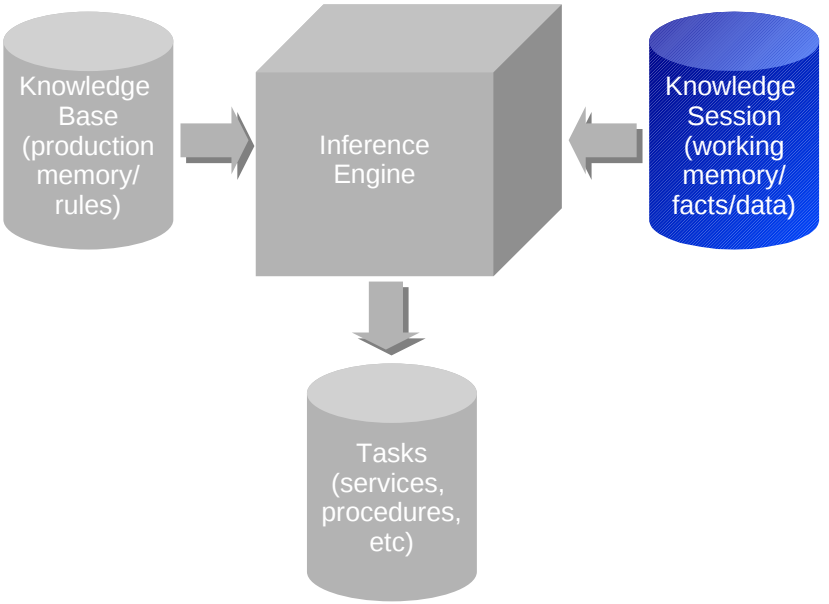
BRMS Best Practices

- **Batch data loads**
  - **It is usually faster to load 1000 facts and then fire the rules than fire rules after each inserted fact**
- **Partition the data into multiple sessions**
  - **Transaction / Service / Unit of work**
- **Creating a new session is cheap**
  - **Sometimes cheaper than removing facts from existing session.**



BRMS Best Practices

- **Quality of the data/fact model is directly proportional to the performance and maintainability of the rules using it**
  - **Think about the DBMS analogy**
  - **Flatter models improve performance**
  - **Smaller classes help avoiding recursions**



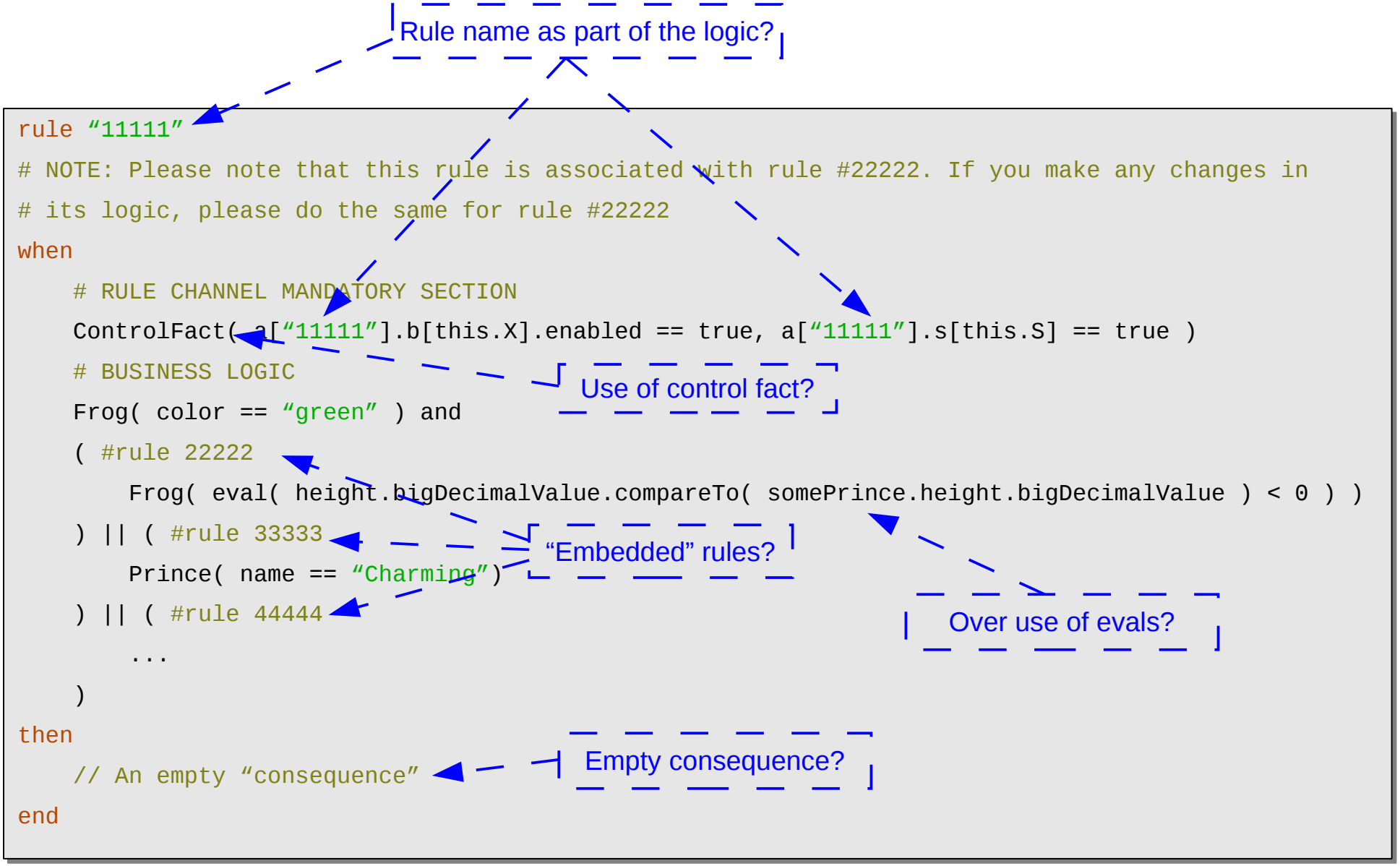
Best Practices in Rules Authoring

A Frog Rule: anything odd about it?

```
rule “11111”
# NOTE: Please note that this rule is associated with rule #22222. If you make any changes in
# its logic, please do the same for rule #22222
when
    # RULE CHANNEL MANDATORY SECTION
    ControlFact( a[“11111”].b[this.X].enabled == true, a[“11111”].s[this.S] == true )
    # BUSINESS LOGIC
    Frog( color == “green” ) and
    ( #rule 22222
        Frog( eval( height.bigDecimalValue.compareTo( somePrince.height.bigDecimalValue ) > 0 ) )
    ) || ( #rule 33333
        Prince( name == “Charming”)
    ) || ( #rule 44444
        ...
    )
then
    // An empty “consequence”
end
```

PS: lets focus on the structure of the rule and not the actual logic.

A Frog Rule: looks suspicious



Why is an empty consequence bad?

```
rule “11111”
when
    ...
then
    // An empty “consequence”
end
```

- ✗ Breaks independent lifecycle management goal
- ✗ Breaks centralization of knowledge goal
- ✗ Breaks clarity goal (what to do?)
- ✗ Breaks explanation facility goal
- ✗ The Frog will never turn into a Prince
- ✓ Should this be a query instead?

Alternatives: Rules vs Queries

	Rules	Queries
Control	Invoked by the engine	Invoked by the application
Parameters	Don't support parameters	Support parameters
Results	Execute actions	Return results

```
rule “Transform Frog into Prince”
when
    $f : Frog( color == “green” )
then
    insert( new Prince( “Mark” ) );
    retract( $f );
end
```

```
query “Get the frog”( $color )
when
    $f : Frog( color == $color )
end
```

Why is bad to use the rule name in the logic?

```
rule “11111”
when
    ControlFact( a[“11111”].b[this.X].enabled == true, a[“11111”].s[this.S] == true )
    ...
    ( #rule 22222
    ...
```

- Breaks clarity goal
- Breaks “one scenario – one action” best practice
- Rule names must be unique within the knowledge base
- Usually leads to micro-control and procedural code

Alternatives for metadata declaration

```
rule “11111 – transform the frog into a prince”
    @requirementId( “11111” ) @author( “Mark” )
    @relatedTo( “22222, 33333” )
    @documentation( “A rule with lots of annotations” )
when
    ...
then
    insert( new Prince( “Mark” ) );
    messenger.announceTheWedding( drools.getRule().getMetaData().get( “requirementId” ) );
end
```

- ✔ **Use annotations**
- ✘ **Do not use metadata in conditions**
- ✔ **It is ok to use metadata in consequences**
- ✔ **If necessary, metadata is available to the enabled attribute**

Why Control Facts should be avoided?

```
rule “11111”
when
    ControlFact( a[“11111”].b[this.X].enabled == true, a[“11111”].s[this.S] == true )
    ...
    ( #rule 22222
    ...
```

- ✗ **Breaks clarity goal**
- ✗ **Breaks one scenario – one action best practice**
- ✗ **Leads to micro-control and procedural code**
- ✗ **Usually leads to runtime overhead**
- ✓ **Sometimes, control facts are useful, but not for the general case**

Alternatives for control facts

```
rule “11111 – transform the frog into a prince”
    agenda-group “x”
    ruleflow-group “y”
    salience 10
    enabled ( ... a boolean expression ... )
when
    ...
```

Usually one of these attributes is enough

- ✔ **Use rule attributes for conflict resolution**
- ✔ **Use enabled expressions to conditionally disable rules**
- ✘ **Do not abuse of salience**
- ✔ **Let the engine do its job**

Why “embedded” rules are bad?

```
( #rule 22222
    Frog( eval( height.bigDecimalValue.compareTo( somePrince.height.bigDecimalValue ) < 0 ) )
) || ( #rule 33333
    Prince( name == “Charming”)
) || ( #rule 44444
    ...
)
```

- Breaks independent lifecycle management
- Breaks clarity goal
- Breaks explanation facility goal
- Breaks performance goal
- Breaks one scenario – one action best practice
- Increases maintenance cost and testing complexity

Alternatives for “embedded rules”

```
rule “11111.a – Teenagers are eligible”
when
    $p : Person( age >= 16 && <= 18 )
then
    insert( new Eligible( $p ) );
end
```

```
rule “11111.b – Elders are eligible”
when
    $p : Person( age >= 60 )
then
    insert( new Eligible( $p ) );
end
```

```
rule “11111.c – Eligibles get discount”
    no-loop
when
    $t : Ticket()
    $e : Eligible()
then
    modify($t) { setDiscount( 0.25 ) }
end
```

Alternatives for “embedded rules”

- One scenario – one action
- Allow the engine to do its job
- Take advantage of flow control features

Q&A

Edson Tirelli
etirelli@redhat.com