RBSLA: Rule Based Service Level Agreement

23.2.2006

IBIS, TU München
+49 89 289 17534
http://ibis.in.tum.de

RBSLA: Rule Based Service Level Agreement

Agenda

- State-of-Art: Service Level Agreement
- ContractLog
- Declarative RBSLA based on RuleML
- Key Findings

Adrian Paschke (TUM)

AORML Workshop Cottbus, 2006

Service Level Agreement (SLA)

- *An SLA is a document that describes the performance criteria a provider promises to meet while delivering a service.*
- *It typically also sets out the rights and obligations each person has in a particular context or situation, the remedial actions to be taken and any penalties that will take effect if the performance falls below the promised standard.*

→ *SLAs are an essential component of the legal contract between a service consumer and the provider.*

State of Art in SLA Management / Monitoring

- Natural language SLAs
  - Large amounts of contracts
  - Distributed environment
  - Different systems and people (roles) are involved
  - Goal:Dynamic SOA environment (utility/on-demand computing)
- Formal representation languages, e.g. XML based WSLA, SLAng:
  - Need interpreter
    - Conventional imperative programming languages, e.g. Java
  - Limited to simple Boolean logic to represent contract rules
  - No variables, no complex terms, no quantifiers, no rule chaining
- Commercial monitoring tools mainly focus on IT systems/resources
  - Contract/Business logic is buried in the code or database tiers
  - Contract rules (logic) are adjusted by parameters
  - Control flow must be completely implemented
  - Missing link between technical view and SLA view

Consequences

- Simplified, static contract rules with draconian penalties
- Simple SLA metrics: Availability
- No continuous monitoring
- No sepreation of concerns in tools
- Many manual processes

→ Difficulties in writing, maintaining and updating SLAs and contract rules

→ Difficulties in managing, monitoring and executing SLAs

→ **SLA Representation needs new levels of Agility, Flexibility and Automation**

→ **Needs expressive KR which keeps computational feasible**

Challenges – Example Rules

- Dependent Rules

  "*If the **average availability** falls below 98% then the **mean time to repair** must be less than 10 min.*"

- Graduated Rules

  - Monitoring Schedules

Schedule	Time	Availability	Response Time
Prime	8 -18	99%	4 sec.
Standard	18-8	95%	10 sec.
Maintenance	0-4 *	30%	-

  - Escalation Levels with Role Model

Level	Role	Time-to-Repair	Rights / Obligations
1	Process Manager	10 Min.	Start / Stop Service
2	Chief Quality Manager	Max. Time-to-Repair	Change Service Levels
3	Control Committee	-	All rights

Challenges – Example Rules

- Dynamic Rules

  "*There might be an **unscheduled period of time** which will be **triggered by the customer**. During this period **bandwidth must be doubled**.*"

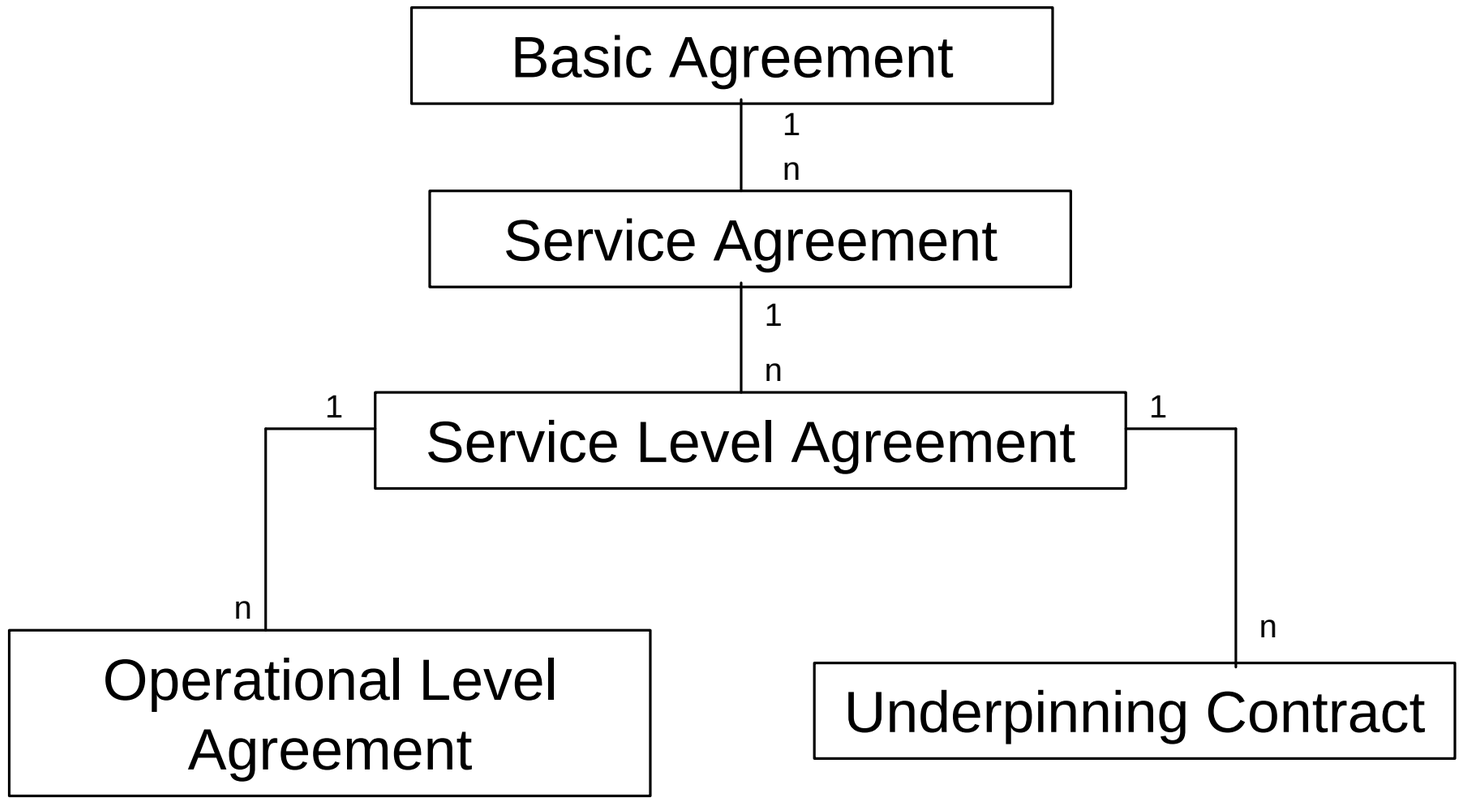
- Normative Rules with Violations and Exceptions

  "*The provider is **obliged** to repair an unavailable service in $t_{time\text{-}to\text{-}repair}$. If she fails to do so (**violation**) the customer is **permitted** to cancel the contract.*"

Further Challenges

- Distributed Environment
  - Distributed contract management
  - Need-to-know principle
  - Communication / Messaging needs (Events, Contract rules, Measurement Data etc.)
- Existing systems and tools
  - System-/network management tools
  - Databases / Data Warehouses
  - Business object implementations, e.g. EJBs, Web Services
  - Domain vocabularies, e.g. Semantic Web ontologies (RDFS/OWL)
  - Process management tools
  - Etc.
- Contract Life Cycle
  - Verification and Validation of SLA specifications
  - Negotiation
  - Automated monitoring and enforcement
  - Reporting, Metering+Accounting, Analysis
  - Refinement (Updates)

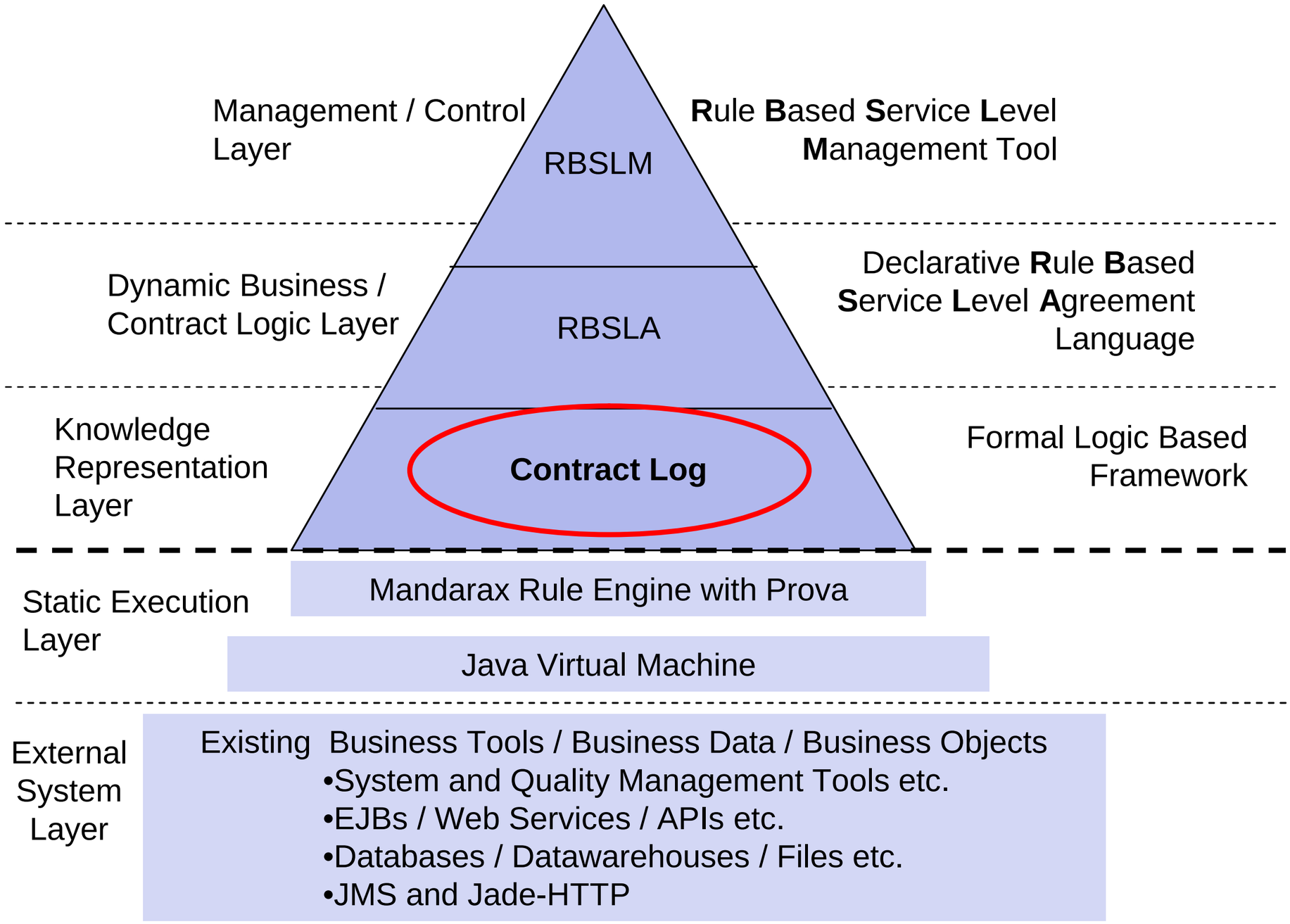
Further Challenges: Unitized, Modular Contract Structure



Possible Representations of SLA Rules

- If-then-else rules in imperative languages, e.g. Java, C++
- Decision Tables/Trees
- Triggers and Effectors (e.g. SQL triggers)
- State Machines
- Implications with constraints (e.g. OCL)
- …
- Logic Programming
  - Declarative programming
  - Compact representation
  - Expressive
  - Nice complexity results (under common restrictions)
  - Theoretically well-understood and uncontroversial for several logic types
  - Widely implemented and deployed, e.g. Prolog, deductive databases, rule engines.

RBSLA: Rule Based Service Level Agreement



ContractLog

ContractLog

Logic	Usage
General Logic Programs + Extensions	Derivation Rules (backward-reasoning) + Negation as Failure, Procedural Attachments, External Data Integration, Typed Logic
Event-Condition-Action rules (ECA)	Active behaviour (events, actions) + Update primitives for Active Rules
Event Calculus / Interval based Event Calculus	Temporal reasoning over effects of events on fluents (contract state tracking) / Complex Event Processing based on Interval based Event Algebra
Defeasible logic	Conflict resolution, default rules and priority relations of rules.
Deontic logic	Rights and obligations with violations an exceptions of norms.
Description logic / Hybrid Typed Description Logic Programs	Contract vocabularies, domain-specific concepts (term typing with typed unification)

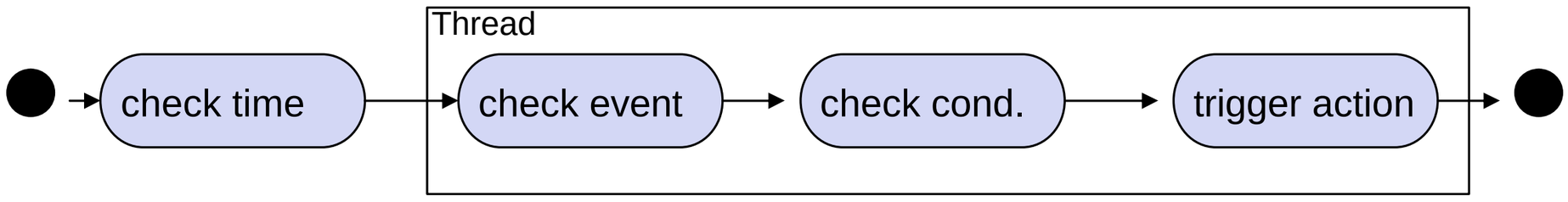
Event Condition Action – Basic Approach

- ECA Rules: eca(time,**event,condition,action**,post-conditon,else)
  - Query-driven Backward Reasoning ➔ simulate active forward driven operational semantics
  - Basic idea: Use “daemon” to frequently query and execute ECA rules
  - Note: Supports both Push and Pull models !!!!!

ECA rule*:* *eca (everyMinute, pingService,notMaintenance notify)*

Referenced Derivation Rules: (queried by daemon process)

(Time):	*everyMinute()*← …
Event:	*pingService()*← …
Condition:	*notMaintenance()* ← …
Action:	*notify()* ← …
(Post-Condition/Else)	…



Parts of the ECA rules

- Optional **time part**, might be used to specify:
  - Validity period of rule, e.g. "*from 1-1-2005 to 10-3-2006*"
  - Monitoring schedules, e.g. " *from 9 a.m. to 12 p.m. every 10 seconds"*
  - Time events, e.g. "at 25-2-2006 at 9:00 a.m."
  - Time and event part might be seen as a complex event consisting of a time event and an "normal" event
  - Reason for explicit time part: most SLAs define (adaptable) monitoring schedules for the reason of scalability and cost optimization

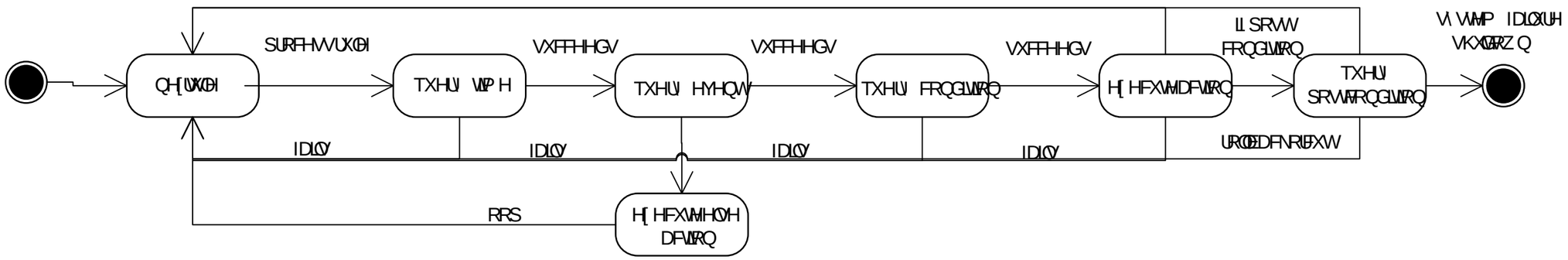
- **PostCondition**
  - Cuts and Counters might be set to prevent backtracking of variable bindings
  - PostCondition test, e.g. test integrity constraints, test special predefined test case etc.
  - If PostCondition test fails, then rollback (internal) update transactions

- **Else**
  - Defines alternative action in case the event fails.
  - Leads to a more compact syntax, e.g.

    ECA1: "*If service ping succeeds then do logging.*"

    ECA2: "*If service ping fails then send notification and do logging.*"

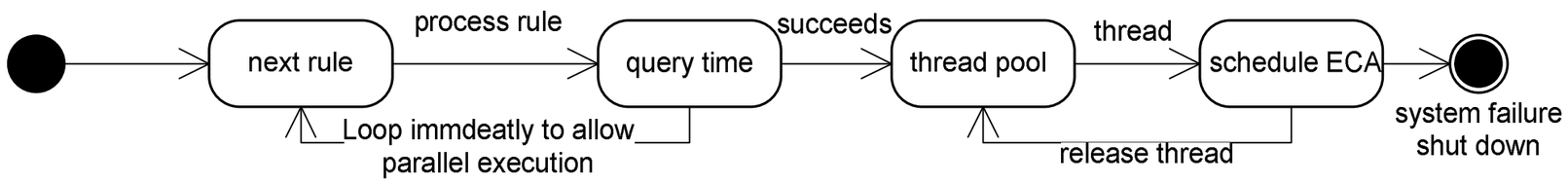
    becomes ECA 1/2: "*If service ping fails send notification and do logging **else** do only logging.*"



Multi-Threading

- Sequential ECA rule execution – Problem:
  - Active event detection (pull models) in external systems might take time, e.g. waiting for Internet answers
  - Computing complex events via event algebra might take time
  - Time-based complex events
  - Complex condition derivations
  - Complex action
  - Communication events/actions relating to communication protocols
  - Etc.

- Solution “Multi-Threading”:
  - Define monitoring / evaluation schedules for each ECA rule
  - Execute time consuming ECA parts in separate thread to enable parallel rule execution



- Only the simple time part needs to be evaluated
- If it fails nothing else has to be done
- If it succeeds, a thread is fetched from thread pool in which the rest of the rule is executed
- ECA interpreter in parallel proceeds with next ECA rule

→Higher scalability

→Push models can be implemented, i.e. ECA rules waiting for incoming events, e.g., incoming event messages

Properties of the ECA interpreter (1)

- Syntax: optional parts, i.e. might be empty, e.g.
  - eca(_, event,condition,action,_,_) reduces to eca(event,condition,action)
  - eca(_,event,_,action,_,_) reduces to eca(event,action)
  - eca(_,_,condition,action,_,_) reduces to eca(condition action)
- Variable bindings
  - eca(event(Service), _,action(Service))
  - Backtracking
  - Example:

    ECA rule: eca(time(),unavailable(**Service**),_,sendNotification(“administrator”), Service).
    Derivation rule: unavailable(**Service**):-service(**Service**),rbsla.utils.WebService.ping(**Service**).
    Facts: service(“**http://www.google.de**”). service(“**http://www.yahoo.com**”).
  - Cuts “!” and number quantifications can be set in post condition (or in associated derivation rules) to prevent backtracking
  - Example:

    ECA rule: eca(time(),event(),getFlight(**Flight**), bookFlight(**Flight**), **!**).
- Supports negation (not and neg)
  - not: not derivable (negation as failure)
  - neg: explicitly negated (strong negation)
  - eca(_,event, not(condition), …)
  - eca(_,neg(available),_,action)
  - Note: variables are not allowed in negated terms

Properties of the ECA interpreter (2)

- Typed Logic / Hybrid Typed Unification
  - Variables might be either un-type or typed.
  - Type is fully qualified class name, e.g. “java.lang.Integer.*X*”, “rbsla.utils.WebService.*S*”
  - During Unification the following rules apply:

    Untyped-Typed Unification:

    - The un-typed query variable assumes the type of the typed target variable or constant (individual)

    Variable-Variable Unification:

    - If the query and the target variable are not assignable, the unification fails otherwise it succeeds
    - If the query variable belongs to a subclass of the class of the target variable, the query variable assumes the type of the target variable.
    - If the query variable belongs to a super-class of the class of the target variable or is of the same class, the query variable retains its class

    Variable-Constant Unification:

    - If a variable is unified with a constant (individual) of its super-class, the unification fails otherwise 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 Unification:

    - The type of term from the head of the fact or rule is the same as or inherits from the type of term from the body of the rule or query
  - Examples:

    p(1). p(“1”). p(q(1)). p(java.lang.Date.X):- X=java.lang.Date(1790000234).

    p(X)? ➔ X=1, X=“1”, X=java.lang.Date.DateObjectID

    p(java.lang.Integer.X)? ➔ X = 1

    p(java.lang.Number.X)? ➔ X=1

➔ Restricts the search space to clauses where the type restrictions including subclasses are fulfilled

Properties of the ECA interpreter (3)

- Typed Logic (2)
  - Types supported in ECA interpreter, e.g.:
    eca(time(java.lang.Date.SysTime), event(java.lang.Date.SysTime, EventVariable), _ , action(EventVariable)).
  - Enables different rule variants, e.g.:
    add(java.lang.String.X , java.lang.String.Y, Result) :-
    Result = X.concat(Y).
    add(java.lang.Integer.X , java.lang.Integer.Y, Result) :-
    Result = X+Y.
  - Modes: Input/Output specifications (+,-,?)
    - add(+,+,-) ➔ add(X,Y,R):-bound(X),bound(Y),free(R), …
    - Further restricts search space and enables static mode checking (type checking)
  - Events might be simple Strings, complex terms, Java objects, RDFS/OWL classes etc.
    - Event types (interfaces and classes) with a hierarchy can be implemented in Java
    - Also possible to use Semantic Web Taxonomies to describe event types and event taxonomies.

- Procedural Attachments
  - Calls on external methods and functions
  - Static and instantiated methods from objects can be called with parameters (constants, variables)
  - Example:
    q(java.lang.Integer.Y) :- p(java.lang.String.X), Y=java.lang.Integer.parseInt(X).
    p(java.lang.String.X):- r(Z), X=Z.toString().
    r(1).
  - ➔ Objects can be bound to variables and their methods can be used during resolution in different rules.
  - Objects can be constructed via calling the constructors dynamically at runtime and bound to variables, e.g.
    X = rbsla.WebService(ServiceWSDL)

- Boolean valued procedural attachments directly supported in ECA rules, e.g.
  eca(everyMinute(), **rbsla.utils.WebService.ping(service)**, …)
  eca(time(), event(), condition(), **rbsla.utils.Reflec.triggerAction(action)**).
  eca(time, event(X), condition, **X.call()**).
  - Directly instantiated by ECA rule interpreter via Java reflection.
  - Bound objects and their properties/methods can be reused
  - ECA rules without using derivation rule engine possible, e.g.:
    eca(rblsa.utils.Time.resolve(10,0,0), rbsla.utils.Event.listen(“www.rbsla.de”), jdbc.SQL.count(…), rbsla.Action.fire(“notify”).

Properties of the ECA interpreter (4)

Based on Procedural Attachments support for messaging based on JMS or Jade-HTTP

- (provided by Prova AA - integration planned with Alex Kozlenkov)
  - *sendMsg(XID,Protocol,Agent,Performative,[Predicate|Args]|Context)*
  - *rcvMsg*
- Communicate events in distributed environment

- Type variables and bind simple datatypes and complex object structures to this variables which can be reused within the further resolution process
- Reuse procedural and highly-optimized functionalities in external implementations, e.g. mathematical computation, filters, aggregations etc.
  - Reuse existing tools, query engines, DBMS, EJBs etc.
- Build un-typed rule sets, which can be typed dynamically at runtime, e.g. un-typed list structures to be populated with different content types, event algebra, e.g.
  - Event Calculus: holdsAt(Fluent,Time), happens(Event,Time) etc.

    Event, Time etc. might be everything, e.g. an event string “event1”, a complex term “time(y,m,d,h,m,s)”, an Java object etc.
- ECA Interpreter provides wrapper interface to derivation rule engines (currently Prova).
  - Other engines which provide a query interface can be integrated.

Properties of the ECA interpreter (5)

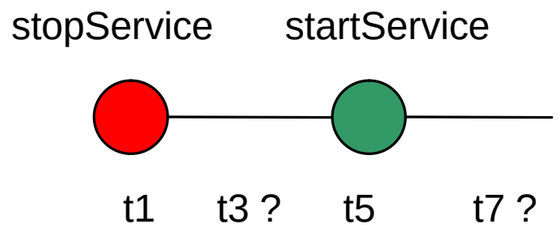
- Complex Events / Actions
  - In simplest case represented via logical connective, e.g.:
    - cplxEvent():- event1(), event2(), not(event3()) % un-ordered conjunction
    - cplxEvent(): event1() ` event2() ➔ cplxEvent() :- event1(). cplxEvent():- event2(). % disjunction
    - cplxEvent():- event1(), event2(), not(event3()), t1<t2<t3 % ordered conjunction
  - Can be combined with logical constructs, e.g. lists and procedural attachments to represent expressive filters, event aggregations, combinations etc.
  - More complex events such as time based events need an event algebra to compute the events / actions
  - Event algebras implemented in terms of interval based Event Calculus
    - Paschke, A. : ECA-RuleML: An Approach combining ECA Rules with interval-based Event Logics and Event Notifications, IBIS, Technische Universität München, Technical Report 11 / 2005.
  - We use the Event Calculus, but other algebras are possible and can be simply added (dynamically at runtime) via updating or exchanging the EC event algebra rule set.
- External effects and integration of external Systems via Procedural Attachments
- Prova AA agent communication for agent oriented conversations
- Internal Updates
  - Arbitrary knowledge updates with ID
    - update(id1,"p(X):-q(X).") % add rule p(X):-q(X) to KB with ID "id1"
    - update(id2,"./rules/new.prova") % add the complete prova script with id2
    - remove(id2) % remove the complete knowledge with id2
    - ➔ Modules with ID can be updated, locally reasoned and removed
  - The ID can be used to build modules (rule sets), prioritize rules and modules (defeasible logic), distinguish local knowledge and general knowledge, integrate external knowledge from files, database etc., selectively update and remove knowledege including bulk and cascading updateds.
  - Transactional Updates with Integrity constraints and test cases
    - transaction(update(…)) update knowledge and then test all test cases and integrity constraints in the KB If test fails then rollback update.
    - transaction(update(..),"transactionTestCase.prova") test update with specified test case and rollback if test fails

Event Calculus as Event Algebra

- **Effects of Events/Actions on Fluents**
  - Rules for state transitions / derive actual contract state ~ Context
  - Contract State Tracking
  - Time-based / Context based complex events
- ***EC Basic Axioms:***
  - *happens(E,T)* — *event E happens at time point T*
  - *initiates(E,F,T)* — *event E initiates fluent F for all time>T*
  - *terminates(E,F,T)* — *event E terminates fluent F for all time>T*
  - *holdsAt(F,T)* — *fluent F holds at time point T*
- ***EC Extensions:***
  - *valueAt(P,T,X)* — *parameter P has change-able value X at time point T*
  - *planned(E,T)* — *event E is believed to happen at time point T*
- ***Derived Events, Delayed Events, Concurrent Events***
  - ***Can be implemented in Event Calculus***
  - *Note: Terms are untyped !!!!!*

Example:

initiates*(stopService,serviceUnavailable,T)*
terminates*(startService,serviceUnavailable,T)*
happens*(stopService,t1);* ***happens****(startService,t5)*
holdsAt*(serviceUnavailable,t3)?* → ***true***
holdsAt*(serviceUnavailable,t7)?* → ***false***



Properties of the Event Calculus

- Computes current state(s) from a sequence of happened events
  - akin to state machines, e.g:
    happens(e2,T) :- holdsAt(f1,T), happens(e1,T).
    initiates(e2,f2,T).
- Completely rule based in terms of derivation rules, i.e. easily extensible, e.g.

  happens(E,T):- database(DB,TABLE), sql_select(DB,TABLE,[event, E], [timepoint, T])

  initiates(E,F,T):- testIntegrity(F), … .

  happens(CplxE,T) :- happens(e1,T), happens(e2,T).
- Completely un-typed: Event, Fluent, Time can be everything
- Supports complex events such as concurrent events, delayed events (t2+X>t1) , derived fluents, countdowns, counter etc.
  - Example: holdsAt(conflict,T):- holdsAt(permit(S,O,A),T), holdsAt(forbid(S,O,A),T).
- Can be combined with other logical formalisms, e.g. deontic logic, defeasible logic other event algebras etc. or replaced.
- Useful to reason over state changes and give explanations (derivation trees) and do computations (compute penalties, average values etc.).
  - Traceability and verifiability are important in SLA domain
- The ECA Interpreter distinguishes non-transient (complex) events which are represented (atomic or complex events), selected according to their type (in case of complex events according to event algebra) and consumed trigger reactions in the ECA rule and transient events (persistent events) which are used in the Event Calculus.
  - Non-transient events are queued for complex event computation according to event algebra (using and, or, time, quantification etc.) and discarded after consumption
  - Persistent events are stored, e.g. in fact base or data base etc., and used in the event calculus.
  - Interval based Event Calculus: Event occur at a time inverval (t1;t2)
  - More information: Paschke, A.: ECA-RuleML: An Approach combining ECA Rules with interval-based Event Logics and Event Notifications, IBIS, Technische Universität München, Technical Report 11 / 2005.

Deontic Logic

- **Logic of normative concepts: oblige, permit, forbid**
  - Normative propositions with truth values
- **Standard Deontic Logic (SDL)**
  - OA , PA, FA …
  - Inference Rules
    OA → PA; FA → O¬A; PA → ¬FA …
    Consequential Closure: A → B / OA → OB
    Valid Scheme: ¬(OA ´ O¬A)
- **Shortcomings**
  - Norms are not personalized / object oriented
  - Norms are time-less
  - Effects of events/actions are ignored
  - Exceptions and Violations lead to Paradoxes, i.e. SDL is inconsistent.

Event Calculus based Deontic Logic

Norms are personalized, time-varying Fluents

$N_{S,O}A$ (N=norm, S=Subject, O=Object, A=Action)

They are relative to their "context"

- OA and O¬A might hold at the same time for different Subjects / Objects (or at different times)

Embedded in Event Calculus

- Initially(**oblige(S,O,A)**) *Norm holds initially*
- Initiates(E1,**oblige(S,O,A)**,T) *Event E1 initiates norm*
- Terminates(E2,**oblige(S,O,A)**,T) *Event E2 terminates norm*

Authorization Control

- Positive Authorization: "*By default everything is forbidden*" - holdsAt (norm,T)?

Deontic Rules with conditional norms

- A1 → OA2 : *happens(A1,T)* *initiates(A1,oblige(S,O,A2),T)*

Defeasible Obligations / Prima Facie Obligations (~ promised duties)

- Subject to **Exceptions:** *OA* and *E→ O¬A* (E=Exception Event)

terminates(E,oblige(S,O,A),T) initiates(E,oblige(S,O,notA),T)

Contrary to Duty Obligations (CTD)

- Subject to **Violations**: (¬A1 ´ OA1) → V → OA2 (V= Violation OA2=CTD)

holdsAt(oblige(S,O,A1),T), happens(notA1,T) → happens(V,T)
initiates(V, oblige(S,O,A2),T)

Event Calculus Deontic Logic Example

Example Violation and Contrary-to-Duty Obligation:

OA1
¬A1
¬A1 ´ OA1 → OA2

(Deontic Rule)

happens(**e1**, t).
initiates(**e1**, **obliged(s,o,a1)** ,T).

(Violation CDT)

happens(v,t) :-
holdsAt(**oblige(s,o,a1)**, T), happens(**neg(a1)**,T).
happens(**neg(a1)**,t1).
initiates(v, oblige(s,o,a2),T).
terminates(v, oblige(s,o,a1),T).

Example Exception:

OA
E→ O¬A

holdsAt(oblige(s,o,a),T).
terminates(e, oblige(s,o,a),T).
initiates(e,oblige(s,o,neg(a)),T).
happens(e,t5).

Integrity Constraints and Test Cases

- Integrity Constraints
  - Define a condition which must always hold
  - In ContractLog 4 types of integrity constraints are supported
    - **Not-constraints** which express that none of the stated conclusions should be drawn:
      integrity(not(p1(…), .. , pn(…))).
    - **Xor-constraints** which express that the stated conclusions are mutual exclusive, i.e. should not be drawn at the same time: *integrity(xor(p1(…),..,pn(…))).*
    - **Or-constraints** which express that at least one of the stated conclusions should be drawn:
      integrity(or(p1(…),..,pn(…))).
    - **And-constraints** which express that all of the stated conclusion should be drawn:
      integrity (and(p1(…), .. , pn(…))).
  - Integrity Constraints might be conditional, i.e. integrity rules: integrity(..) ← body.

- Test Cases
  - test input facts
  - meta test rules
  - queries,
  - expected results
  - ➔ a test case without input facts and test rules reduces to a integrity constraint

- Test predicates
  - testIntegrity() test all integrity constraints in the KB
  - testIntegrity(Literal) test all constraints as if literal is in KB (hypothetically)
    ➔ Useful for hypothetical update tests

Example Integrity Constraints / Test Cases

- Integrity Constraint Example:

```
integrity(xor(p(X),neg(p(X)).    % p ´ ¬ p are mutual exclusive
integrity(xor(permit(S,O,A), forbid(S,O,A)). % PA ´ FA are mutual exclusive
integrity(xor([P|Args],neg([P|Args])):-bound(P). % positive and negative are not allowed
integrity(and(q(X),p(X)). % Q ´ P must appear at the same time
integrity(not(a(),b())) :- c(). % in case c can be derived a and b are not allowed
```

- Test Cases

```
% ID
testcase(id). % ID need for adding/removin/local reasoning

% Test Facts
f(1). f(2).

% Meta Test rules

% Tests
testSuccess("test 1",""):-testcase(id), f(1),f(2). % name and message used in JUnit
testFailure("test 1","can not derive a and b"):- not(testSuccess("test1",Message)).
runTest(id):- testSuccess("test1").
```

Conflict Handling and Prioritization via Defeasible Logic

- Reasoning with incomplete or inconsistent information
  - Strict rules: body → head
  - Defeasible rules: body ==> head
  - Superiority relations: overrides(r1,r2)
  - Defeaters defeat knowledge
- Normal defeasible logic handles conflicts between positive and negative information
  Generalized Courteous Logic Programs handle mutex conflicts (comparable to XOR integrity)
- In ContractLog generalized defeasible theory
  - Based on integrity constraints and test cases to define conflicting knowldege, e.g.

```
integrity(xor(p(x), neg(p(x))).          % p(x) and neg(p(x)) are mutual exclusive
integrity(xor(p1(x),p2(x),p3(x))).       % p1(x), p2(x) and p3(x) are mutual exclusive
integrity(and(p(x),q(x))).               % p(x) and q(x) must be derivable at the same time
```

- Transformation into meta-program in LP
  1. Every strict knowledege is also defeasibly provable
     defeasible([P|Args]):-bound(P), derive([P|Args]).
  2. Each priority relation “r1>r2” where rl are the rule names (rule object identifiers) is stated as:
     overrides(r1,r2)
  3. Each defeasible rule r: p <= q1,..,qn is translated into the following set of clauses:
     oid(r,p).
     neg(blocked(defeasible(p))):- defeasible(qi), testIntegrity(p).
     defeasible(p) :-
     neg(blocked(defeasible(p))), testDefeasibleIntegrity(r, p).
  4. A meta program defeasibly tests the integrity of the rule, i.e. test whether there are conflicting rules, test if this conflicting rules are blocked or have lower priority than the tested rule.
- Priority relations over rules and modules (rule sets) can be defined via their ID:
  overrides(module1,module2) overrides(ruleID1,ruleID2)

Further Issues

- OWL2Prova and Description Logic Programs
  - Inference Rules implemented as Derivation Rules, e.g.:
    a:C, i.e., the individual *a* is an instance of the class *C*: **C(a)**
    C⊆ D, i.e., class C is subclass of D: **D(X) ← C(X)**
    C≡ D, i.e., class C is equivalent to class D: **D(X) ← C(X)**
    C(X) ← D(X)
  - Used for term typing based on class taxonomies: type(x,C) % is x of type Class C.

- Verification and Validation of SLA Specifications / Rule Sets
  - Via test cases defined as LPs
  - Tested via test predicates
  - Java Implementation to execute LP Test Cases and create JUnit report

- Communication aspects to support distributed environment
  - Synchronized / A-synchronized conversations via JADE or JMS protocols
  - Joint work with Alex Kozlenkov, combining ECA rules with Prova AA (agent communication architecture)

RBSLA

Rule Based Service Level Agreement Language

- Abstract declarative syntax → Simplify authoring/writing and **interchange** of SLAs
- **Based on RuleML**
- Goals:
  - Machine-Readability and execution in LP inference engine ~ rule engine (via Transformation)
  - Tool-Support
  - Interoperability with other (rule) languages and distributed rule systems
- RuleML
  - Standardization: Open, producer-independent, XML/RDF based web language for rules
  - Rule types:
    - Derivation rules (business rules), e.g. representation with LP
    - Reaction rules (production rules, ECA rules) (not specified yet)
    - Transformation rules, Integrity constraints
  - Currently: Derivation Rules
  - Unitized Structure: Modules for DataLog, HornLog (with Naf), Disjunctive LP, FOL (extended LPs)
  - Since version 0.85: Object-Oriented KR (User-Level Roles, URI-Grounded Clauses, Order Sorted Terms)
  - Not intended to be executed directly, but transformation (e.g. XSLT) into target language, e.g. Prolog.

RBSLA Language

- Main extensions to RuleML:
  - *Typed Logic and Procedural Attachments*
  - *External Data Integration*
  - *Event Condition Action Rules with Sensing, Monitoring and Effecting*
  - *(Situated) Update Primitives*
  - *Complex Event Processing and State Changes (Fluents)*
  - *Deontic Norms and Norm Violations and Exceptions*
  - *Defeasible Rules and Rule Priorities*
  - *Built-Ins, Aggregate and Compare Operators, Lists*
- **RBSLA v0.2 based on RuleML 0.9**
  - **Several changes, e.g. events, actions became role tag (can be omitted → compact syntax)**
- Layered structure (unitized in modules):
  - KR Layers: ContractLog layers: ECA, EC, Defeasible, Deontic, Typed LP, domain-specific …
  - Syntax Layers: (striped) RuleML, RBSLA, SLA-specific RBSLA, if-then syntax

RBSLA Interpretation / Execution

- Transformation to ContractLog KR
- Interpretation and execution in Prova/Mandarax rule engine (backward-reasoning)
- Type and Mode Checking
  - Static, e.g. *p(+)* ← *p(-)*
- Validation
  - Dynamic via test cases
- Defeasible Compiler (handling incomplete, contradicting knowledge)
  - Translation into LP meta program
- RDFS/OWL Inference Layer
  - RDFS/OWL typed rules in addition to Java
  - Hybrid approach: Combination of rules and description logic programs

Key Findings in a Nutshell

- Rules (contract logic) are separated from the application logic
  - Easier management and maintenance
  - Compact representation via rule chaining
- Logic based formalization
  - Automation and Execution in rule engine (+extension)
  - Verification and Validation
- Combine logic programming and OO Programming
- Reuse existing functionalities from external implementations and databases
- Complex Event Processing
- (Pro-)active Monitoring and Contract State Tracking
- Time and Event-based Rights and Obligations Management
- Automated conflict detection and resolution (e.g. rule prioritization)

Thank you for attention !!!!

Questions?

Backup

Example Generalized Defeasible Logic Program

```
% Facts
status("Adrian",gold).
status("Adrian",silver).
neg(payed("Adrian")).

% Defeasible rules
discount5: discount(X,5):= defeasible(status(X,silver)).
discount10: discount(X,10):= defeasible(status(X,gold)).
nodiscount: neg(discount(X)):=
                    defeasible(neg(payed("Adrian")).

% Integrity constraints
integrity(xor(discount(X,_),neg(discount(X))).
integrity(xor(discount(X,5),discount(X,10))).

% Priorities
overrides("nodiscount","discount10").
overrides("discount10","discount5").
```

Example Generalized Defeasible Logic Program (2)

```
% facts
status("Adrian","gold"). % strict fact
status("Adrian","silver"). % strict fact
neg(payed("Adrian")). % strict fact

% integrity constraints
integrity(xor(discount(X,_),neg(discount(X)))).
integrity(xor(discount(X,5),discount(X,10))).

% defeasible rules
% defeasible rule “discount5”
oid("discount5",discount(X,5)).
defeasible(discount(X,5)):-
    defeasible(status(X,"silver")),
    testIntegrity(discount(X,5)),
    testDefeasibleIntegrity("discount5",discount(X,5)).
neg(blocked(defeasible(discount(X,5)))):-defeasible(status(X,"silver")), testIntegrity(discount(X,5)).
```

Example Generalized Defeasible Logic Program (3)

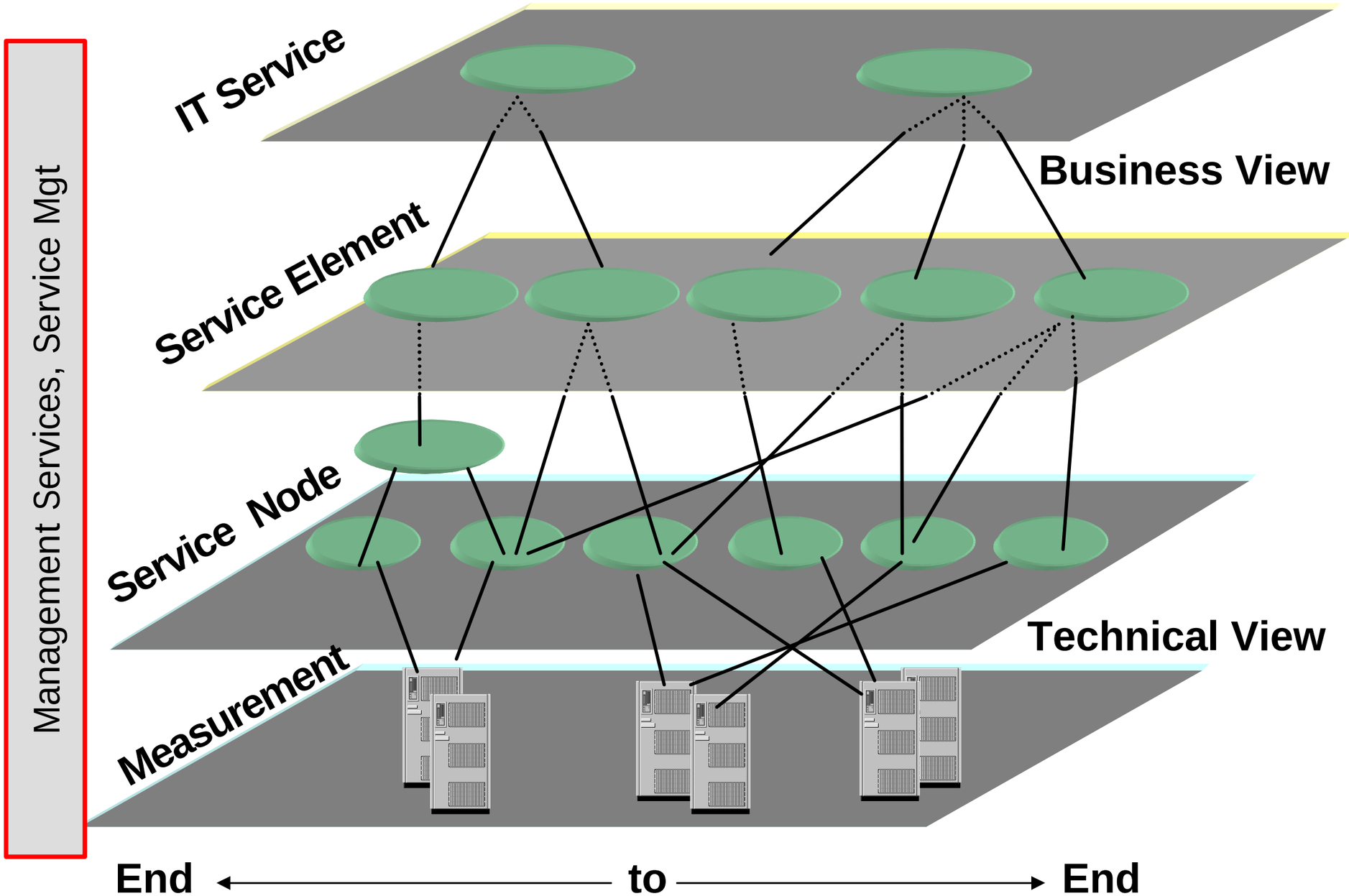
```
% defeasible rule “discount10”
oid("discount10",discount(X,10)).
defeasible(discount(X,10)):-
    defeasible(status(X,"gold")),
    testIntegrity(discount(X,10)),
    testDefeasibleIntegrity("discount10",discount(X,10)).
neg(blocked(defeasible(discount(X,10)))):-defeasible(status(X,"gold")), testIntegrity(discount(X,10)).

% defeasible rule “nodiscount”
oid("nodiscount",neg(discount(X))).
defeasible(neg(discount(X))):-
    defeasible(neg(payed(X))),
    testIntegrity(neg(discount(X))),
    testDefeasibleIntegrity("nodiscount",neg(discount(X))).
neg(blocked(defeasible(neg(discount(X))))):-defeasible(neg(payed(X))), testIntegrity(neg(discount(X))).

% priority definitions
overrides("nodiscount","discount10").
overrides("discount10","discount5").
```

Query: discount(X,D) ➔ fails / Query nodiscount(X) succeeds

IT Service Management



General Architecture