

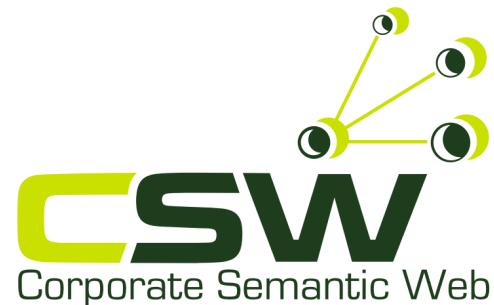
# Prova



## Network-based Information Systems WS'08/09

Prova: A Semantic Web Rule Engine

Prof. Dr. Adrian Paschke



# Agenda

---

- Motivation: Declarative Logic Programming
- Prova – Syntax and Semantics
- ContractLog Library
- Prova Agent Architecture
- Programming with Prova

# Advantages of Logic Programming

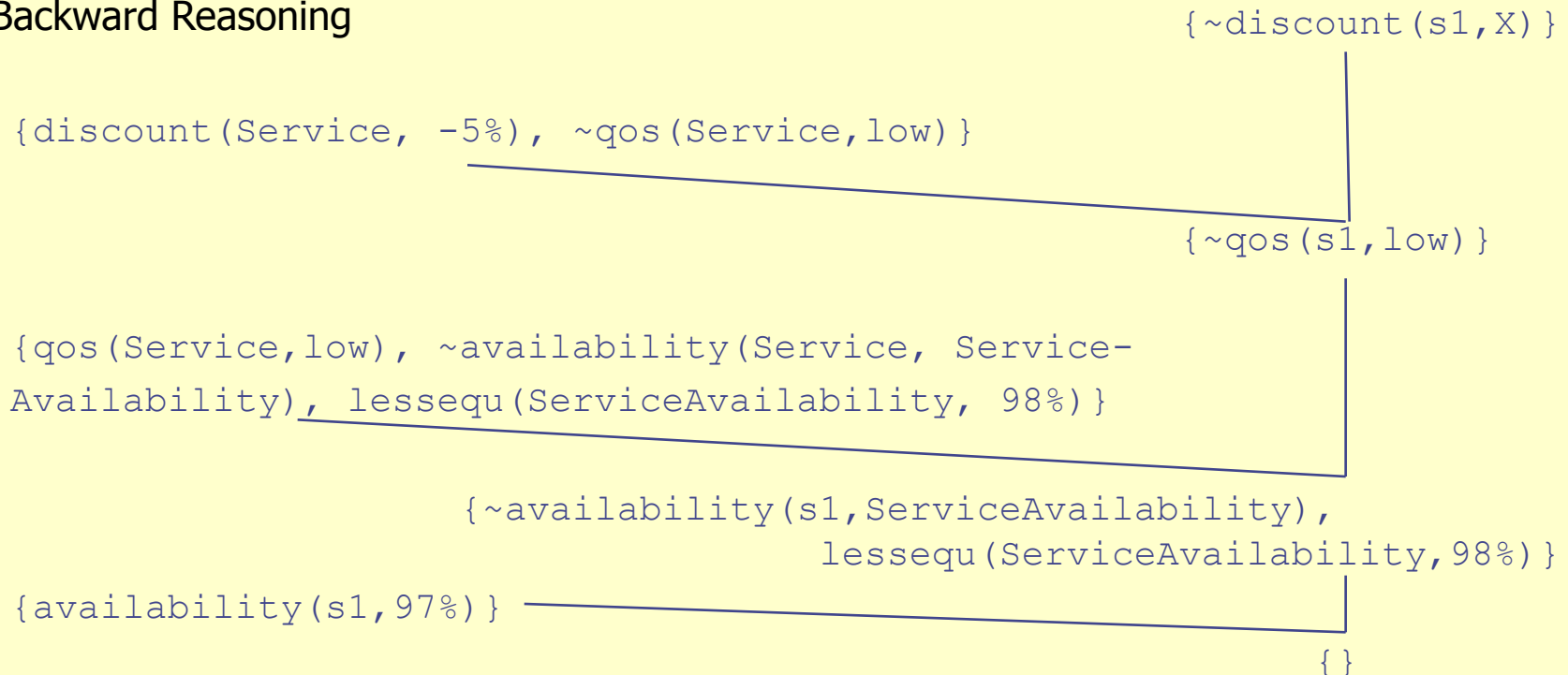
---

1. Compact declarative representation of rules by
  - Global validity within a scope (module)
  - Separation of contract rules and application code
  - Simple extension of the rule base (without changing the interpreter)
2. Efficient, generic interpreter (LP inference engines) for automated rule changing and rule derivation
3. Automated conflict resolution
  - Traceable and verifiable rule sets
  - Integrity constraints are possible
  - Automated conflict resolution (e.g., by rule prioritization)

# Simplified Example: Resolution

- **Rules:** `discount(Service, -5%) :- qos(Service, low).`  
`qos(Service, low) :-`  
`availability(Service, ServiceAvailability),`  
`lessequ(ServiceAvailability, 98%).`
- **Fact:** `availability(s1, 97%).`
- **Query:** `:-solve(discount(s1, X))` → **result:** `X = -5%`

## Backward Reasoning



# Declarative Knowledge Representation

## Logic Programming

```
discount(Service, 5%) :- qos(Service,high).
discount(Service, -5%) :- qos(Service,low).
qos(Service,high):- availability(Service) = 1.
qos(Service,low):- availability(Service) < 0,98.
```

### Queries

```
discount(Service,X)? All discounts for all service
discount(s1,X)?      Discount for „s1“
discount(s1,5%)?     Service „s1“ → discount 5%?
discount(Service,5%)? All serv. with discount 5%
qos(Service,Y)? All service levels for all services?
qos(s1,Y)?          Service level for “s1”?
```

...

## Procedural Programming

```
boolean getsDiscount(Service s, int value) {
    if (getAvailability(s)==1) && (value==1) return true;
    else if (getAvailability(s)<0,98) && (value<0,98) return
true;
    else return false;
}
...
Service getService(int value) {
    for (int i=0;i<getAllServices();i++) {
        Service s = getService(i);
        if (getAvailability(s)==1) && (value==1) return s;
        else if (getAvailability(s)<0,98) && (value < 0,98)
return s;
        else return null;
    }
}
...
int getDiscount(Service s) {
    if (getAvailability(s)==1) return 5;
    else if (getAvailability(s)<0,98) return -5;
    else return 0;
}
...
```

# High Flexibility

- Add new rules without extending the inference machine / interpreter (!)

*If availability of service is higher than 99.98% then quality of service is high*

Prerequisite		
Predicate	Complex Term	Constant
>	availability(Service)	99.98%



Conclusion		
Predicate	Variable	Constant
qos	Service	high



*If quality of service is high then discount for service is +5 %*

Prerequisite		
Predicate	Variable	Constant
qos	Service	high



Conclusion		
Predicate	Variable	Constant
discount	Service	+5%

# Consistency of Rule Base

---

- Automated conflict detection

## Logic Programming:

```
discount (Customer, X) ? X=5%; X=10%
```



### → Integrity Constraints

```
integrity (xor (
  discount (Customer, 5%), discount (Customer, 10%)))
```

### → Automated Conflict Resolutions by e.g. priorities

```
overrides (discount10%, discount5%)
```

# Prova

a

Distributed Semantic Web Rule Engine

<http://prova.ws>

# What is Prova?

---

- Open-Source project hosted at Sourceforge
- Loosely based on Mandarax
  - extends and rewrites the Java-based Mandarax inference system
- Historically, initially focusing on bioinformatics applications
- Now: Distributed Semantic Web Rule Engine
  - redeveloped (e.g. completely new inference engine) and heavily extended, e.g. in the Rule-based Service Level Agreements (RBSLA project) at the Technical University Munich
- The tool designed specifically for distributed information, knowledge, and computation integration
  - Separation of logic, data access, and computation
- Built-in integration of various data sources and technologies: relational databases, XML, RDF, flat files, message oriented middleware.
- Combines the benefits of declarative (rule-based), procedural (e.g., Java), and workflow languages (e.g., BPEL)

# Rules for Java

---

- ◆ Combine the benefits of **declarative and object-oriented programming**;
- ◆ Interpreted scripting syntax combines those of ISO **Prolog and Java**;
- ◆ Expose logic as rules;
- ◆ Access data sources via wrappers written in Java, query language built-ins (e.g. SQL, SPARQL, RDF Triples, XQuery) or message-driven (Web) service interfaces;
- ◆ Make *all* Java **API** from available packages directly accessible from **rules**;
- ◆ **Run within the Java runtime**;
- ◆ Be compatible with modern enterprise service and agent-based software architectures

# Prova / ContractLog Inference Engine

---

- Linear tabling (goal memoization)
  - resolve infinite loops + redundant computations, enabling tabled predicates for sequential operators such as cuts, built-in and procedural attachments
- Efficient memory structures in trampoline style
  - no stackoverflows for very large derivation trees (large knowledge bases)
- Extended key indexing (faster access to KB and narrower search space)
- Procedural Semantics: *SLE Resolution*
- Temporarily undefined truth value e.g. for long running, distributed derivations in (open) distributed KBs (e.g. in the Semantic Web)
- Declarative Semantics:  $WFSX_{DefL}$ 
  - Terminating and sound and complete wrt to the extended well-founded defeasible logic semantics  $WFSX_{DefL}$
- Complexity is comparable with SLG resolution
- Downward compatible with SLDNF resolution

# Well Founded Defeasible Semantics for Extended Logic Programs (WFSX<sub>DefL</sub>)

---

- Well-founded Semantics (WFS)

A. Van Gelder, K. Ross, and J. Schlipf. The well-founded semantics for general logic programs. JACM, 38(3):620650, 1991.

- 3-valued logic (true, false, unknown)

- Well-founded semantics for extended logic programs (WFSX)

L. M. Pereira and J. J. Alferes. Well founded semantics for logic programs with explicit negation. Proceedings of ECAI'92, 1992.

- Extended Herbrand Base:

explicit negation  $\neg \perp$  ; default negation  $\sim \perp$

- Coherence: *Explicit negation implies Default negation*:  $\neg \perp \rightarrow \sim \perp$

- Extended Well-founded Defeasible Semantics (WFSX<sub>DefL</sub>)

A. Paschke: Rule Based Service Level Agreements, PhD thesis, Technical University Munich.

- Four-valued logic (true, false, undefined, temp. undef.)

Conflicts defined by integrity constraints

- $\Delta \neg L \rightarrow - \Delta L$  (strict coherence)

- $\partial \neg L \rightarrow - \partial L$  (defeasible coherence)

# SLE Resolution

---

- Linear resolution with Selection function for Extended WFS
  - Note: SLG = non-linear approach (e.g. XSB Prolog)
- Extends linear SLDNF
  - Top-down proof procedure (deep-first, linear)
  - Goal memoization based on linear tabling
  - Loop cutting
- Four truth values (true, false, unknown, temp. Unknown)

# Syntax design features of Prova

---

- ◆ The syntax combines ISO Prolog and Java but the key is simplicity:

```
% Prolog                                     % Prova
N2 is N + 1,                                  N2 = N + 1,
-----
// Java                                        % Prova
List l = new java.util.ArrayList();          L=java.util.ArrayList(),
```

- Low-cost creation of distributed integration and computation workflows.
- Prova separates logic, data, and computation.
- Low-cost integration of rule-base scripted agents inside Java and Web applications.
- For more information check the User`s Guide in the prova web page.

# Using colours for language elements

---

The colours below are used to distinguish the language elements:

```
% Comments are in green

% Built-in predicates are brown
tokenize_list(Line, "\t", [T|Ts])

% User-defined predicates are blue
member(X, [X|Xs])

% Java calls and constructors are red
Text1=P1.toString()

% Table names are pink
sql_select(DB, cla, px(PXA), pdb_id(PDB))

% Message performatives (speech acts) are navy blue
rcvMsg(Protocol, From, query_ref, [X|Xs]) :-
```

# Prova Syntax

- Variables (upper case), Constants (lower case)
- Fact: `availability(s1, 99%)` .
- Rule: `qos(S, high) :- availability(S, 99%)` .
- Query 1: `:- solve(not(qos(S, high)))` .
- Query 2: `:- eval(not(qos(S, high)))` .
- Derive: `, derive([X|Args], ...`
- Scoping: `scope(p(X), "www.prova.ws"), ...`
- Memoization: `cache(p(X)), ...`
- Lists: `[Head|Tail] = [Head, Arg2, ..., ArgN] = Head(Arg2, ..., ArgN)`
- Module Imports:
  - `:- eval(consult("ContractLog/list.prova"))` .
  - `:- eval(consult("http://rule.org/list.prova"))` .
- Meta data annotation:  
`metadata(label(r1), src("www.prova.ws"), dc_author("AP")) :: qos(S, medium) :- availability(S, 98%)` .

# Example: General language constructs

---

Prova extends ISO Prolog syntax:

```
% Facts
i_am("mediator").
portfolio("balanced",P).
reachable(X,X).
is_a("anticoagulant","molecular_function").

% Clauses (head true if conditions true in the body)
parent(X,Y) :-
    is_a(Y,X).
parent(X,Y) :-
    has_a(X,Y).

% Goals (note there is no head)
:- solve(parent(Parent,"anticoagulant")). % Print solutions
:- eval(parent(Parent,"anticoagulant")). % Just run exhaustive search
```

Format of the output for the goal *solve* above:

```
Parent="molecular_function"
```

# Pervasive use of the Java type system

---

- ◆ Java typed and untyped variables;
- ◆ Natural rules for subclasses unification;
- ◆ Java variables prefixed by full package
  - ◆ prefix with *java.lang* being default;

```
:- solve(member(X, [1, Double.D, "3"])).
:- solve(member(Integer.X, [1, Double.D, "3"])).

% Standard type-less rules for the standard member predicate
member(X, [X|Xs]).    % X is a member of a list if it is the first element
member(X, [_|Xs]) :- % X is a member of a list if it is in the list tail
    member(X, Xs) .

-----

> X=1
> X=java.lang.Double.D
> X=3

> java.lang.Integer.X=1
```

# Description Logic (DL) type system

---

- ◆ DL-typed and untyped variables;
- ◆ Uses Semantic Web ontologies as type systems
- ◆ Uses external DL reasoner (e.g. Pellet) for dynamic type checking
- ◆ Syntax:  $[Variable] : [nameSpace] \_ [ClassType]$   
 $[individualConstant] : [nameSpace] \_ [ClassType]$

```
:- eval(consult('ContractLog/owl.prova')). % needed
% import external type system (T-Box model) and individuals (A-Box)
import("http://example.org/WineProjectOWL.owl").
% use OWL-DL reasoner; for a list of available predefined reasoners see OWL2PROVA.java
reasoner ("dl").
% typed rule
serve (X:default_Wine) :- recommended(X:default_Wine).
% ground fact; defines an instance of class
recommended(default_White_Wine:Chardonnay).
% non ground DL facts are interpreted as queries on external ontology
recommended(X:default_White_Wine).
recommended(X:default_Red_Wine).
:-solve(recommended(X:default_Wine)).
:-solve (recommended(X:default_White_Wine)).
```

# Java Method Calls

---

- ◆ Constructors, instance and static methods, and public field access;
- ◆ Ability to embed Java calls makes the Prolog-like programming style more suitable for integration and computation workflows.

```
hello (Name) :-
```

```
    S = java.lang.String ("Hello") .
```

```
    S.append (Name) ,
```

```
    java.lang.System.out.println (S) .
```

# Example: Java-based XML Processing (DOM)

---

- ◆ A small wrapper for XML DOM in Java is needed because of bugs in reflection and to improve access to elements.
- ◆ Non-deterministic iteration over elements and attributes node collections with *nodes* method.

```
:- eval(test_xml()).
test_xml() :-
    % This extends the standard DOM API: create a Document based on File name
    Document = XML("blast.xml"),
    Root = Document.getDocumentElement(),
    Elements = Root.getElementsByTagName("Hit"),
    % This extends the standard DOM API: enumerate the nodes
    Elements.nodes(Element),
    SubElements = Element.getElementsByTagName("Hsp"),
    SubElements.nodes(SubElement),
    ChildNodes = SubElement.getChildNodes(),
    ChildNodes.nodes(ChildNode),
    ChildNodeName = ChildNode.getNodeName(),
    DataName = ChildNode.getFirstChild(),
    StringName = DataName.getNodeValue(),
    println(["  Child name: ",ChildNodeName]),
    println(["  Child value: ",StringName]).
```

# Exception Handling

---

- ◆ Exception handling that results in a failure and backtracking
- ◆ Compensation handling in which the control flow continues

```
:- eval(raise_test()).

handle_exceptions (Msg) :-
    exception(Ex),
    printl([Ex]),
    println([Msg]),
    fail().%fail

raise_test () :-
    on_exception(java.lang.Exception,
                 handle_exceptions(" in raise_test()")),
    Ex = java.lang.Exception("A Prova exception"),
    raise(Ex). % throw exception
```

# Built-Ins

---

- Simple arithmetic relations (+ - = ...)

```
% Prova  
N2 = N + 1,
```

- Negations (not, neg)

```
% Default Negation  
register(User) :- not(known(User)), ...
```

- Fact base updates

```
% Update global facts  
register(User) :- not(known(User)), assert(known(User)).
```

- Variable mode tests (free, bound, type)
- String manipulation predicates

```
% Concat Strings  
concat(["{",In,"}"],Out), % prepend "{" and append "}"
```

# External Data and Object Integration

- File Input / Output

```
..., fopen(File, Reader), ...
```

- XML (DOM)

```
document (DomTree, DocumentReader) :-  
    XML (DocumenReader),
```

- SQL

```
..., sql_select(DB, cla, [pdb_id, "1alx"], [px, Domain]).
```

- RDF

```
..., rdf(http://..., "rdfs", Subject, "rdf_type", "gene1_Gene"),
```

- XQuery

```
..., XQuery = ' for $name in  
StatisticsURL//Author[0]/@name/text() return $name',  
xquery_select(XQuery, name(ExpertName)),
```

- SPARQL

```
..., sparql_select(SparqlQuery, name(Name), class(Class),  
definition(Def)),
```

# Prova's SQL Support

---

## Open database

```
:-eval(consult("utils.prova")). % import SQL functions
...
location(database,"jdbc
string","database_name","username","password").
...
dbopen("database_name",DB)
```

## Query database

```
sql_select(DB,From,[N1,V1],...,[Nk,Vk],
           [where,Where],[having,Having],[options,Options])
```

## Manipulate database

```
sql_insert(DB,Table,[N1,...,Nk],[V1,...,Vk])
```

# Prova SQL Examples

```
sql_select (DB, cla, [px, PXA], [pdb_id, PDB_ID]),  
sql_select (DB, cla, [px, PXB], [pdb_id, PDB_ID]),  
PXA < PXB
```

%===== The same query as one statement  
that is executed much faster =====

```
sql_select (DB, 'cla as c1, cla as  
c2', ['c1.px', PXA], ['c2.px', PXB], ['c1.pdb_id', PD  
B_ID], [where, 'c1.pdb_id=c2.pdb_id and  
c1.px < c2.px'])
```

```
db_import (DB, scop, des, Line) :-  
    tokenize_list (Line, "\t", [T|Ts]),  
    sql_insert (DB, des, [id, type, sccs, sid, descript
```

# Examples: File I/O and SPARQL

```
test_fopen() :-  
    fopen(File,Reader) ,  
    % Non-deterministically enumerate lines in the file  
    read_enum(Reader,Line) ,  
    println([Line]) .          % Print one line at a time
```

```
exampleSPARQLQuery(URL,Type|X) :-  
    QueryString =  
    ' PREFIX foaf: <http://xmlns.com/foaf/0.1/>  
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>  
    SELECT ?contributor ?url ?type  
    FROM <http://planetrdf.com/bloggers.rdf>  
    WHERE {  
        ?contributor foaf:name "Bob DuCharme" .  
        ?contributor foaf:weblog ?url .  
        ?contributor rdf:type ?type . } ' ,  
    sparql_select(QueryString,url(URL),type(Type)|X) ,  
    println([[url,URL],[type,Type]|X],",") .
```

# ContractLog Library

# Overview ContractLog

Logic	Usage
<b>Extended Logic</b> math.prova datetime.prova list.prova list_math.prova	<b>Deductive reasoning with negation-as-finite-failure (not) and explicit negation (neg).</b> Mathematical computations and comparisons Date, time and interval functions List manipulating and querying functions Mathematical list operations and filters
<b>Dynamic Transactional Update Logic</b>  update.prova	<b>Dynamic ID based updates of the knowledge base including bulk updates with user-defined or external modules (sets of rules and facts) and transactional updates with integrity tests and roll-backs of update transitions via their IDs.</b> Expressive ID-based transactional update functions
<b>Description Logic Typed Logic</b> owl.prova	<b>Homogeneous and heterogeneous integration of Description Logics ontologies such as RDFS or OWL ontologies into Prova scripts.</b> Wraps the OWL2Prova API which supports integration of Semantic Web ontologies written in RDFS or OWL
<b>(Re)active Logic</b> ecaruntime.prova	<b>Active event detection/event processing and event-triggered (re-)actions.</b> Integrates and configures the ECA-LP runtime environment in Prova scripts
<b>Temporal Event/Action Logic</b>  ec.prova	<b>Temporal reasoning about dynamic systems and state changes such as interval-based complex event /action definitions (event/action algebra) and effects of events and actions on changeable knowledge state properties.</b> Implementation of the Event Calculus and various event/action logics extensions
<b>Deontic Logic</b> deontic.prova	<b>Normative reasoning on state based rights and obligations.</b> Implementation of a temporal role-based deontic logic with violations and exceptions
<b>Test Logic</b> integrity.prova testcase.prova	<b>Validation, verification and integrity testing of rule bases</b> Integrity tests of dynamic knowledge states which might be actual or possible hypothetical (future) states Functionalities for dynamically loading, testing and unloading test cases and tests
<b>Defeasible Logic</b> defeasible.prova	<b>Default rules and priority relations of rules and modules (rule sets). Facilitates conflict detection and resolution as well as revision/updating and modularity of rules. → default rules and rule priorities</b>

# ContractLog Examples

```
% import the ContractLog update module
:-eval (consult("./ContractLog/update.prova")).
% dynamically add a rule and two facts as module "id1"
a(X,Y,Z):-add(id1,"b(X,Y,Z):-c(Y), d(X,Z). c(_1). (_0,_2).", [X,Y,Z]).
% import the ContractLog list module
:-eval(add("./ContractLog/list.prova")).
% append two lists
:-solve(append([1,2],[3,4],Result)).
% size of list
:-solve(size([1,2,3,4],Length)).
% import the ContractLog math module
:-eval(add("./ContractLog/math.prova")).
% add two values
:-solve(math_add(1,1,Result)).
% Result = value1 mod value2
:-solve(math_mod(99,3,Result)).
% value1 <= value between <= value2
:-solve(between(1,5,10)).
% import the ContractLog datetime module
:-eval(add("./ContractLog/datetime.prova")).
timeInMillis(Epoch) :- sysTime(T), datetime_epoch(T,Epoch).
```

# Integrity Constraints (ICs)

---

- An IC is defined as a set of conditions that the constrained KB must always satisfy.
  - Satisfaction of an IC is the fulfillment to the conditions imposed by the constraint.
  - Violation of an IC is the fact of not giving strict fulfillment to the conditions imposed by the constraint.
- Four types of ICs:
  - **Not-constraints** which express that none of the stated conclusions should be drawn.
  - **Xor-constraints** which express that the stated conclusions should not be drawn at the same time.
  - **Or-constraints** which express that at least one of the stated conclusions must be drawn.
  - **And-constraints** which express that all of the stated conclusion must draw.
- Function: `integrity(< operator >, < conditions >)`
- Example:  
`integrity(xor(p(), q())) .`
- Meta Test Functions:
  - `testIntegrity()` test all integrity constrains in KB
  - `testIntegrity(<Literal>)` hypothetical test with literal, e.g. rule head

# Prova Agent Architecture

# Prova Agent Architecture

---

*Prova Agents Architecture is an intrinsic part of the Prova rule language providing reactive agent functionality.*

## **Prova-AA** offers

- ✓ message-oriented context-dependend reaction rules;
- ✓ message sending and receiving for local and remote communication actions;
- ✓ uniform handling of communication protocols (JMS, JADE, plus any of the more than 30 protocol supported by the Enterprise Service Bus).
- ✓ message payload as complex terms containing typed or untyped Java variables and serializable Java objects;
- ✓ state machine, Petri nets, or pi-calculus based conversation protocols;
- ✓ context-dependent inline reactions for asynchronous message exchange;
- ✓ ability to distribute mobile rulebases to remote agents;
- ✓ concurrent rule processing
- ✓ *Communicator* class for simplified embedding of Prova agents in Java and Web applications;
- ✓ Prova Universal Message Object gateway for reactive agents on ESB.
- ✓ Multi-threaded Swing programming with Prova Java calls and reaction rules.

# Messaging Reaction Rules

---

- Send a message

*sendMsg(XID, Protocol, Agent, Performative, [Predicate|Args]|Context)*

- Receive a message

*rcvMsg(XID, Protocol, Agent, Performative, [Predicate|Args]|Context)*

- Receive multiple messages

*rcvMult(XID, Protocol, Agent, Performative, [Predicate|Args]|Context)*

- Description:

- *XID is the conversation identifier*
- *Protocol: transport protocol e.g. self, jade, jms, esb*
- *Agent: denotes the target or sender of the message*
- *Performative: pragmatic context, e.g. FIPA ACL*
- *[Predicate|Args] or Predicate(Arg<sub>1</sub>, ..., Arg<sub>n</sub>): Message payload*

# Example: Remote Job/Task Scheduling

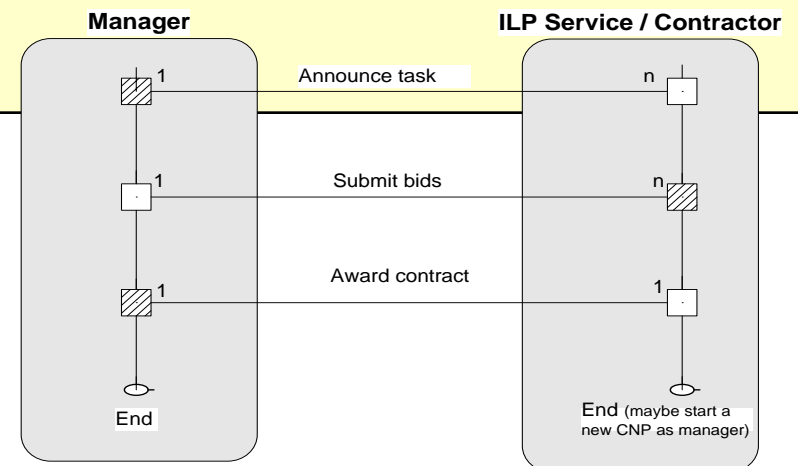
**% Manager**

```
upload_mobile_code (Remote, File) :  
  Writer = java.io.StringWriter(), % Opening a file  
  fopen (File, Reader),  
  copy (Reader, Writer),  
  Text = Writer.toString(),  
  SB = StringBuffer (Text),  
  sendMsg (XID, esb, Remote, query-ref, consult (SB)).
```

**% Service (Contractor)**

```
rcvMsg (XID, esb, Sender, eval, [Predicate | Args]) :-  
  derive ([Predicate | Args]).
```

Contract Net Protocol



# Programming with Prova

# Installing Prova

---

- ◆ *Please follow the Tutorials in your Lab 1 sheets*
- ◆ *Prova is available from Subversion version control system at [Sourceforge.net](https://sourceforge.net) as sub-project "**prova**".*

<https://mandarax.svn.sourceforge.net/svnroot/mandarax/prova>

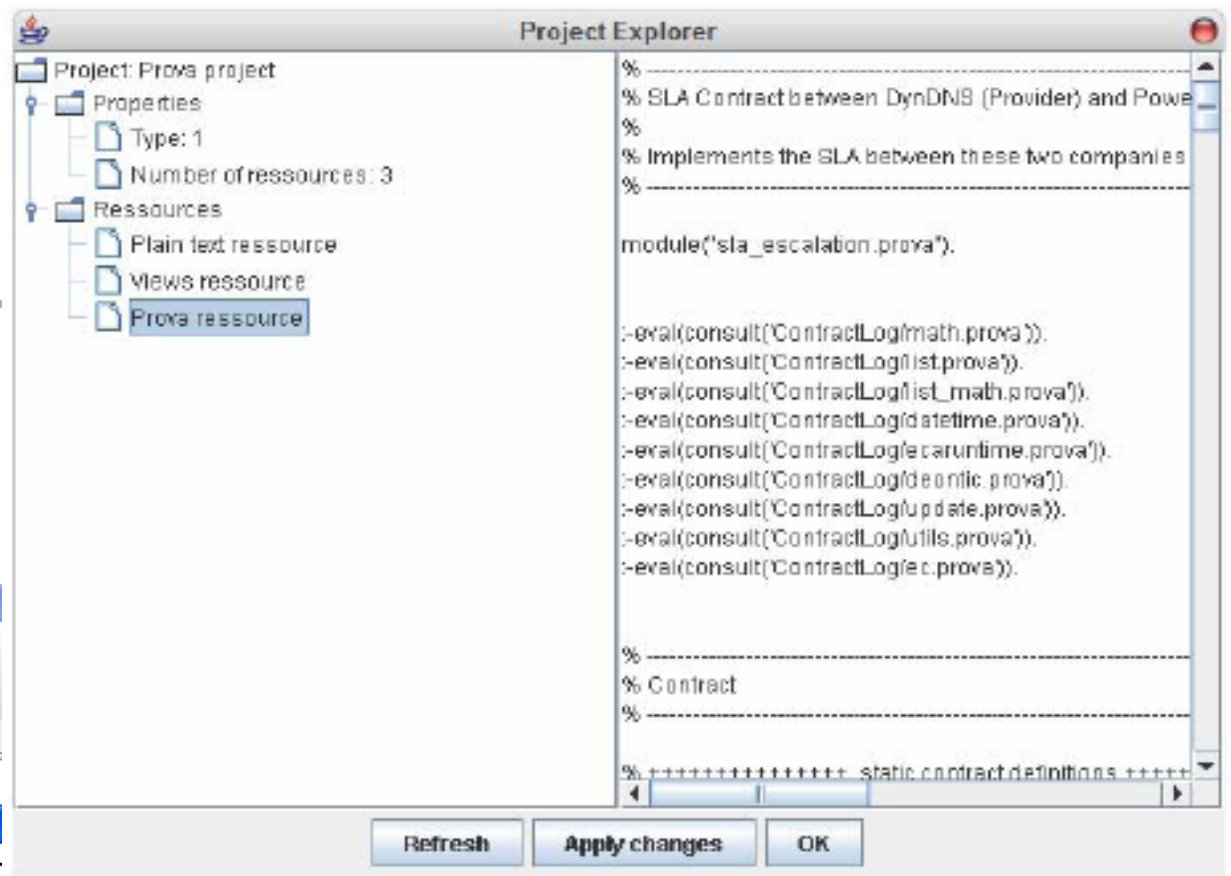
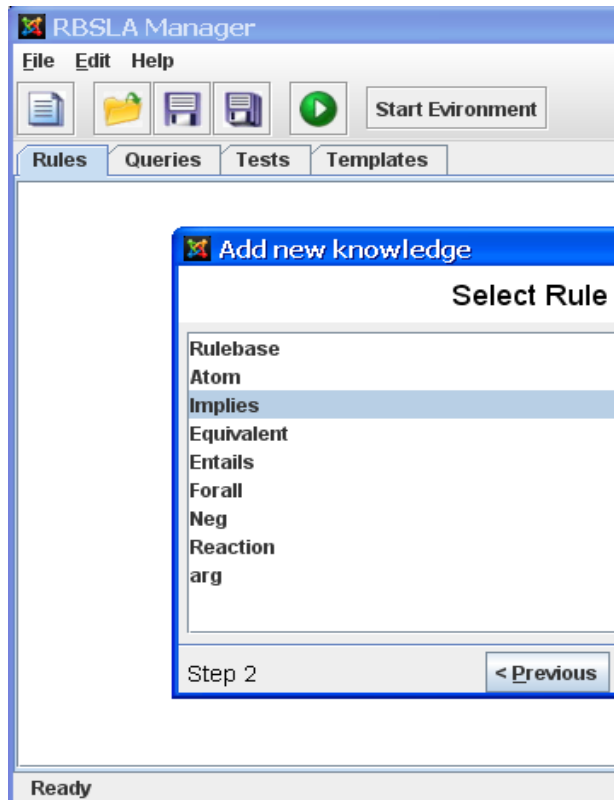
- ◆ *Frequently check the SVN for updates !!!*

# Running Prova

---

- ◆ *Prova provides an **Interpreted Scripting Syntax** + runs in Java*
- ◆ *To run the Prova scripts type on your console*
  - ./rules/prova + filename.prova***
  - ◆ *the prova script / batch file under **rules/prova/prova-examples** provides debugging logs*
- ◆ *You might run **ws.prova.reagent** directly from your Eclipse IDE*
  - ◆ *you need to add all required libraries under **./lib** to your project classpath*

# RBSLA Manager



1. **Runtime User Interface**
  - Vizualisation + Exploring
2. **Rule Editor / Project Manager**
  - (Authoring + Management)

# Implementing with Prova in Java

---

- *Prova controls the workflow.* The Prova engine is started from the command line specifying an initial rulebase file as an input.
- **Java controls the workflow.** An instance of the Prova engine is run from a Java application or server-side component.
- Calling Prova from Java with Prova Java Communicator

```
try {
    Communicator comm = new Communicator("prova", null,
                                         "./example.prova", -1, Communicator.SYNC);

    List resultSets = comm.consultSync(query, "", objects);
    Iterator rsit = resultSets.iterator();
    if (rsit.hasNext()) {
        ResultSet rs = (ResultSet) rsit.next();
        return rs;
    }
} catch (Exception e) {...}
```

# Complexity of Rule Languages (Examples)

- The number of primitives in the rule language
  - e.g., number of connectives used
- The depth of the syntax tree
  - e.g., the depth is restricted for logic programs without function symbols, but unrestricted if connectives and terms can be nested
- Restrictions in the rule language
  - e.g., no negations in rule heads
- Non-standard language elements and procedural elements
  - e.g., priorities, cut, different flavors of logical and procedural conjunctions, procedural attachments
- Different language elements with a similar meaning
  - weak and strong negation,
  - *OR* and *XOR*
- Polymorphic language elements
  - Polymorphic negation interpreted as a weak or strong negation depending on the predicate symbol of the negated atom
- The lack of a standard interpretation for certain language elements
  - Flavors of modal logic
  - Deontic modalities
  - Negation as Failure
- Cross-references between rules.
  - loops in the dependency graph between predicate symbols

➔ Simple Verification, Validation, and Integrity testing (V&V&I) techniques are needed !!!

# Basics in Verification&Validation&Integrity

---

- V&V&I Pattern:

Validation: *"Are we building the right program?"*

Verification: *"Are we building the program right?"*

Integrity : *"Are we keeping the program right?"*

- Errors and Anomalies

- Errors represent problems which directly effect the execution of rules, e.g. typographical errors, incompleteness, contradictions
- Anomalies are considered as symptoms of genuine errors

- Taxonomy of anomalies (Preece and Shinghal)

- Semantic checks, e.g., consistency and completeness
- Structural checks, e.g., redundancy, relevance and reachability

# Lessons learned from Software Engineering

---

- Different SE approaches and methodologies
  - e.g. Rational Unified Process (RUP) Modeling, Waterfall-based, V-Model, Model Checking, Algebraic Code Inspection, Structural / Operational Debugging etc.
- 1. Heavy-weight V&V
  - e.g. Waterfall-based, V-Model
  - Induce high costs of change
  - Can not check dynamic behaviors and interactions between dynamically updated and interchanged rule bases
- 2. Simple Structural / Operational Debugging
  - Instrument program and watch execution trace
  - Place huge cognitive load and deep understanding of the underlying processes on the user
- 3. Model Checking and Algebraic, Graph, Petri-Net based V&V Methods
  - e.g. Petri-Nets, Process Algebras, ACTL
  - Computationally very costly
  - Presuppose a deep understanding of both domains: Rule language and testing language/models
  - Often more complex than the rule base itself

# XP – Test Driven Development (1)

- Introduced in the late 90ties Ken Beck, Ward Cunningham, Erich Gamma and others.
- Test-driven Extreme Programming and agile SE
  - Empirical evidence that XP works better for small and medium projects

Layman, L.: “Empirical Investigation of the Impact of Extreme Programming Practices on Software Projects,”
  - Supported by industry, e.g. IBM
  - Neglects heavy-weight analysis and design but is nevertheless a formal approach
  - Speeds up development process , facilitate backtracking (redesign), collaboration of roles (domain experts, system developers, knowledge engineers), evolutionary modeling, program rule interchange, dynamic testing

# XP – Test Driven Development (2)

## Some ideas from test-driven development:

1. Write executable test cases first, i.e. write constraints which describe the intended models
  - Written in the programming language, no need for separate constraint language
  - Black box view
  - Abstraction from the program -> much simpler
  - Reusable for different programs / methods
2. Little upfront design but evolving design. Permanent redesign supported by refactoring (optimizing the structure but retaining the behavior)
  - Modify and adapt test cases and program rules in an iterative process according to changing requirements and detected faults (bugs)
3. Build often, extremely short iterations, tool supported builds (Ant JUnit), large tool support for coverage measurement, automated refactoring, dependencies analysis
4. Tests are constraints on the set of possible models of a program and therefore describe an approximation of the intended model(s)
  - The approximation level, i.e. the test coverage, is extended and dynamically adapted in an adaptive iterative process (adaptive modeling)
  - Better reflect the dynamics of many businesses nowadays than to big upfront design used in predictive modeling approaches
5. Test cases are managed, maintained and interchanged together with the program under test, are written in the same programming language and can be executed within the same execution environment

# Java JUnit Test Cases

---

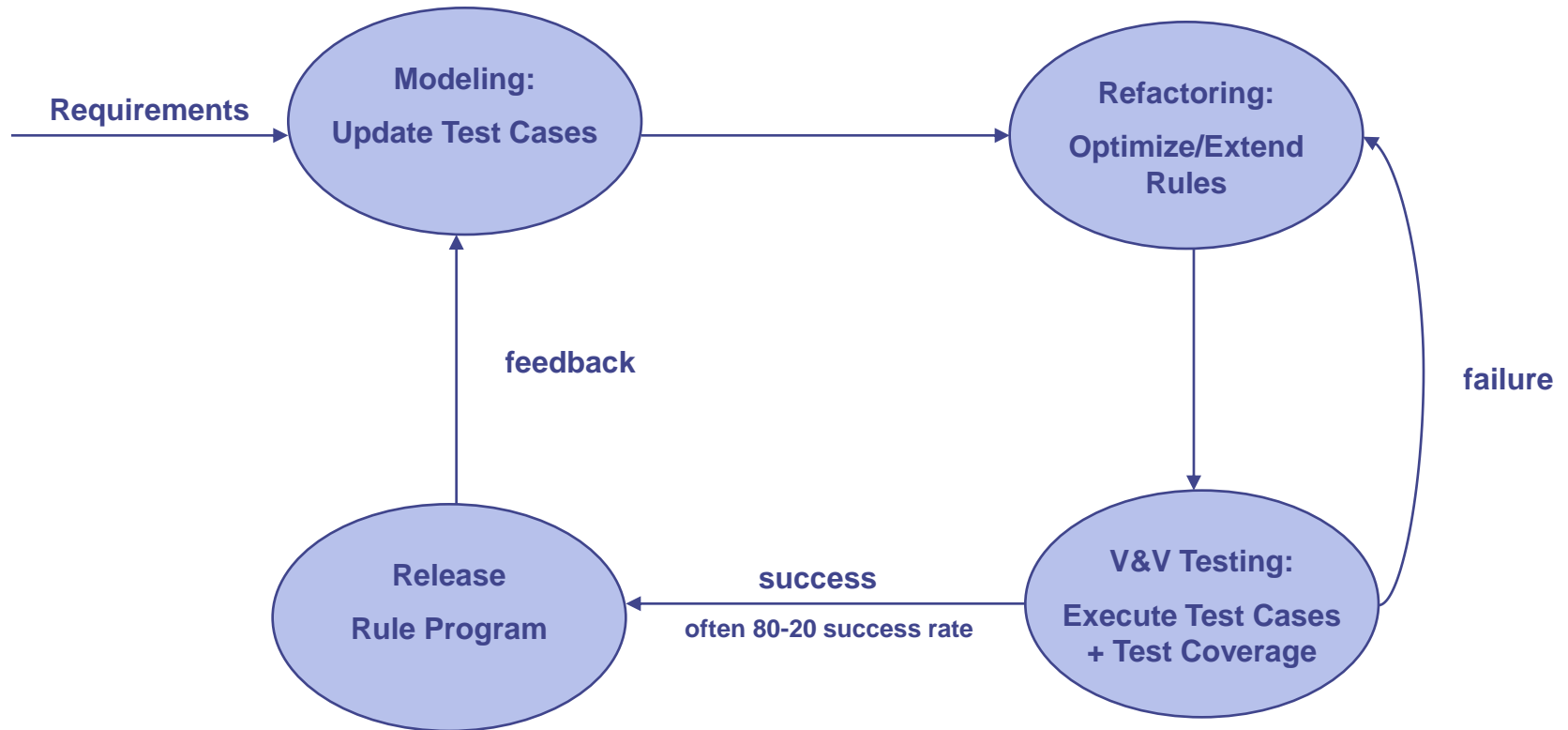
- Discipline of software development based on simplicity, communication and feedback,
  - Very short cycles of adding a test, then making it work.
- JUnit tests do not require human judgement to interpret, and is easy to run many of them at the same time:
  1. Create an instance of TestCase (JUnit.jar):
  2. Create a constructor which accepts a String as a parameter and passes it to the superclass.
  3. Override the method runTest()
  4. When you want to check a value, call assertTrue() and pass a boolean that is true if the test succeeds
- More Information:
  - <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>
  - <http://members.pingnet.ch/gamma/junit.htm>
  - <http://junit.sourceforge.net/doc/cookstour/cookstour.htm>

# Test Cases for Rule Bases

- Rules constrain the set of possible models (possible worlds)
  - But are relatively complex (as argued before)
- Test Cases constrain the possible models and approximate the intended models of the rule base engineer
  - Queries are used to test the rule base
  - they are inherently simple, e.g. often ground (no variables), flat (no functions), no negations
- A test case is defined by  $TC := \{A, T_i\}$ , where
  - $A \subseteq L$  assertion base (input data, e.g. facts )
  - $T_i \in L$  ,  $i > 0$ , a set of formula denoting test queries, where  $T_i$ :
    1. A test query  $Q := q(t_1, ..t_n)?$ , where  $Q \in \text{rule}(P)$
    2. A result  $R$  being either a positive "*true*", negative "*false*" or "*unknown*" label
    3. answer set of expected variable bindings  
 $\theta := \{\{X_1/a_1, X_1/a_2, \dots, X_1/a_n\}, \dots, \{X_m/c_1, \dots, X_m/c_k\}\}$
- $T = A \cup \{Q \Rightarrow R : \theta\}$  or simply  $T = \{Q \Rightarrow R : \theta\}$  if no assertions
- Example:  
 $T_1 = \{p(X) \Rightarrow \text{true} : \{X/a, X/b, X/c\}, q(Y) \Rightarrow \text{false}\}$

# Test-driven Development Feedback Loop

---



# ContractLog LP Cover

---

- ContractLog Logic Programming Coverage Suite
- Test Cases and Test Suites are written as Prova scripts and can be stored/published and interchanged together with the rule programs
- Integration into Maven tests
- Automated JUnit Test Reports
- Test Coverage reports in ContractLog KR

# Example Test Case

---

```
% testcase oid
testcase("./examples/tc1.test").

% assertions via updates adding one rule and two facts
:-solve(add("tc1.test","a(X):-b(X). b(1). b(2).")).

% positive test with success message for JUnit report
testSuccess("test1","succeeded"):-
    testcase("./examples/tc1.test"), testQuery(a(1)).

% negative test with failure message for Junit report
testFailure("test1","can not derive a"):-
    not(testSuccess("test1",Message)).

% define the active tests - used by meta program
runTest("./examples/tc1.test"):-testSuccess("test 1",Message).
```

# Prova's Polymorphic Order Sorted Typed Logic

# Types in LP Rule Languages

---

- Add Software Engineering principles such as data abstraction or modularization
  - Vital for the development and maintenance of large distributed rule-based systems
- Distributed System Engineering and Collaboration
  - Domain-independent rules need to be interchanged and given a domain dependent meaning in their target environments by domain-dependent vocabularies / ontologies
- Types can be considered as an approximation of the intended interpretation; They allow programming of functional rule variants e.g. via ad-hoc polymorphism enabling overloading and coercion by type casting
- They constrain the level of generality in queries and lead to much smaller search spaces and therefore improve the execution efficiency of query answering.

# OWL2Prova Homogeneous Integration

---

1. Different reasoners
  - *Transitive, RDFS, OWL, OWL Mini*
  - Precompile RDF/RDFS/OWL model into inferred triple statement model
2. Converters are used to translate triple statements into arbitrary LP facts / rules
  - SimpleConverter translates to arbitrary outputs according to user-defined patterns, e.g:
    - ["predicate", "subject", "object"] => predicate(subject,object)
    - ["rdf", "subject", "object"] => rdf(subject,object)
    - ["rdfTriple", "subject", "predicate", "object"] => rdfTriple(subject,predicate,object)
  - DLPCConverter translates to DLPs:
    - $C(X) \rightarrow D(x)$  class C is subclassOf class D
    - $P(x,y) \rightarrow Q(x,y)$  property P is a subproperty of property Q
    - $P(x,y) \rightarrow C(y)$  range of property P is class C
    - $C(X) \rightarrow D(X)$
    - $D(X) \rightarrow C(X)$  Class C and Class D are equivalent
    - ...

(Note: **Needs loop checker** – OWL2Prova adds a linear of memoization of critical subgoals in order to prevent loops)
  - DefeasibleDLPCConverter translates into defeasible DLPs
  - Further user-defined converters can be easily added, extending the OWL2Prova Converter interface
3. The precompiled and translated models can be added to an existing KB and the rule engine is used for querying

# OWL2Prova Converter Example

---

## Ontology

```
...  
<owl:Class rdf:ID="Red_Wine">  
  <rdfs:subClassOf rdf:resource="#Wine"/>  
</owl:Class>  
...
```

## Homogeneous Integration

```
:-eval(consult('ContractLog/owl.prova')).  
  
% translate and include OWL Facts  
:-eval(assertOWL(  
  "", % no reasoner  
  ["triple", "subject", "predicate", "object"],  
  "./WineProjectOWL.owl")). % input file  
  
% query all triples  
:-solve(triple(Subject, Predicate, Object)).
```

## Translated facts

```
... triple(wine_Red_Wine, rdfs_subClassOf, wine_Wine). ...
```

# OWL2Prova DL-Typed Hybrid Description Logic Programs

---

- **Hybrid integration**
  - of DL ontologies serialized in OWL Lite / DL (or RDFS) into Rules as types of logical terms
- **Prescriptive Typing Approach: “*term:type*”**
  - Fresh individuals are allowed in rule heads
  - Free DL-typed variables are allowed in facts → Query on *known* individuals in A-box
- **Dynamic Type Checking**
  - Via calling external DL reasoner for e.g. *subsumption inference*, *instantiation inference*, *equivalence mapping* etc.
  - Caches inferred models for faster access on DL knowledge base
- **Polymorphic Order-Sorted Unification**
  - Ad-hoc polymorphism: Variables might change their types during unification
  - Type-casting in the sense of coercion is supported by typed unification
- **Supports different Semantic Web Type Systems:**
  - RDFS, OWL-Lite, OWL-DL , DL ontol.

# Typed Unification Rules – Operational Semantics

---

- **Untyped Unification**
  - Ordinary untyped unification without type checking
- **Untyped-Typed Unification**
  - The untyped query variable assumes the type of the typed target
- **Variable-Variable Unification:**
  - If the query variable is of the same type as the target variable or belongs to a subtype of the target variable, the query variable retains its type, i.e. the target variable is replaced by the query variable.
  - If the query variable belongs to a super type of the target variable, the query variable assumes the type of the target variable, i.e. the query variable is replaced by the target variable.
  - If the query and the target variable are not assignable the unification fails

# Typed Unification Rules – Operational Semantics

---

- Variable-Constant Term Unification:
  - If a variable is unified with a constant of its super-type, the unification fails.
  - If the type of the constant is the same or a sub-type of the variable, it succeeds and the variable becomes instantiated.
- Constant-Constant Term Unification:
  - Both constants are equal and the type of the query constant is equal to the type of the target constant.
- Complex terms such as lists are untyped by default and hence are only allowed to be unified with untyped variables resp. variables of type "Resource".

# Example (1)

---

## **% Imports**

```
import("http://.../dl_typing/businessVocabulary1.owl").
import("http://.../dl_typing/businessVocabulary2.owl").
import("http://.../dl_typing/mathVocabulary.owl").
import("http://.../dl_typing/currencyVocabulary.owl").
```

```
reasoner("dl"). % configure reasoner (OWL-DL=Pellet)
```

## **% Rule-based Discount Policy**

```
discount(X:businessVoc1_Customer, math_Percentage:10) :-
    gold(X: businessVoc1_Customer).
discount(X: businessVoc1_Customer, math_Percentage:5) :-
    silver(X: businessVoc1_Customer).
discount(X: businessVoc1_Customer, math_Percentage:2) :-
    bronze(X: businessVoc1_Customer).
```

## Example (2)

---

```
% Note that this rules use a different vocabulary
% if the classes "Client" and "Customer" are equal
% both typed rule sets unify
% Class equivalence between both "types"
% is defined in the second OWL-DL ontology
```

```
gold(X:businessVoc2_Client) :-
    spending(X:businessVoc2_Client, S:currency_Dollar),
    S:currency_Dollar > currency_Dollar:1000.
```

```
silver(X:businessVoc2_Client) :-
    spending(X:businessVoc2_Client, S:currency_Dollar),
    S:currency_Dollar > currency_Dollar:500,
    S:currency_Dollar < currency_Dollar:1000.
```

```
bronze(X:businessVoc2_Client) :-
    spending(X:businessVoc2_Client, S:currency_Dollar),
    S:currency_Dollar > currency_Dollar:100,
    S:currency_Dollar < currency_Dollar:500.
```

# Example (3)

---

## **% Facts**

```
spending (businessVoc1_Customer:Adrian, currency_Dollar:1100) .
```

```
spending (businessVoc2_Client:Aira, currency_Dollar:200) .
```

## **% Query**

```
:-solve (discount (X:businessVoc2_Client,  
Y:math_Percentage)) .
```

## **Result:**

```
X:businessVoc2_Client = businessVoc1_Customer:Adrian
```

```
Y:math_Percentage = math_Percentage:10
```

```
X:businessVoc2_Client = businessVoc1_Customer:Aira
```

```
Y:math_Percentage = math_Percentage:2
```

# Types in RuleML

---

- Types can be assigned to terms using the type attribute
- Types are valid for <Ind>, <Var> and <Cterm> terms
- <Plex>s and <Atom>s cannot be given types
- Examples:

```
<Var type="Vehicle" />
```

```
<Ind type="Sedan ">
```

```
    2000 Toyota Corolla
```

```
</Ind>
```

# Advantages

---

- Static and dynamic type checking
- Supports SE principles such as data abstraction or modularization for large distributed unitized rule bases
- Reduces search space of queries on typed rule sets
- Extends expressive power of rules with Description Logics KR
- Exploits external optimized DL
- Caches inferred models for faster access on DL knowledge base
- Allows ad-hoc polymorphism with overloading and coercion
- Integration of domain-specific Semantic Web ontologies (vocabularies) into domain-independent rule specifications and executions
  - Facilitates Rule Interchange
  - Facilitates Collaboration
  - Facilitates Distributed Management

# Prova Reaction Rules

# Messaging Reaction Rules

---

- Send a message

*sendMsg(XID, Protocol, Agent, Performative, [Predicate|Args]|Context)*

- Receive a message

*rcvMsg(XID, Protocol, Agent, Performative, [Predicate|Args]|Context)*

- Receive multiple messages

*rcvMult(XID, Protocol, Agent, Performative, [Predicate|Args]|Context)*

- Description:

- *XID is the conversation identifier*
- *Protocol: transport protocol e.g. self, jade, jms, esb*
- *Agent: denotes the target or sender of the message*
- *Performative: pragmatic context, e.g. FIPA ACL*
- *[Predicate|Args] or Predicate(Arg<sub>1</sub>, ..., Arg<sub>n</sub>): Message payload*

## Example: "Request – Response" Flow with Delegation

---

```
% receive query and delegate it to another agent
rcvMsg(CID, esb, Requester, acl_query-ref, Query) :-
    ... (other goals) ...
    sendMsg(Sub-CID, esb, Agent, acl_query-ref, Query),
    rcvMsg(Sub-CID, esb, Agent, acl_inform-ref, Answer),
    ... (other goals) ...
    sendMsg(CID, esb, Agent, acl_inform-ref, Answer).
```

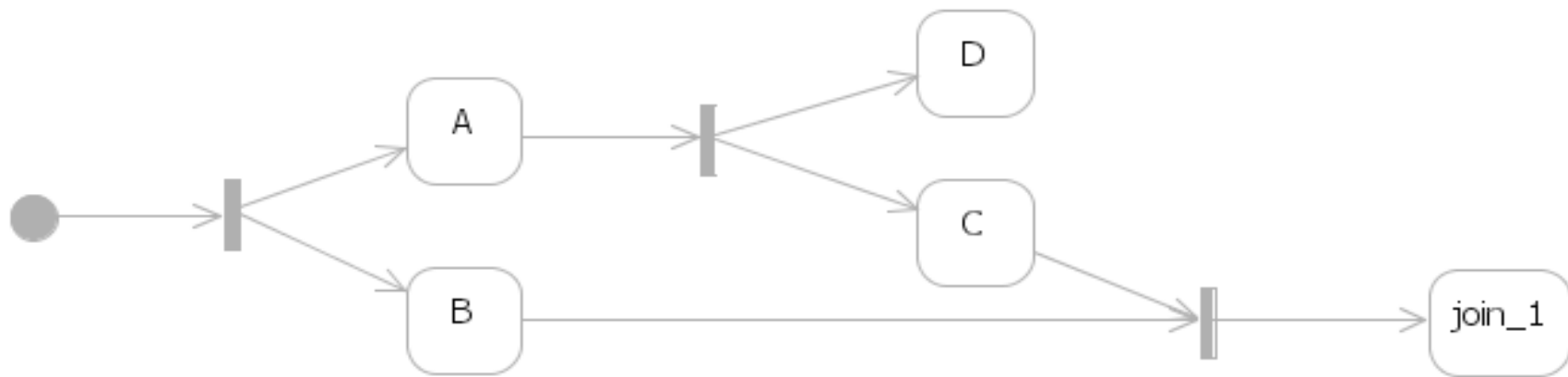
```
% answers query "Agent"
rcvMsg(XID, esb, From, Performative, [X|Args]) :-
    derive([X|Args]),
    sendMsg(XID, esb, From, answer, [X|Args]).
```

# Example: Mobile Code

---

```
% Manager
upload_mobile_code(Remote,File) :
  Writer = java.io.StringWriter(), % Opening a
  file fopen(File,Reader),
  copy(Reader,Writer),
  Text = Writer.toString(),
  SB = StringBuffer(Text),
  sendMsg(XID,esb,Remote,eval,consult(SB)).
```

```
% Worker Service (Contractor)
rcvMsg(XID,esb,Sender,eval,[Predicate|Args]) :-
  derive([Predicate|Args]).
```



```

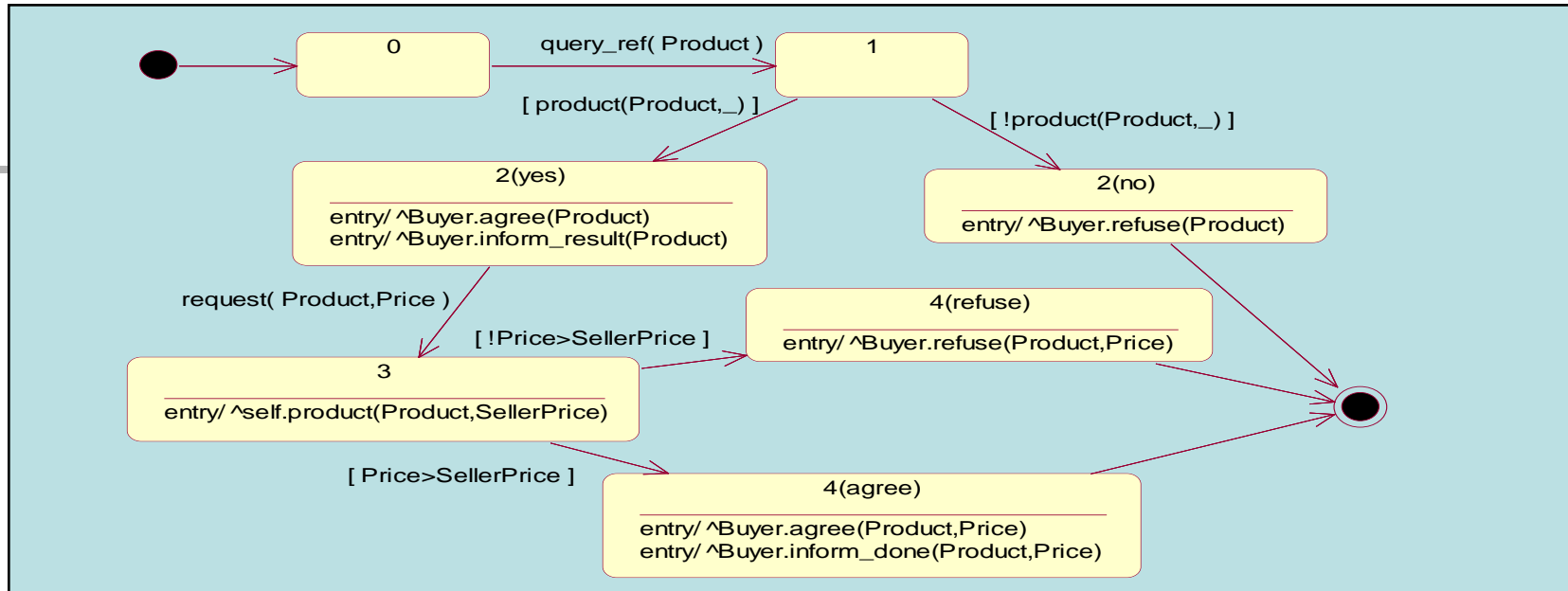
process_join() :-
iam(Me) ,
init_join(XID,join_1,[c(_),b(_)]),
fork_a_b(Me,XID) .
fork_a_b(Me,XID) :-
rcvMsg(XID,self,Me,reply,a(1)) ,
fork_c_d(Me,XID) .
fork_a_b(Me,XID) :-
rcvMsg(XID,self,Me,reply,b(1)) ,
join(Me,XID,join_1,b(1)) .
fork_c_d(Me,XID) :-
rcvMsg(XID,self,Me,reply,c(1)) ,
% Tell the join join_1 that a new pattern is ready
join(Me,XID,join_1,c(1)) .

% The following rule is invoked by join once all the inputs are assembled.
join_1(Me,XID,Inputs) :-
println(["Joined for XID=",XID," with inputs: ",Inputs]) .

% Prints
% Joined for XID=agent@hostname001 with inputs [[b,1],[c,1]]

```

# State machines based conversations



```

directbuy_seller_1 (XID,Protocol,From,Product) :-
product (Product|_),
!,
sendMsg (XID,Protocol,From,agree,Product,seller),
sendMsg (XID,Protocol,From,inform_result,Product,seller),
directbuy_seller_2 (yes,XID,Protocol,From,Product).
directbuy_seller_1 (XID,Protocol,From,Product) :-
sendMsg (XID,Protocol,From,refuse,Product,seller),
directbuy_seller_2 (no,XID,Protocol,From,Product).
directbuy_seller_2 (yes,XID,Protocol,From,Product) :-
!,
rcvMsg (XID,Protocol,From,request,[Product,Price],buyer),
product (Product,SellerPrice),
directbuy_seller_3 (XID,Protocol,From,Product,Price,SellerPrice).
directbuy_seller_2 (no,XID,Protocol,From,Product).

```

# Summary

---

## ***Prova*** offers

- ✓ External Data and Object Integration + Query Built-Ins
  - Java Integration
  - XML Integration
  - SQL Integration
  - RDF Integration
- ✓ External Type Systems: Order-Sorted Polymorphic Typed Logic
  - Java Class Hierarchies
  - Semantic Web Ontologies
- ✓ Input/Output Mode Declarations
- ✓ Module Import and Integration: Order Modularized Logic Programs
- ✓ Meta Data Labels and Scopes (constructive views)
- ✓ Integrity Constraints and Test Cases for Verification and Validation
- ✓ Backward-reasoning derivation rules + forward-processing reaction rules
- ✓ Messaging Reaction Rules and Complex Event Processing
- ✓ State machine, Petri nets, or pi-calculus based conversation protocols;
- ✓ Dynamic Transactional Updates

<http://prova.ws>



## Prova: A Language for Rule Based Java Scripting, Information Integration, and Agent Programming

Alex Kozlenkov<sup>(1)</sup>, Adrian Paschke<sup>(2)</sup>, and Michael Schroeder<sup>(3)</sup>

(1) Advanced Technology Group of [Betfair](#), London (2) Technical University, Munich (3) Biotec, Dresden

### home

News

Discussion forum

Collaboration site (Wiki)

Downloads

Language grammar

### associated projects

RuleML


GoPubMed

Mandarax

RBSLA

Jena

### The current version is Prova 2.0 RC2 (Update 8) of March 30, 2008

You can now subscribe to the [Prova News feeds](#). 

*March 30, 2008:* Prova 2.0 Update 8 is available from the Subversion repository at Sourceforge.net (see details [here](#)). "Predicate" (conditional) JOIN can now run concurrently. Semantics of *sendMsg* are changed to avoid possible race condition with subsequent inline reactions *rcvMsg*. Important bug fixes are also included.

*March 10, 2008:* Prova 2.0 Update 7 is available from the Subversion repository at Sourceforge.net (see details [here](#)). Prova can now safely process multiple queries or inbound messages concurrently using a thread pool. A new built-in *increment\_value* facilitates concurrent atomic changes to data stored as triples. Parallel rule stream process "JOIN" is also made thread-safe. We'll keep you posted on Prova concurrency in a separate story here shortly.

*March 31, 2007 (Update 4 on April 15, 2007):* Prova 2.0 source code distribution is now available from the Subversion repository at Sourceforge.net (see all details [here](#)). Prova Workflows for [Mule Enterprise Service Bus](#) is also available as a subproject from the same repository (the direct [archive download](#) is also available). Prova Workflows for Mule now include the following examples of [Workflow Patterns](#): Join, Simple Merge, Cancel Activity, Multi-Choice, Structured Loop, Deferred Choice and Milestone.

*March 26, 2007 (updated April 15, 2007):* [Prova 2.0 RC1](#), the first Maven 2.0 based distribution, is available [locally](#). [Prova Forum](#) and [Prova collaboration site](#) will continue providing more details about the features and enhancements in the new version.

*June 18, 2006:* Xcalia product "Xcalia Core for Services 4.3.0" includes Prova runtime and specialised Prova rule base used for efficiently computing global execution plans across multiple disparate data sources, such as Web services, TP monitors transactions like CICS or IMS, messages of MOM like MQ-Series, packaged applications with a JCA connector, legacy data sources on mainframes with a JCA connector, remote EJB Java objects considered as data providers or even local Java objects (see [Introduction/Third-party licenses](#)).

Quoting from Xcalia testimonial: "Prova is the critical part of our new XIC for Services product, responsible for dynamic orchestration of services. We have found Prova to be an excellent rule language using Java, with complete functionality and good performance. The Prova team has been very reactive and helpful."

# Links

---

- Prova: <http://www.prova.ws/>
- Paschke, A.: Rule-Based Service Level Agreements - Knowledge Representation for Automated e-Contract, SLA and Policy Management, Book ISBN 978-3-88793-221-3, Idea Verlag GmbH, Munich, Germany, 2007.