



Reasoning About Objects

Jens Dietrich

Massey University

Institute of Information Sciences and Technology

j.b.dietrich@massey.ac.nz

Summary

1. Introduction
2. Building Rules With Objects
3. Rules as Service
4. Validating and Verifying Rules
5. Integrating Rules
6. Executing Rules
7. Serializing Rules

Introduction: Combing OO and LP: F-Logic

- Alternative paradigms, LP and OO as contradicting programming paradigms.
- Existing research to merge OO and LP started the 90ties (deductive, OO databases):
 - O-Logic [Maier 86]
 - F-Logic [Kifer, Lausen, Wu 95]
- In recent years: OO is **the** mainstream programming paradigm, increasing interest in rule technology to make software more agile.
- In particular to represent business logic in application areas such as public administration, investment banking, risk management.
- Efforts started to define standards for rules (RuleML, OMG, JSR-94).

Our Approach

- Practical Approach – working with real world objects.
- Accepting constraints imposed by design patterns (like object creation).
- Suitable for distributed applications.
- Focus on re-using and harvesting facts and rules.
- Focus on API development (not implementation, although it exists).
- No attempt to create a programming language.

Rules are already there

```
/**
 * Compares two dates for equality.
 * @param  obj    the object to compare with.
 * @return true if the objects are the same;
 *         false otherwise.
 */
public boolean equals(Object obj) {
    return obj instanceof Date &&
           this.getTime() == ((Date) obj).getTime();
}
```

The equals() method in java.util.Date

```
IF isDate(obj1) AND isDate(obj2) AND (getTime(obj1)==getTime(obj2))
THEN equals(obj1,obj2)
```

Written as rule ..

Rules are already here (ctd)

```
IF isDate(obj1) AND isDate(obj2) AND (getTime(obj1)==getTime(obj2))  
THEN equals(obj1,obj2)
```

- Typed logic with classes as types and constant terms obj1 , obj2.
- Complex terms, with getTime as function.
- Unary predicate isDate().
- Execution model is **backward reasoning** :
 - query is equals(obj1,obj2) (issued when calling obj1.equals(obj2))
 - answer is true or false.
- Problems:
 - methods might return an exception type (even if not declared, e.g. a java runtime exception).
 - terms in query must be ground.
 - derivation is not exposed.

.. and should not be re-invented

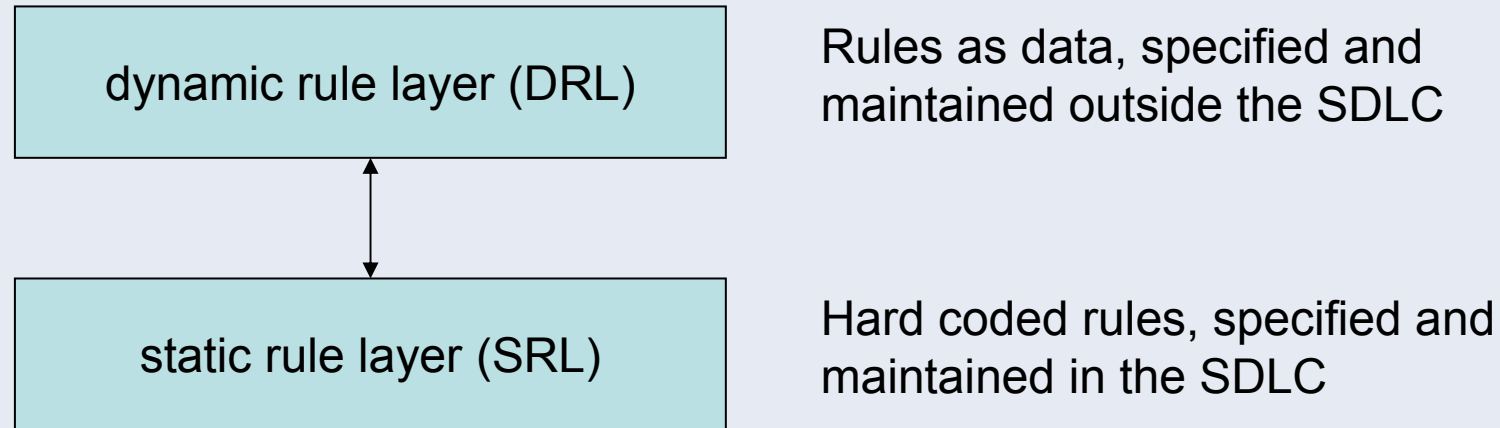
- re-use as design goal for OO-inference systems
- hardcoded rules represent static rules which are less likely to change
- It is **plumbing**, done by professional plumbers (= programmers)

Why extra rules on top of this

- Implementing high – level (business) logic that is likely to change.
- Short-cuts SDLC (software development life cycle).
- Enables separate people within the organization to maintain business rules, and reduces communication overhead between business experts, business analysts, designers and programmers.
- High-Level rules as services attached to code which can be configured by a specialized group of people within the respective organization.

Rule Layers

Therefore, we are dealing with two separate rule layers:



Questions:

- How can rules in the DRL be implemented (object model) ?
- How can rules in both layers be integrated and the differences be hidden (transparency) ?

From Objects to Logic: Constants and Variables

- Arbitrary objects considered as constant terms. Classes are types.
- Due to inheritance, objects can have more than one types. But there is a unique primary type.
 - $\text{type}(o, \text{primary_type}(o))$
 - if $\text{type}(o, T_1)$ & $\text{superclass}(T_2, T_1)$ then $\text{type}(o, T_2)$
- Objects can be stringified (e.g., to build formulas) with (logical) object ids "42:Integer", "New Zealand:Country".
- It is convenient to exclude certain classes, e.g. if predicate are defined as objects, to circumvent problems with higher order logic.
- Classes are reified (objects themselves).
- Variable terms are pairs (name,C) consisting of a name (a string) and a class (the primary type of the term).

From Objects to Logic: Functions and Complex Terms

- Methods m defined in a class C_0 with parameter types $(C_1..C_n)$ and a return type C_{return} define **functions**:
 $f_m: C_0 \times C_1 \dots \times C_n \rightarrow \{C_{\text{return}}\} \cup \text{EXCEPTION_TYPES}$
 $(C_0 \times C_1 \dots \times C_n)$ is called the signature of f .
- **Complex terms**: if f is a function as defined above, t_0, t_1, \dots, t_n are terms and $\text{type}(t_i, C_i)$ for each i , then $f(t_0, \dots, t_n)$ is a complex term with a primary type C_{return} .
- Ground complex terms can be **simplified**: if $f_m(t_0, \dots, t_n)$ is ground and (t_0, \dots, t_n) are constant (objects) replace this term by the result of invoking the method m (in OO syntax: $t_0.f_m(t_1, \dots, t_n)$). This can be applied recursively.
- Example: simplify “2+3” to “5” and “n*(4+2)” to “n*6”.

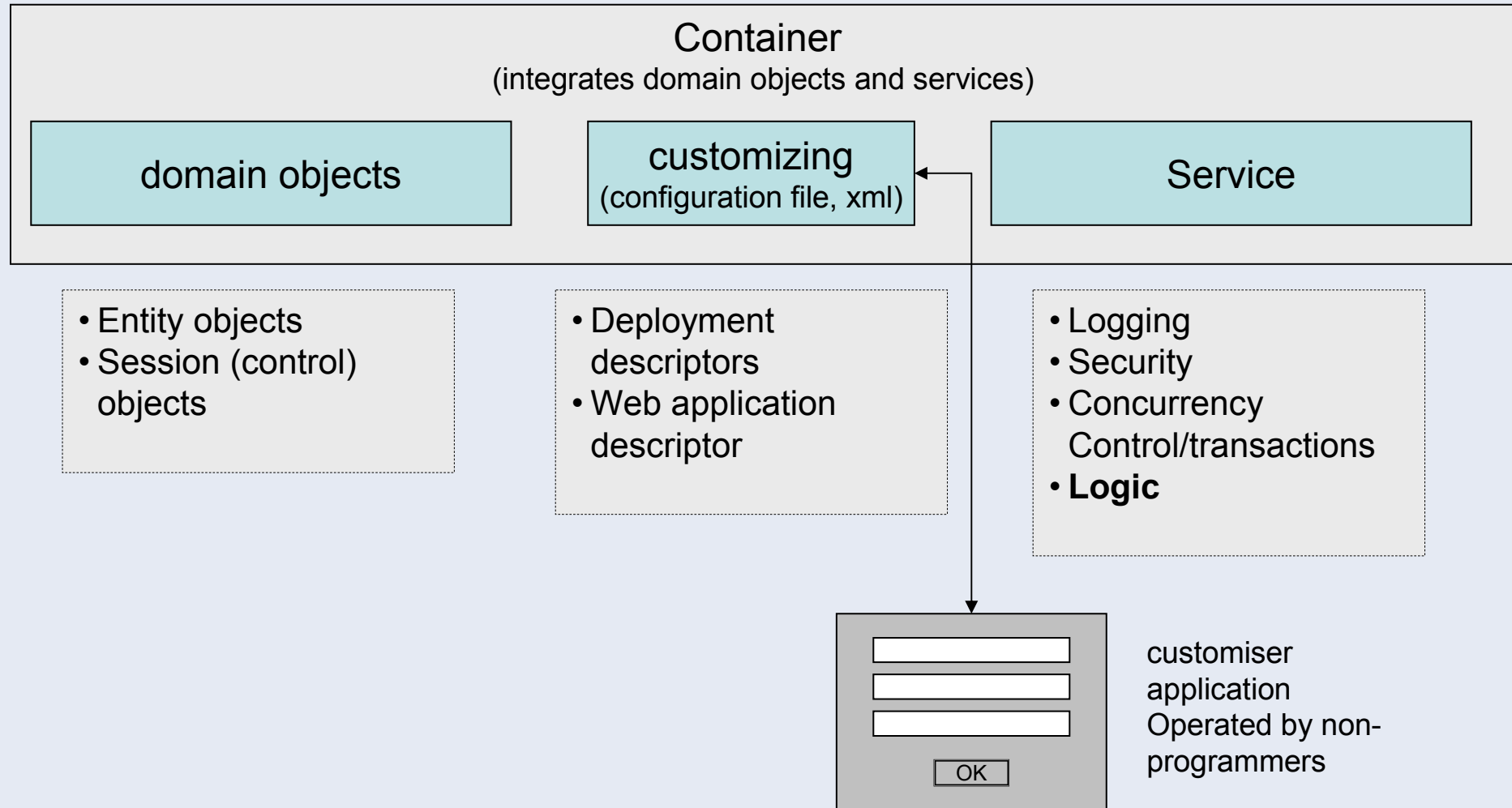
From Objects to Logic: Predicates, Literals and Rules

- **Predicates** p consists of a predicate name and a signature $\text{sign}(p) = (C_1, \dots, C_n)$.
- **Atoms**: $p(t_1, \dots, t_n)$, with $\text{sign}(p) = (C_1, \dots, C_n)$ and $\text{type}(t_i, C_i)$.
- **Literals**: atoms + negated atoms.
- **Rules**: $\text{prerequisite}_1, \dots, \text{prerequisite}_n \rightarrow \text{conclusion}$,
prerequisite are literals and the conclusion is an atom. Rules with empty list of prerequisites are called **facts**.
- **Queries**: list of atoms.
- **Knowledge base**: list of rules + list of queries.

Methods as Predicate

- Methods m returning boolean define a predicate p_n .
- Atoms: $p(t_1, \dots, t_n)$, with $\text{sign}(p) = (C_1, \dots, C_n)$ and $\text{type}(t_i, C_i)$.
- These predicates are called **S-predicates**. Literals with S-predicates are called **S-literals**.
- Special property: if S-literals are ground, the method can be invoked and “true” or “false” can be computed.
- Integration point between SRL and DRL.

Rules as Service



Challenges

1. Validation and verification that rules do the right thing (describe the intended model).
2. Saving and exchanging rules.
3. Smooth integration of rules into the OO landscape.
4. Some understanding of rule execution (loops, order of rules and order of prerequisites in rules might matter, performance critical issues like extensive use of negation as failure).
5. Verbalization of rules.

Verifying and Validating the Rules

- **Validation** .. refers to a set of activities that ensure that the software that has been built is traceable to customer requirements. Are we building the right product?
- **Verification** .. refers to a set of activities that ensure that the software correctly implements a specific function. “Are we building the product right? [Pressman: Software Engineering 5th Ed.]
- In OOA (RUP), rules usually captured in use case descriptions. OCL can also be used.
- Agile SE (in particular extreme programming, [Beck 97]) introduces test driven development: requirements are translated into re-usable and executable test cases. This greatly facilitates validation, and has revolutionized SE practice.

Using Unit Tests

- XUnit family of tools introduced by Beck/Gamma (starting with **SUnit** for Smalltalk and **JUnit** for Java)) has quickly developed into de facto standard tool for software validation and verification.
- Tests are written early (before domain objects are implemented) to specify the meaning of the objects (while their syntax is defined by abstract classes/interfaces).
- Using test cases to validate rules means not only to externalize rules, but parts of the software development process with them.

“Test-first coding is not a testing technique.” —Ward Cunningham

- This includes not only specification and testing but also deploying.

Unit testing Rules

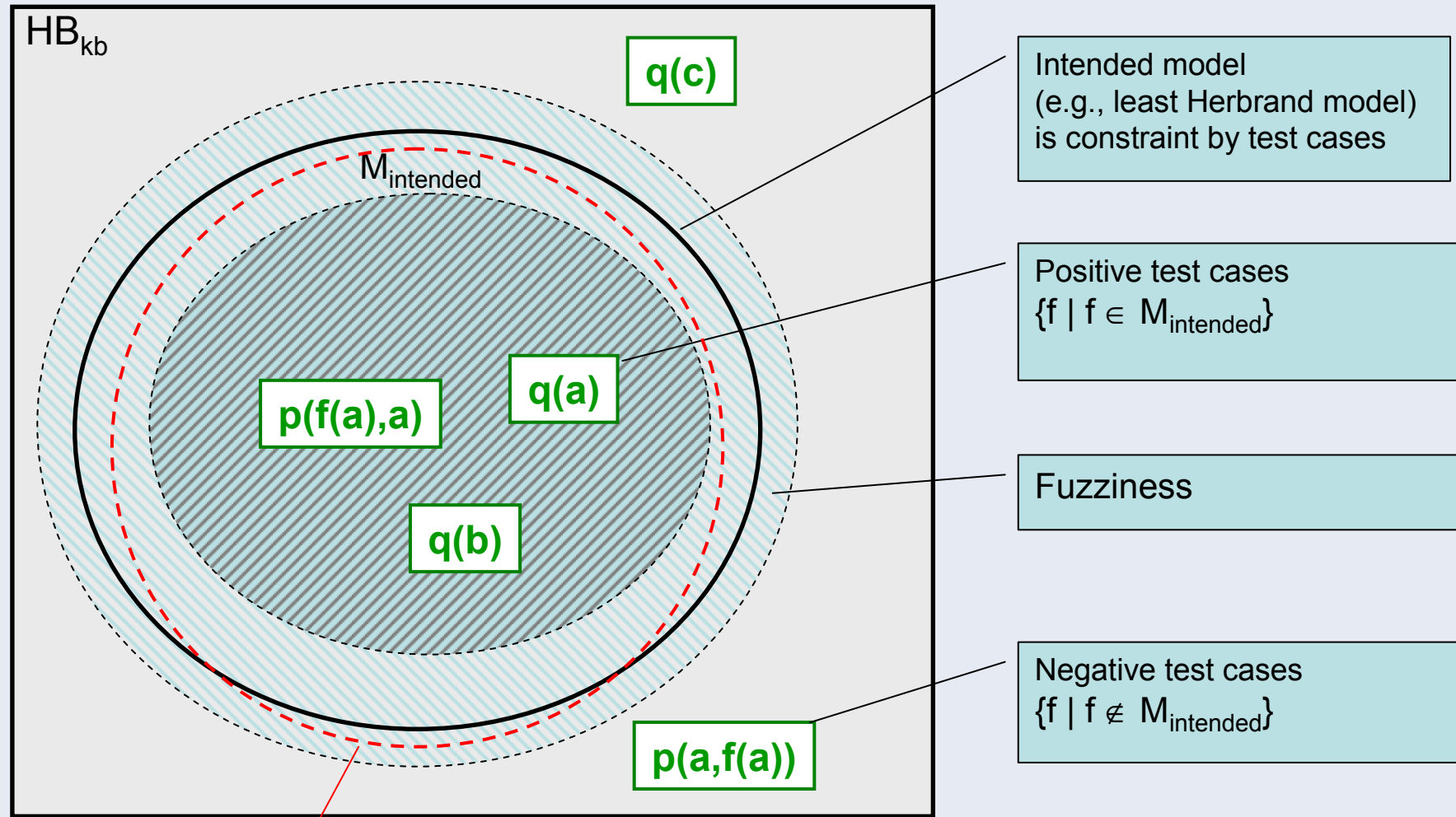
Unit tests for rules consist of are queries and expected results (provided a query based execution model).

1. Set up a test fixture (set of facts or other objects) – this is the test environment or context.
2. Build a query.
3. Add expected results (expected substitutions for query variables, or a boolean if query is ground).
 - Positive test cases – bindings we expect to find.
 - Negative test cases – bindings we don't expect to find.
4. Run query.
5. Compare computed and expected results.
6. Clean up / release test fixture.
7. Return test result (SUCCESS, FAILURE or ERROR).

The Herbrand Base

- The Herbrand base is used to establish the semantics of the rules.
- Precise semantics (how to select the model(s) from the Herbrand base) not given here, different semantics possible (in particular if negation is present in rules: well founded semantics, stable models etc).
- Herbrand universe HU_{kb} : terms built from constants used (referenced) in the rules of a knowledge base.
- Herbrand base HB_{kb} : atoms built with predicates and terms from the Herbrand universe.

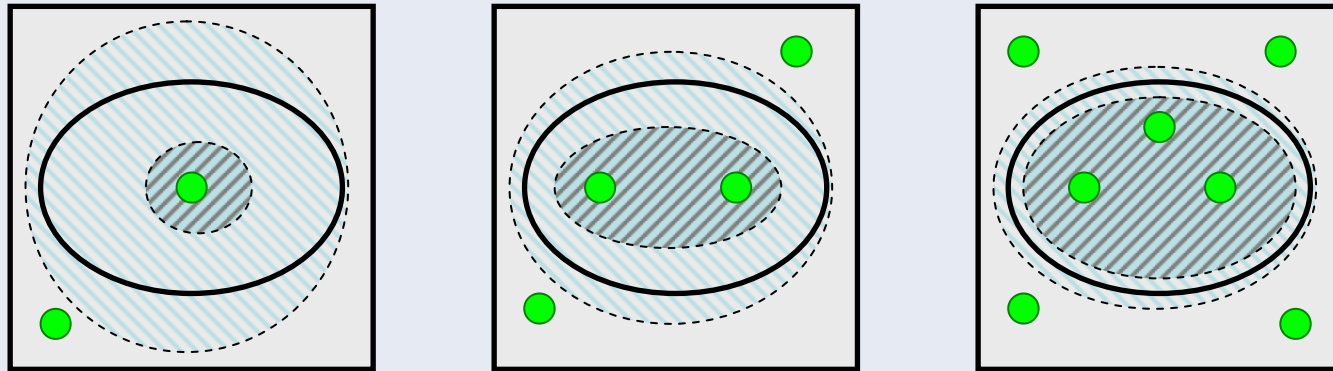
Unit Tests and Semantics



Model defined by kb

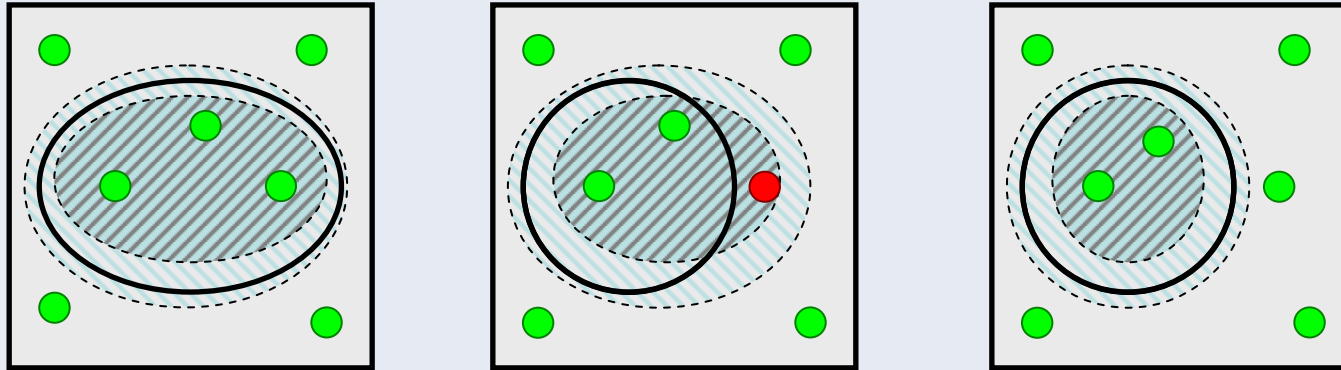
Running test cases proof that the model defined in the KB is within the fuzziness zone.

Specifying the intended Model with test cases



- Adding test cases reduces the fuzziness and improves the approximation of the intended model.

Changing the Model



- Radical changes are unlikely.
- Usually only a small part of the rule base changes.
- Most test cases can be re-used.

Unit Tests and Semantics (ctd)

- Declarative programming paradigm: the intended model is specified by entering rules.
- Tests specify the intended model as well, and executing the tests validates the consistency between the models specified by the tests and specified by the rules. Difference: tests can only be used to verify the model boundaries, not to compute the model (by answering queries).
- Test driven model specification is useful (or even necessary) if
 - Rules are complex.
 - Rules are maintained by different people.
 - Systems are described as black boxes (the rules are not made explicit, e.g. in a document describing the business rules within an organisation).

What can be tested

- Mainly query results, but also other properties (of the model) such as:
 - Number of different results in a query.
 - Derivations (rules supporting a particular result).
 - Order of results.
 - Profiling.
- Some of these properties are computational properties and not related to the semantics of the knowledge base.

Integrating knowledge bases and objects

- Integration between static and dynamic rule layer
 - S-Predicates
 - Methods as functions
 - In rule processing, methods can be invoked using reflection
 - Finding facts
- Integration between client application and rule module
 - Rule management and rule processing API
 - Reflection: representing rules, knowledge bases, terms, functions and predicates as objects

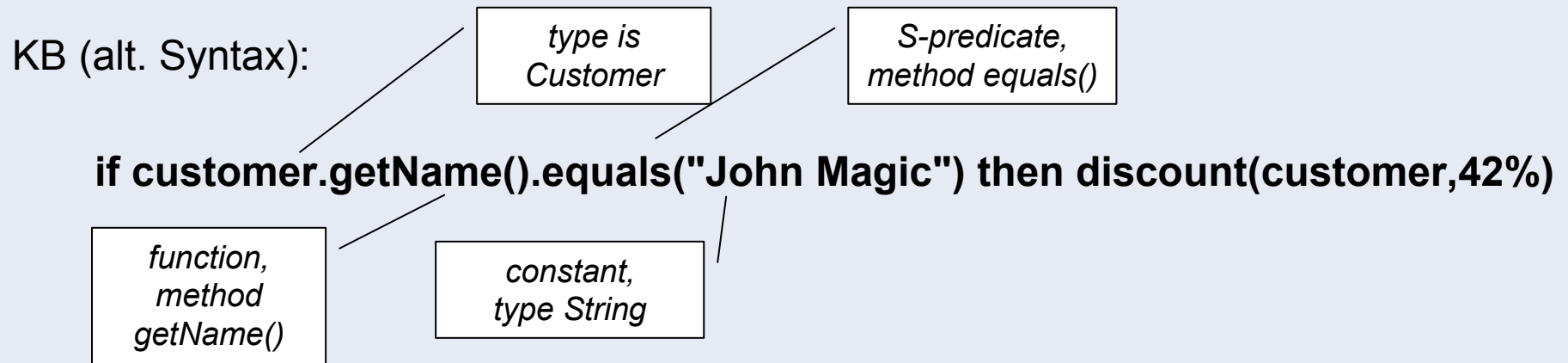
Integrating the SRL: Using Reflection

- Reflection: OO concepts (classes, methods,..) are exposed as objects at runtime. In particular, methods can be invoked dynamically (e.g., Java *invoke*, Smalltalk *perform*, C# *Invoke*).
- Reflection is used for a smooth SRL integration.
- Before a goal is proved by matching rules from the knowledge base, terms are simplified (using reflection for functions).
- Goals which are ground, have a S-Predicate and the invocation with the method associated with this predicate yields true, are removed from the agenda.

Example

KB: if the name of *a customer* is John Magic then give *a customer* a discount of 42%)

QUERY: discount(aCustomer[John Magic],x)



Proof – nodes represented as pairs (agenda, replacements)

1. {discount(aCustomer[John Magic],x)} // query
2. {aCustomer[John Magic].getName().equals("John Magic") }, x/42% // apply rule
3. {"John Magic". equals("John Magic") }, x/42% // simplify first term
4. {}, x/42% // performing S-predicate yields true, remove from agenda
5. SUCCESS !

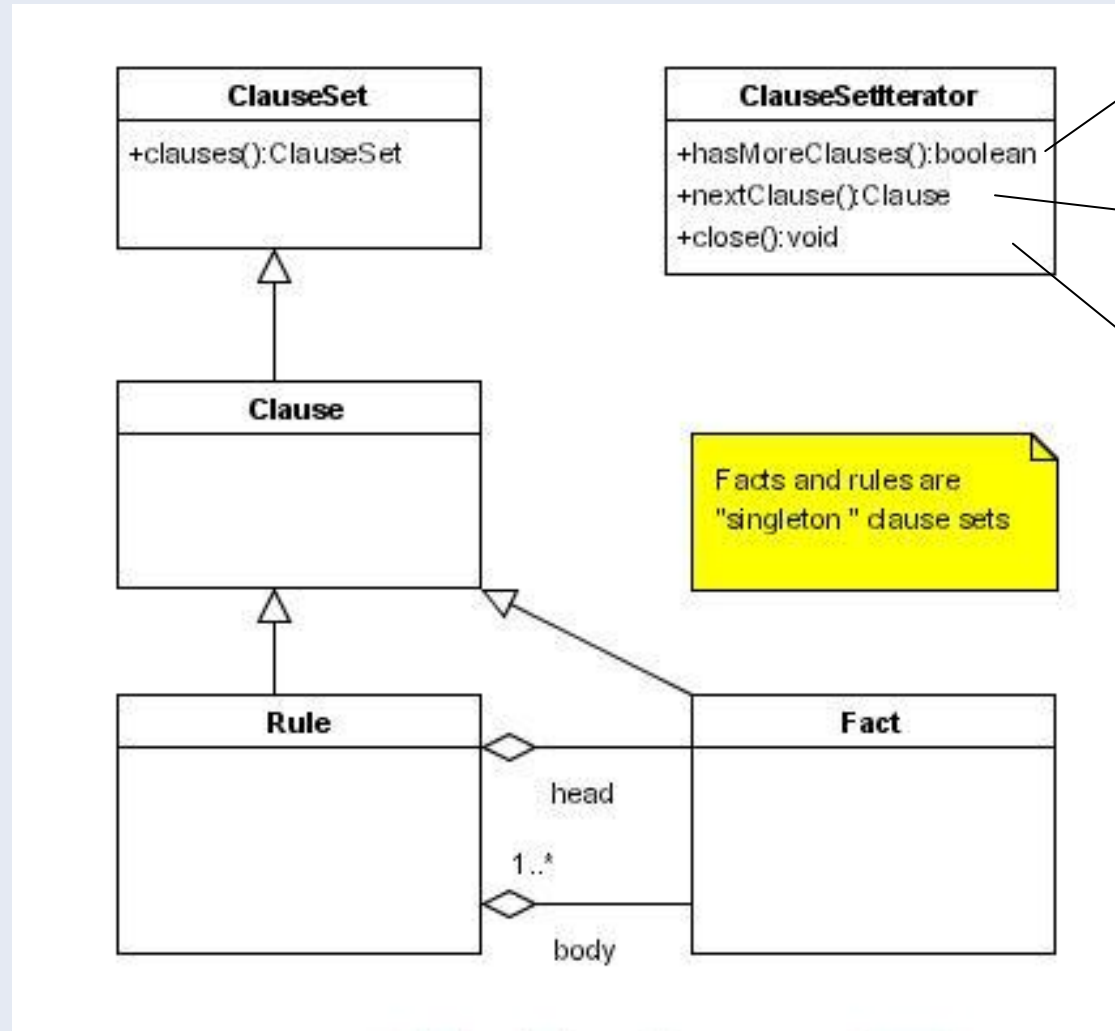
Facts

- Facts are persistent, stored using special software components (databases, with support for querying, indexing, transactions).
- Fact sets are often large (customer data, transaction data, ..).
- In general, this makes full replication in memory impossible. But facts must be accessible when rules are evaluated (at “query time”).
- Solution: design a data structure to access facts (or more general, clauses) on demand.

Clause Sets

- Uses modified version of iterator pattern from the GangOf4 pattern set [Gamma, Helm, Johnson, Vlissides 95].
- Modified as in Java Collection Library: only forward navigation.
- Clause sets are references to sets of clauses, and provide a clause set iterator to access those clauses. Building the clauses can be delayed.
- Iterator based access to data: RDBMS cursors / result sets, XML Pull parsers, RDF Jena Interface (Statement Iterators).

Clause Sets – UML



Whether there are more clauses.

Access the next clause.

Close the iterator to release (external) resources if the iterator is not longer needed.

Clause Sets – some Examples

- SQL Clause Sets – built from SQL result sets (JDBC).
- XML Clause Sets – built from XML documents accessed with PULL parsers.
- RDF Clause Sets – built from RDF sources accessed with iterator based APIs such as Jena.
- All clause sets need an adapter that converts the objects returned by the underlying iterators to facts.
- Rules with OR in the body – iterator splits rule into simple rules without disjunction.

Processing Rules

- Integration of knowledge base in application programs, "execution" of rules.
- Two primary execution models: backward and forward reasoning.
- Backward reasoning – queries are issued to "pull" information.
- Forward reasoning – rules are used to produce objects.

The Case for Backward Reasoning

- Most applications use a pull model. In particular, the following query technologies are frequently used:
 - (synchronous) method calls
 - Data base queries (in particular SQL, but also query languages for OODBMS, RDF, XML).
 - HTTP GET and POST requests.
- Integration of SRL is query based (see [example](#)).
- No need to propagate changes in fact base, no need to replicate fact based in memory. [TODO inset remark on Jess + memory]

Designing a query API

- An inference engine component is in charge.
- Iterator based access to query results. This enables the IE to perform work on demand (while transparent pre-reading is still possible).
- An object presenting the derivation should be accessible. This object reveals the applied rules.
- Query and knowledge base must be passed.

Exception Handling

- As support for distributed knowledge (in particular "remote" facts referenced as clause set) requires two features:
 1. A clean interface to release external resources (closing clause sets).
 2. A method to deal with errors (e.g. resulting from network failure).
- Exception handling is a strategy to deal with errors.
- A derivation algorithm can implement several exception handling strategies, including the two extremes:
 1. BUBBLE – each exceptions encountered when accessing a result set forces the derivation to fail (named after "event bubbling", a design pattern in GUI libraries).
 2. IGNORE – the exceptions are logged, and the next clause set or clause is used.

Processing Queries

- Processing a query means finding substitutions for the variable terms in the queries.
- Several algorithm can be implemented by an inference engine, including the SLD / SLDNF [TODO] resolution or alternative algorithms with improved results for knowledge bases with negation.
- This algorithm can be slightly modified to take S-Literals, exception handling and clause sets into account.
- Resolving S-literals can be seen as adding the respective ground literals to the kb. Note that the negation used in S-literals is not negation as failure.
- An alternative to interpreter based rule engine are compiled rule engines, e.g. reflection calls could be replaced by direct (generated) calls to the respective method. This leads to problems when maintaining / deploying knowledge base. In languages like Java, class loaders can be used to achieve this.

Example: Modified SLD Resolution

INPUT: A goal G and a knowledge base KB

OUTPUT: An instance of **goal** or **NO** or **ERROR**

ALGORITHM: Initialize **agenda** := **goal**

while (**agenda** is not empty) do

- simplify terms (perform methods associated to functions)*
- if exception output **ERROR***
- evaluate S-literals in **agenda**, if true remove from **agenda***
- if exception output **ERROR***
- choose goal A from the agenda
- choose clause set **CS***
- while (**CS** has more clauses) do*
- choose next clause $B_1..B_n \rightarrow A'$ from **CS** such that
 - A and A' unify with **mgu**
 - if no such goal exists, exit loop
- if exception output **ERROR***
- replace A by B_1, \dots, B_n in agenda
- apply **mgu** to **goal** and **agenda**

if (**agenda** is empty) output **goal**

else output **NO**

Processing Query Results

- Query result sets contain in general various results (different replacements).
- The order does matter and reflects the order of clause sets within the knowledge base.
- This order can be set either directly, or in a descriptive manner by using **comparators** (e.g., to prefer ground facts over rules containing variables).
- Post-processing of result sets is an alternative to handle the complexity of knowledge bases. This can be achieved with result set filters. Filters can be used to select, sort and aggregate results – similar to the use of WHERE, GROUP BY and ORDER BY clauses in SQL (backed by the relational model).

Processing Query Results (ctd)

- The iterator based access to result sets means in particular that result sets itself can be used as clause sets. This is the base for creating distributed knowledge bases.
- Besides result set filters, result sets can be transformed to support other query protocols, e.g. SQL query result sets, RDF sets etc.

Serializing Rules

- Serialization of rules can be used to make them persistent and invoked them at runtime.
- Various persistency mechanism including RDBMS based , binary serialization, XML, save as classes.
- Advantages of XML: flexible data model supporting deep polymorphic structures (e.g. as opposed to relational databases).
- Platform independent rule representation.
- Tools and services available (e.g. validation against constraint models, parsers, style sheet transformations).
- Fits well into the service/container system architecture – services are configured using XML files.

Representing objects

- Representing objects as structures within rules:
 - the internal state of the object is accessible (using properties / instance variables).
 - the object is made persistent by saving property name - property value associations.
 - The type (class) is saved either separately or as (meta) property

Problems representing Objects

- When reading objects, real-world classes must be instantiated. The representing objects as values approach implies that a generic instantiation mechanism (like java's `Class.newInstance`) must be used.
- This contradicts common design patterns aiming to improve encapsulation and modularity. In particular, factories are used and public types and implementation types are different.
- A major problem is the object id. Unless special meta properties for object ids are used, it is impossible to distinguish between persistent identical and equal objects.

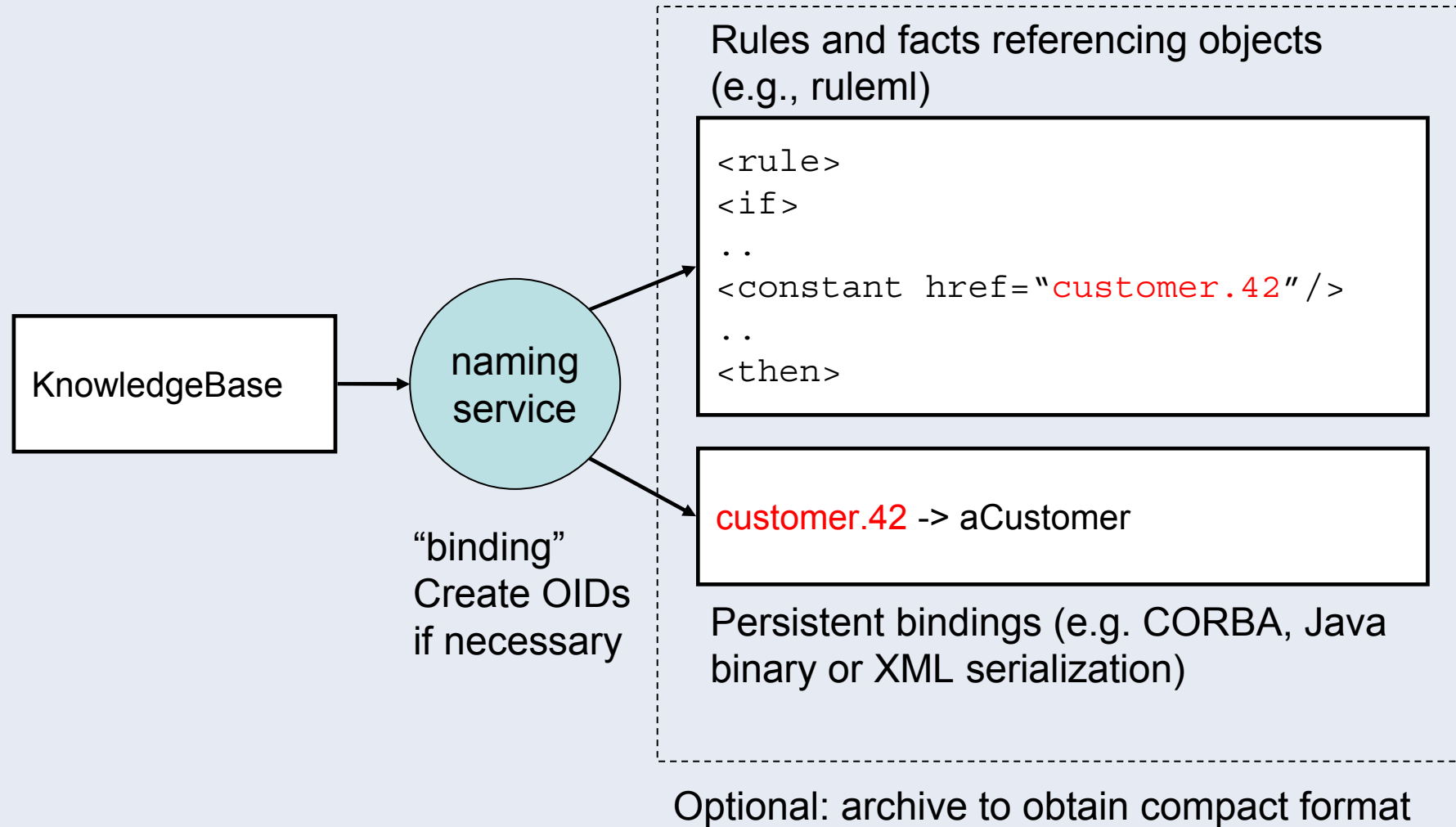
Problems representing Objects (ctd)

- Classes may change, and classes and persistent objects tend to become incompatible. This requires a versioning mechanism.
- These problems are related to the problems which occur when objects are mapped to relational databases.

Physical Object Ids

- A possible solution is simple: delegate the problem and store only object ids (oids). This relates to the “separation of concerns” design principle.
- Physical object ids (pointer addresses) are usually not accessible, object id here means a logical id generated by the application.
- One approach is to use a naming services (e.g., CORBA): objects can be associated with a name (= the oid, “binding”), and objects can be found by this name (“lookup”).
- In RuleML, the href attribute defined in various elements (constants, predicates, functions) can be used to store oids.

Serializing Rules



Deserializing Rules

Rules and facts referencing objects
(e.g., ruleml)

```
<rule>  
<if>  
..  
<constant href="customer.42" />  
..  
<then>
```

customer.42 -> aCustomer

Persistent bindings (e.g. CORBA, Java
binary or XML serialization)

