Rule responder : rule-based agents for the semantic-pragmatic web

Paschke, Adrian; Boley, Harold

Publisher's version / Version de l'éditeur:
https://doi.org/10.1142/S0218213011000528
International Journal on Artificial Intelligence Tools (IJAIT), 20, 6, pp. 1043-1081, 2011

NRC Publications Record / Notice d'Archives des publications de CNRC:
https://nrc-publications.canada.ca/eng/view/object/?id=6595b500-9df5-4d69-91f7-8c54095cb082
https://publications-cnrc.canada.ca/fra/voir/objet/?id=6595b500-9df5-4d69-91f7-8c54095cb082

International Journal on Artificial Intelligence Tools

Rule Responder: Rule-based Agents for the Semantic-Pragmatic Web

Adrian Paschke
AG Corporate Semantic, Department of Computer Science, Freie Universitaet Berlin, Germany
adrian.paschke AT inf.fu-berlin.de

Harold Boley
Institute for Information Technology, National Research Council Canada
Fredericton, NB, Canada
harold.boley AT nrc.gc.ca

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Rule Responder is a Pragmatic Web infrastructure for distributed rule-based event processing multi-agent eco-systems. This allows specifying virtual organizations – with their shared and individual (semantic and pragmatic) contexts, decisions, and actions/events for rule-based collaboration between the distributed members. The (semi-)autonomous agents use rule engines and Semantic Web rules to describe and execute derivation and reaction logic which declaratively implements the organizational semiotics and the different distributed system/agent topologies with their negotiation/coordination mechanisms. They employ ontologies in their knowledge bases to represent semantic domain vocabularies, normative pragmatics and pragmatic context of event-based conversations and actions.

Keywords: Pragmatic Web; Semantic Web; Multi Agent System.

1. Introduction

Rule Responder[a] extends the Semantic Web towards a Pragmatic Web infrastructure for collaborative rule-based agent networks realizing distributed rule inference services, where independent agents engage in conversations by exchanging event messages and cooperate to achieve (collaborative) goals.

Rule Responder agents communicate in conversations that allow implementing different agent coordination and negotiation protocols. By means of pragmatic primitives, such as speech acts, deontic norms, etc., which are represented as ontologies, Rule Responder attaches the semantic and pragmatic context, e.g. organizational norms, purposes or goals and values, to the interchanged messages. In its multi-agent architecture Rule Responder utilizes messaging reaction rules from Reaction

---

[a] *http://responder.ruleml.org*

RuleML[b] for communication between the distributed agent inference services. The Rule Responder middleware is based on modern enterprise service technologies and Semantic Web technologies for implementing intelligent agent services that access data and ontologies, receive and detect events (e.g., for complex event processing in event processing agent networks), and make rule-based inferences and (semi-)autonomous pro-active decisions for reactions based on these representations.

The core of a Rule Responder agent is a rule engine, such as Prova[c], OO jDREW, DR-Device (initially in Emerald), Euler, or Drools, which implements the decision and behavioral reaction logic of the agents' roles. An agent can employ vocabularies defined as Semantic Web ontologies (e.g., based on RDFS or OWL) to give its rules a domain-specific meaning. The vocabularies can be used within the conversation with other agents to enable a semantic and pragmatic interpretation of the messages. For the deployment of agents on the Web and for the communication in agent networks, Rule Responder uses an enterprise service bus middleware, which supports a multitude of synchronous and asynchronous transport protocols (>40) – such as SMTP, JDBC, TCP, HTTP, XMPP – to transport rulebases, queries and answers between the agents. The de facto standard Reaction RuleML is used as a platform-independent rule interchange format for agent conversation.

In summary, Rule Responder can be seen to support a *digital agent ecosystem*, evolving from the Semantic Web to the Pragmatic Web [d]. Such an ecosystem consists of all the semantic agents in one or more virtual organizations, as well as all the other components of this environment with which the agents interact, such as other services, tools, the ESB middleware, etc.

The rest of the article is organized as follows. Section 2 explains a typical distributed agent topology for virtual organizations and the types of agents used to implement it. Section 3 discusses the components and used technologies of the Rule Responder framework. Section 4 focuses on interchange between the semantic agents which communicate by using (Reaction) RuleML as common rule interchange format. Section 5 describe how Rule Responder agents are implemented using the expressive Semantic Web rule engine Prova. Section 6 demonstrates some application use cases of Rule Responder by means of selected Rule Responder instantiations. Section 7 discusses related work and section 8 concludes the paper.

2. Rule Responder Multi Agent Architecture

Rule Responder supports the implementation of various distributed agent coordination topologies, from centralized orchestration, executed in star-like agent nodes,

---

[b] *http://reaction.ruleml.org*

[c] *http://prova.ws*

[d] *http://www.pragmaticweb.info/*

to decentralized ad-hoc choreography within the Rule Responder agent network. In the following, we describe a common hierarchical agent topology which represents a centralized star-like structure for virtual organizations (and many orchestrated distributed systems). Organizational Agents (OAs) act as central orchestration nodes which control and disseminate the information flow from and to their internal Personal Agents (PAs), and the External Agents/Services (EAs) and internal Computational Agents/Services (CAs).

2.1. *Organizational Agent*

An Organizational Agent (OA) represents a virtual organization (respectively network of agents) as a whole. An OA manages its local Personal Agents (PAs), providing control of their life cycle and ensuring overall goals and policies of the organization and its semiotic structures. OAs can act as a single point of entry to the managed sets of local PAs to which requests by EAs are disseminated. This allows for efficient implementation of various mechanisms of making sure the PAs functionalities are not abused (security mechanisms) and making sure privacy of entities, personal data, and computation resources is respected (privacy & information hiding mechanisms). For instance, an OA can disclose information about the organization to authorized external parties without revealing private information and local data of the PAs, although this data might have been used in the PAs to compute the resulting answers to the external requester.

2.2. *Personal Agents*

Personal Agents (PAs) assist the local entities of a virtual organization (respective network). Often these are human roles in the organization. But, it might be also services or applications in, e.g. a service oriented architecture. A PA runs a rule engine which accesses different sources of local data and computes answers according to the local rule-based decision logic of the PA. Depending on the required expressiveness to represent the PAs rule logic arbitrary rule engines can be used as long as they provide an interface to ask queries and receive answers which are translated into the common Reaction RuleML interchange format in order to communicate with other agents.

Importantly, the PAs might have local autonomy and might support privacy and security implementations. In particular, local information used in the PA rules becomes only accessible by authorized access of the OA via the public interfaces of the PA which act as an abstraction layer supporting security and information hiding. A typical coordination protocol is that all communication to EAs is via the OA, but the OA might also reveal the direct contact address of a PA to authorized external agents which can then start an ad-hoc conversation directly with the PA [BP07]. A PA itself might act as a nested suborganization, i.e. containing itself an OA providing access to a suborganization within the main virtual organization. For instance, this can be useful to represent nested organizational structures such as departments, project teams, service networks, where e.g., the department chair is a personal agent within the organization and at the same time an organizational chair for the department, managing the personal agents of the department.

2.3. *Internal Computational Agents*

Computational Agents (CAs) act as wrappers around internal computational services and data which is used by the PAs. They fulfill computational tasks such as collecting, transforming and aggregating data from internal data sources or computational functions. The PAs can communicate with the CAs decoupled via interchanging event messages or loosely coupled via the built-ins and service interfaces of the rule engines in the PAs.

2.4. *External Agents*

External Agents (EAs) constitute the points-of-contact that allow an external user or service to query the Organizational Agent (OA) of a virtual organization. An EA is based, e.g., on a Web (HTTP) interface that allows such an enquiry user to pose queries, employing a menu-based Web form, which gets translated to equivalent RuleML/XML message that is sent to the OA via HTTP Post or Get. An external agent – from the point of view of a Rule Responder agent organization – can be an external human agent, a service/tool, or another external Rule Responder organization, i.e. leading to cross-organizational Rule Responder communications.

3. The Rule Responder Framework

Three interconnected architectural layers constitute the Rule Responder framework, listed here from top to bottom:

- Computationally independent user interfaces such as template-based Web forms or controlled English rule interfaces.
- Reaction RuleML as the common platform-independent rule interchange format to interchange rules, events, actions, queries, and data between Rule Responder agents and other agents (e.g., Semantic Web services or humans via Web forms).
- A highly scalable and efficient enterprise service bus (ESB) as agent/service-broker and communication middleware on which platform-specific rule engines are deployed as distributed agent nodes (respective semantic inference Web services). These engines manage and execute the logic of Rule Responder's semantic agents in terms of declarative rules which have access to semantic ontologies.

In the following, the Rule Responder framework will be explained from bottom to top.

3.1. *Enterprise Service Bus Communication Middleware*

To seamlessly handle message-based interactions between the Rule Responder agents/services and other agents/services using disparate complex event processing (CEP) technologies, transports, and protocols, an enterprise service bus (ESB) – the Mule open-source ESB [e] – is used in Rule Responder as the communication middleware. This ESB allows deploying the rule-based agents (see Figure 1) as distributed rule inference services installed as Web service-based endpoints on the Mule object broker and supports the communication in this rule-based agent processing network via a multitude of transport protocols. That is, the ESB provides a highly scalable and flexible application messaging framework to communicate synchronously or asynchronously amongst the ESB-local agents and with agents/services on the Web.

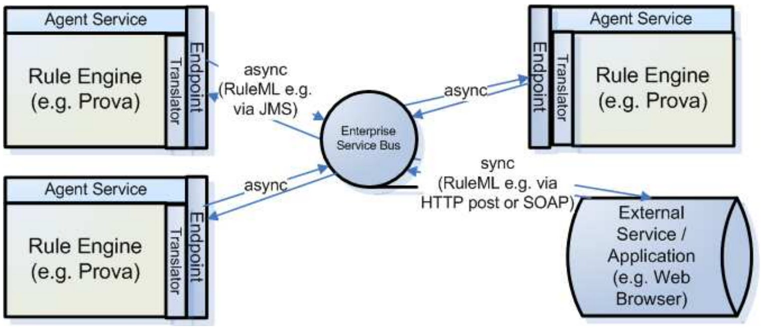

Fig. 1. Distributed Rule Responder Agent Services

The agent object broker follows the Staged Event Driven Architecture (SEDA) pattern [WCB01]. The basic approach of SEDA is to decompose a complex, event-driven application into a set of stages connected by queues. This design decouples event and thread scheduling from application logic and avoids the high overhead associated with thread-based concurrency models. That is, SEDA supports high concurrency demands on Web-based services and provides a highly scalable approach for asynchronous communication.

Figure 2 shows a simplified breakdown of the integration of Mule into the Rule Responder framework.

Distributed agent services (see Figure 1), which at their core run a platform specific rule engine, are deployed as Mule components which listen at configured endpoints, e.g., JMS message endpoints, HTTP ports, SOAP server/client addresses or JDBC database interfaces, etc. Reaction RuleML is used as a common platform-independent rule interchange format between the agents (and possible other rule

---

[e] *www.mulesoft.org*

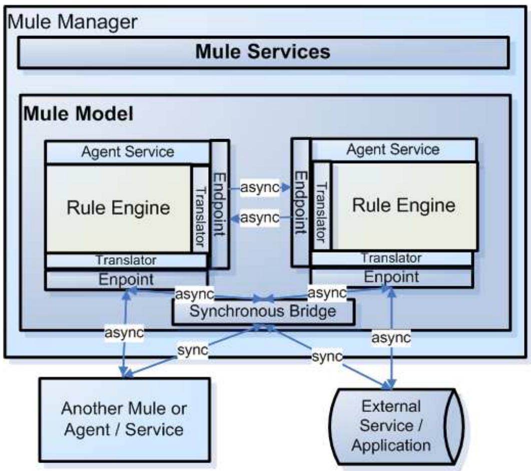

Fig. 2. Layering of Rule Responder on Mule ESB

execution / inference services). Translator services are used to translate inbound and outbound (event) messages from platform-independent Reaction RuleML into the platform-specific execution syntaxes of rule engines, and vice versa.

The large variety of transport protocols provided by Mule can be used to transport the messages to the registered endpoints or external applications / tools. Usually, JMS is used for the internal communication between distributed Prova rule agent instances, while HTTP and SOAP is used to access external Web services. The usual processing style is asynchronous using SEDA event queues. However, sometimes synchronous communication is needed. For instance, to handle communication with external synchronous HTTP clients such as Web browsers where requests, e.g. by a Web from, are sent through a synchronous HTTP channel. In this case, a synchronous bridge component (see Figure 1) dispatches the requests into the asynchronous messaging framework and collects all answers from the internal service nodes, while keeping the synchronous channel with the external service open. After all asynchronous answers have been collected, they are sent back to the still connected external service via the HTTP-synchronous channel.

3.2. *Platform-Specific Rule Engines for Rule Responder Agents*

The core of a Rule Responder agent, which is deployed as a service component on the Rule Responder ESB, is a platform-specific rule engine. These engines might differ, e.g., in their supported rule types, state representation, rule evaluation mechanism, conflict resolution and truth maintenance. Hence, depending on their expressiveness and functionalities, these rule engines might be capable of implementing agents in the strong sense of cognitive architectures for intelligent agents with goal/task-based, utility-based and learning-based functionalities, or in the weak sense of inference agent services with simple reflexive functionalities for, e.g., deductive query-answering capabilities. Following the general consensus defined by the strong notion of agency in ^{Woo01}, a Rule Responder agent, in addition to being (semi-)autonomous, should be capable of reactive, proactive, and communicative behavior. Additionally, it is often important that certain mentalistic notions[f] can be used in the rule language for describing the agent behavior in an abstract and intuitive way, e.g. in the interactions between agents to communicate the pragmatics of the interchanged information.

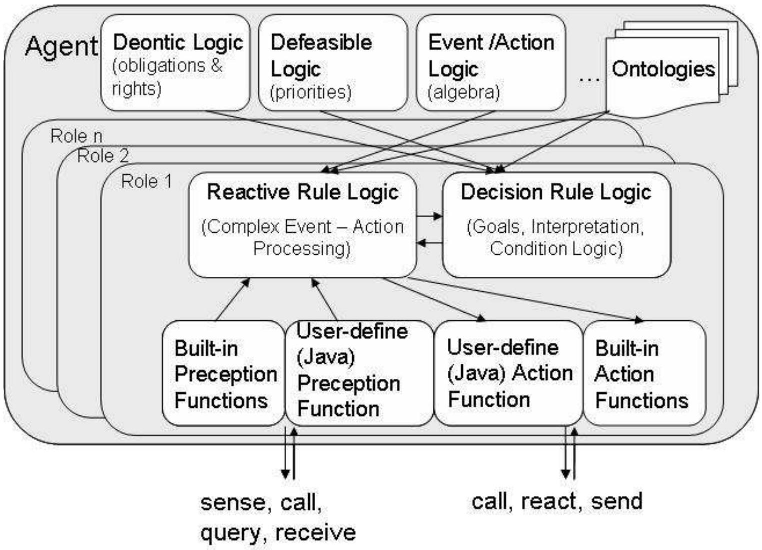
Agent
Deontic Logic
(obligations & rights)
Defeasible Logic
(priorities)
Event /Action Logic
(algebra)
...
Ontologies
Role n
Role 2
Role 1
Reactive Rule Logic
(Complex Event – Action Processing)
Decision Rule Logic
(Goals, Interpretation, Condition Logic)
Built-in Preception Functions
User-define (Java) Preception Function
User-define (Java) Action Function
Built-in Action Functions
sense, call, query, receive
call, react, send

Fig. 3. Rule Responder Agent

Figure 3 shows the architecture of an intelligent cognitive Rule Responder agent as it is implemented in the Prova. Prova[g] is an enterprise-strength, highly expressive distributed Semantic Web logic programming (LP) rule engine.

3.3. *Conversations Between Rule Responder Agents*

Rule Responder permits agents to use local platform specific languages and engines, only requiring that all rulebases, (event) messages, queries, and answers will be translated to Reaction RuleML as standard rule and event (message) interchange format for transmitting them to other agents.

---

[f] The term *mentalistic notions* aka *mental attitudes* refers to human properties such as beliefs, goals, etc. when transferred to describing machine agents.

[g] *http://prova.ws*

Reaction RuleML provides a translator service framework with Web form interfaces accepting controlled natural language input or predefined selection-based rule templates for the communication with external (human) agents on the computational independent level, as well as HTTP interfaces, and Web service SOAP interfaces, which can be used for translation into and from platform-specific rule languages such as Prova.

On the platform-independent and platform-specific level, the translator services are using different translation technologies such as XSLT stylesheet, JAXB, etc. to translate from and to Reaction RuleML as a general rule interchange format. The Reaction RuleML translator services are configured in the transport channels of the inbound and outbound links of the deployed rule engines on the ESB. Incoming Reaction RuleML messages (receive) are translated into platform-specific rulebases which can be executed by the rule engine, e.g. Prova, and outgoing rulebases (send) are translated into Reaction RuleML in the outbound channels before they are transferred via a selected transport protocol such as HTTP or JMS.

On the computation-independent level, online user interfaces allow external human agents issuing queries to Rule Responder agents (typically the OA) in a controlled natural language or with template-driven Web forms and receive answers. The translation between the used controlled English rule language (Attempto Controlled English [SG06]) and Reaction RuleML is based on domain-specific language translation rules in combination with a controlled English translator service.

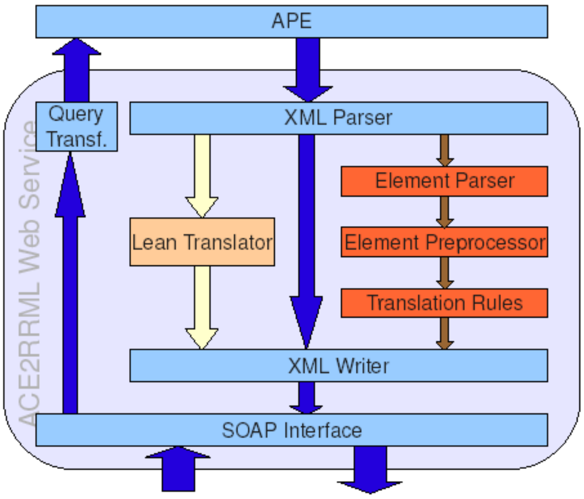

Fig. 4. ACE2RuleML Translator Web Service

Figure 4 shows the architecture of the ACE-to-Reaction RuleML translator Web service of the Rule Responder infrastructure. Queries to Rule Responder are formulated in Attempto Controlled English. The ACE2RML translator forwards the text to the Attempto Parsing Engine (APE), which translates the text into a discourse representation structure (DRS) and/or advices to correct malformed input. The DRS gives a logical/structural representation of the text. It is fed into an XML parser which translates it into a domain-specific Reaction RuleML representation of the query. Besides parsing and processing the elements of the DRS, the parser additionally employs domain-specific transformation rules and vocabularies to correctly translate the query into a public interface call of a Rule Responder OA.

4. Reaction RuleML as Standard Semantic Rule and Event Interchange Format

Reaction RuleML is used in Rule Responder as standardized rule and event interchange format between the (different) rule engines used for implementing the agents. Reaction RuleML incorporates various kinds of production, action, reaction, and KR temporal/event/action logic rules as well as (complex) event/action messages into the native RuleML syntax. The Reaction RuleML subfamily addresses the four major reaction rule types:

- Production Rules (Condition-Action rules) in the Production RuleML subfamily
- Event-Condition-Action (ECA) rules in the ECA RuleML subfamily
- Rule-based Complex Event Processing (complex event processing reaction rules, (distributed) event messaging reaction rules, query reaction rules etc.) in the CEP RuleML subfamily
- Knowledge Representation Event/Action/Situation Transition/Process Logics and Calculi in the KR Reaction RuleML subfamily

The general syntax of reaction rules is as follows:

```
<Rule style="active|messaging|reasoning">

     <meta>   <!-- (semantic) metadata of the rule -->                 </meta>
     <evaluation>   <!-- intended  semantics  -->                <evaluation>
     <qualification> <!-- e.g. qualifying rule declarations, e.g.
                         priorities, validity, strategy -->   </qualification>
     <quantification> <!-- quantifying rule declarations     </quantification>
     <scope>  <!-- scope of the rule e.g. a local rule module -->     </scope>
     <oid>     <!-- object id of the rule -->                           </oid>

     <on>      <!-- event part -->                                      </on>
     <if>      <!-- condition part -->                                  </if>
     <then>    <!-- (logical) conclusion part -->                     </then>
     <do>      <!--  action part -->                                    </do>
     <after>   <!-- postcondition part after action,
                   e.g. to check effects of execution -->           </after>
     <else>    <!-- (logical) else conclusion -->                     </else>
     <elsedo>  <!-- alternative/else action,
                     e.g. for default handling -->                </elsedo>
</Rule>
```

Depending on which parts of this general rule syntax are used different types of reaction rules can be expressed, e.g. if-then (derivation rules), if-do (production rules), on-do (trigger rules), on-if-do (ECA rules).

```
Derivation Rule:

<Rule style="reasoning">
  <if>...</if>
  <then>---</then>
</Rule>
```

```
Production Rule:

<Reaction style="active">
  <if>...</if>
  <do>---</do>
</Reaction>
```

```
ECA Rule:

<Reaction style="active">
  <on> ____ </on>
  <if> ...  </if>
  <do> ---- </do>
</Reaction>
```

```
CEP Rule:

<Rule style="active">
  <on> event 1 </on>
  <do> action  </do>
  <on> event 2 </on>
  <if> condition</if>
  <do> action </do>
  ...
</Rule>
```

Reaction RuleML provides several layers of expressiveness for adequately representing the agents' logic and for interchanging events (queries, actions, event data) and rules. In the following some of the needed Rule Responder expressiveness constructs of Reaction RuleML 1.0 are described.

4.1. *Reaction RuleML Messaging for Distributed Agent Conversation*

For communication between distributed rule-based (agent) systems Reaction RuleML provides a general message syntax:

```
<Message directive="<!-- pragmatic context -->">
   <oid>          <!-- conversation ID-->         </oid>
   <protocol>   <!-- transport protocol -->   </protocol>
   <sender>   <!-- sender agent/service -->     </sender>
   <receiver> <!-- receiver agent/service --> </receiver>
   <content>     <!-- message payload -->      </content>
</Message>
```

In the context of these Reaction RuleML messages agents can interchange events (e.g., queries and answers) as well as complete rule bases (rule set modules), e.g. for remote parallel task processing. Agents can be engaged in long running possibly asynchronous conversations and nested sub-conversations using the conversation id to manage the conversation state. The protocol is used to define the message passing and coordination protocol. The directive attribute corresponds to the pragmatic instruction, e.g. a FIPA ACL primitive, i.e. the pragmatic characterization of the message context broadly characterizing the meaning of the message.

For sending and receiving (event) messages Reaction RuleML 1.0 supports serial messaging CEP reaction rules which *receive* and *send* events in arbitrary combinations. A serial (messaging) reaction rule starts with a receiving event *on* followed by an arbitrary combination of conditions *if*, events *receive* and actions *send* in the body of the rule for expressing complex event processing logic. This flexibility with support for modularization and aspect-oriented weaving of reactive rule code is in particular useful in distributed systems where event processing agents communicate and form a distributed event processing network, as e.g. in the following example:

```
<Rule style="active">
  <on><Receive> receive event from agent 1 </Receive></on>
  <do><Send> query agent 2 for regular products in a new sub-conversation </Send></do>
  <on><Receive> receive results from sub conversation with agent 2   </Receive></on>
  <if> prove some conditions, e.g. make decisions on the received data </if>
  <do><Send> reply to agent 1 by sending results received from agent 2 </Send></do>
```

```
</Rule>
```

For better modularization the sub-conversation logic can be also written with an inlined reaction rule as follows:

```
<Rule style="active">
  <on><Receive> receive event from agent 1 </Receive></on>
  <if> <!- this goal activates the inlined reaction rule -- see below -->
    <Atom><Rel>regular</Rel><Var>prod</Var></Atom>
  </if>
  <do><Send> reply to agent 1 by sending results received from agent 2 </Send></do>
</Rule>

<Rule style="active">
  <then>
    <Atom><Rel>regular</Rel><Var>prod</Var></Atom>
  </then>
  <do><Send> query agent 2 for regular products in a new sub-conversation </Send></do>
  <on><Receive> receive results from sub conversation with agent 2   </Receive></on>
</Rule>
```

Reaction RuleML messaging reaction rules can be translated, e.g., into serial messaging Horn rules and executed in the Prova rule engine, so that incoming event queries can be processed by the Prova agents and derived answers can be send back as event messages.

4.2. *Event/Action/Situation Processing in Reaction RuleML*

Events, actions and situations states play a central role in the reactive (behavioral) logic of Rule Responder agents. Reaction RuleML defines a standard library of typical event and action algebra operator constructs to define complex event and actions:

```
Event Algebra:
    Sequence (Ordered), Disjunction (Or) , Xor (Mutal Exclusive),
    Conjunction (And), Concurrent, Not, Any, Aperiodic, Periodic
Action Algebra:
    Succession (Ordered Succession of Actions), Choice
    (Non-Deterministic Choice), Flow (Parallel Flow),
    Loop (Loops)

Complex Sequence (A;(B;C))
<on>
    <Sequence><Atom> A <Atom> <Sequence> <Atom> B </Atom> <Atom> C </Atom> </Sequence> </Sequence>
</on>
```

Furthermore, Reaction RuleML provides support for (re)using external event/action ontologies and metamodels which can be applied in generic operator definitions *Operator* for defining semantic (complex) event/action types and in the events to define specific event types. Different selection, consumption, and (transactional) execution policies for events and actions can be specified in the *evaluation* semantics for the complex event/action descriptions. This allows for a highly extensible and flexible Semantic CEP (SCEP) approach which (re-)uses external semantic models.

Reaction RuleML also defines syntax and semantics for knowledge representation event/action calculi such as Situation Calculus [MH69], Event Calculus [KS86] and Temporal Action Languages [DGKK98] etc. Specifically the notion of an explicit *state* (a.k.a. as state or fluent in Event Calculus) is introduced in the KR Reaction RuleML layer. An event/action *initiates* or *terminates* a state. That is, a state explicitly represents the abstract effect of occurred events and executed actions. Such states can be e.g. used for situation reasoning in the condition part of reaction rules.

```
<Rule style="reasoning">
  <on> <Receive> ... event message  ... </Receive> </on>
  <if> <HoldsState> ... state individual ... </HoldsState> </if>
  <do> <Send> ... action message ... </Send></do>
</Rule>
```

4.3. *Agent Interface Descriptions with Reaction RuleML Signatures*

Information hiding is another important concept for virtual collaboration of independent agents. In particular, local information used in the PAs becomes only accessible by authorized access via the public interfaces of the OAs which act as an abstraction layer supporting security and information hiding. To achieve this, Reaction RuleML provides an interface definition language which allows descriptions of the signatures of publicly accessibly rule functions together with their mode and type declarations. Modes are states of instantiation of the predicate described by mode declarations, i.e. declarations of the intended input-output constellations of the predicate terms with the following semantics:

- ”+” The term is intended to be input
- ”−” The term is intended to be output
- ”?” The term is undefined/arbitrary (input or output)

For instance, the interface definition for the function $add(Result, Arg1, Arg2)$ is $add(-, +, +)$, i.e. the function predicate add returns one output argument followed by two input arguments. Serialized in Reaction RuleML this would be:

```
<signature>
  <Atom>
    <Rel>add</Rel>
    <Var mode="-"/>
    <Var mode="+"/>
    <Var mode="+"/>
  </Atom>
</signature>
```

External agents can access the virtual organization only via these public interfaces, which often only reveal abstracted information to authorized users and hence hide local information of the organization and its PAs. That is, e.g. $add(X, 1, 1)?$ would be a valid query to this public signature function of an agent.

Other expressive constructs of Reaction RuleML needed for adequate implementations of Rule Responder agents will be discussed on the platform specific level in the next section.

5. Prova Semantic Rule Engine for Rule Responder Agents

On the platform-specific level Rule Responder allows arbitrary rule engines to be deployed as inference and execution environments for the agents as long as they provide a translator into the standard Reaction RuleML and as long as they support the required expressiveness to represent the respective agent's logic. In the following we describe the implementation of Rule Responder agents using Prova[h], which is a highly expressive Semantic Web rule engine.

The Prova rule engine supports different rule types:

- Derivation rules to describe the agent's decision logic
- Integrity rules to describe constraints and potential conflicts
- Normative rules to represent the agent's permissions, prohibitions and obligation policies
- Defeasible rules to prioritize rules for, e.g. handling conflicts between agent's goals and modularization of the agent's KB to support multiple roles of an agent
- Global ECA-style reaction rules to define global reaction logic which are triggered on the basis of detected (complex) events
- Messaging reaction rules to define the agents conversation-based workflow reactions and behavioral logics based on complex event processing

In the following subsections several extra logical expressiveness features of Prova will be describe, which are useful for implementing the logic of Rule Responder agents.

5.1. *Access to External Data, Type Systems and Procedural Attachments*

Prova follows the spirit and design of the W3C Semantic Web initiative and combines declarative rules, ontologies and inference with dynamic object-oriented programming and access to external data sources and type systems. Therefore, Prova assumes not just a single universe of discourse, but several domains, so called sorts (types) which are interpreted in a multi-sorted logic. The extension of the signature and the typed variables of the language alphabet with sorts (aka types) is defined as follows.

Definition 5.1. (**Multi-sorted Signature**) The multi-sorted signature S of Prova is defined as a tuple $\langle \overline{T}, \overline{P}, \ \overline{F}, arity, \overline{c}, sort \rangle$ where $\overline{T} = \{T_1, .., T_n\}$ is a set of sort/type symbols called sorts. The function *sort* associates with each predicate, function or constant its sorts:

- if c is a constant, then $sort(c)$ returns the type T of c.

---

[h] *http://prova.ws*

- if p is a predicate of arity k, then $sort(p)$ is a k-tuple of sorts $sort(p) = (T_1, .., T_k)$ where each term t_i of p is of some type T_j, i.e., $t_i : T_j$.
- if f is a function of arity k, then $sort(f)$ is a $k+1$-tuple of sorts $sort(f) = (T_1, .., T_k, T_{k+1})$ where $(T_1, .., T_k)$ defines the sorts of the domain of f and T_{k+1} defines the sorts of the range of f

Prova supports the following three basic types of sorts

(1) primitive sorts are given as a fixed set of primitive data types such as integer, string, etc.
(2) function sorts are complex sorts constructed from primitive sorts $T_1 \times ... \times T_n \rightarrow T_{n+1}$ or other complex sorts defined in the external type alphabet
(3) Boolean sorts are (predicate) statement of the form $T_1 \times ... \times T_n$

Definition 5.2. (**Multi-sorted Logic**) Prova's multi-sorted logic associates which each term, predicate and function a particular sort:

(1) Any constant or variable t is a term and its sort T is given by $sort(t)$
(2) Let $f(t_1, .., t_n)$ be a function then it is a term of sort T_{n+1} if $sort(f) = \langle T_1, .., T_n, T_{n+1} \rangle$, i.e., f takes argument of sort $T_1, .., T_n$ and returns arguments in sort T_{n+1}.

The intuitive meaning is that a predicate or function holds only if each of its terms is of the respective sort given by *sort*.

The alphabet of the Prova language builds on top of the standard ISO Prolog syntax standard, but further extends it. For typing each variable X_j in the multi-sorted alphabet of the Prova language is associated with a specific sort $sort(X_j) = T_i$, written as $X_j : T_i$, where X_j is a variable and T_i is a type sort associated with the variable. That is, the extended Prova language considers external sort/type alphabets. The combined signatures of the Prova rule language and the external type languages form the basis for combined hybrid knowledge bases and the integration of external type systems into the rule system.

Definition 5.3. (**Type alphabet**) An external type alphabet $\overline{T}$ is a finite set of monomorphic sort/type symbols built over the distinct set of terminological class concepts of a (external type) language.

Definition 5.4. (**Combined Signature**) A combined signature $\overline{S}$ is the union of all its constituent finite signatures: $\overline{S} = \langle S_1 \cup .. \cup S_n \rangle$

The type systems considered in Prova are order-sorted (i.e., with sub-type relations):

Definition 5.5. (**Order-sorted Type System**) A finite order-sorted type system TS comes with a partial order $\leq$, i.e., TS under $\leq$ has a greatest lower bound $glb(T_1, T_2)$ for any two types T_1 and T_2 having a lower bound at all. Since TS is finite also a least upper bound $lub(T_1, T_2)$ exists for any two types T_1 and T_2 having an upper bound at all.

Definition 5.6. (**Combined Knowledge Base**) The combined knowledge base of a typed Prova $\overline{KB} = \langle \Phi, \Psi \rangle$ consists of a finite set of (order-sorted) type systems / type knowledge bases $\Psi = \{\Psi_1 \cap .. \cap \Psi_n\}$ and a typed Prova KB Φ.

The combined signature is the union of all constituent signatures, i.e., each interpretation of a Prova rule program has the set of ground terms of the combined signature as its fixed universe.

Definition 5.7. (**Extended Herbrand Base**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a typed combined Prova KB P. The extended Herbrand base of P, denoted $\overline{B}(P)$, is the set of all ground literals which can be formed by using the predicate/function symbols in the combined signature with the ground typed terms in the combined universe $\overline{U}(P)$, which is the set of all ground typed terms which can be formed out of the constants, type and function symbols of the combined signature.

The grounding of the combined KB is computed wrt the composite signature.

Definition 5.8. (**Grounding**) Let P be a typed (combined) Prova KB and $\overline{c}$ its set of constant symbols in the combined signature. The grounding $ground(P)$ consists of all ground instances of all rules in P w.r.t to the combined multi-sorted signature which can be obtained as follows:

- The ground instantiation of a rule r is the collection of all formulae $r[X_1 : T_1/t_1, .., X_n : T_n/t_n]$ with $X_1, .., X_n$ denoting the variables and $T_1, .., T_n$ the types of the variables (which must not necessarily be disjoint) which occur in r and $t_1, .., t_n$ ranging over all constants in $\overline{c}$ wrt to their types.
- For every explicit query/goal $Q[X_1 : T_1, .., X_m : T_m]$ to the type system, being either a fact with one or more free typed variables $X_1 : T_1, .., X_m : T_m$ or a special built-in Prova query literal *rdf*(...) with variables as arguments in the triple-like query, the grounding $ground(Q)$ is an instantiation of all variables with constants (individuals) in $\overline{c}$ according to their types.

Using equalities Prova assumes a notion of default inequality for the combined set of individuals/constants which leads to a default unique name assumption:

Definition 5.9. (**Default Unique Name Assumption**) Two ground terms are assumed to be unequal, unless equality between the terms can be derived.

The interpretation I of a typed Prova program P then is a subset of the extended Herbrand base $\overline{B}(P)$.

Definition 5.10. (**Multi-sorted Interpretation**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a combined KB and $\overline{c}$ its set of constant symbols. An interpretation I for a multi-sorted combined signature $\overline{S}$ consists of

(1) a universe $|M| = T_1^I \cup T_2^I \cup .. \cup T_n^I$, which is the union of the types (sorts), and
(2) the predicates, function symbols and constansts/individuals $\overline{c}$ in the combined signature, which are interpreted in accordance with their types.

The assignment function σ from the set of variable $\overline{X}$ of P into the combined universe $\overline{U}(P)$ must respect the sorts/types of the variables (in order-sorted type systems also subtypes). That is, if X_i is a variable of type T, then $\sigma(X) \in T^I$. In general, if ϕ is a typed predicate or function in Φ and σ an assignment to the interpretation I, then $I \models \phi[\sigma]$, i.e., ϕ is true in I when each variable X of ϕ is substituted by the values $\sigma(X)$ wrt to its type. Since the assignment to constant and function symbols is fixed and the domain of discourse corresponds one-to-one with the constants $\overline{c}$ in the combined signature $\overline{U}(P)$, it is possible to identify an interpretation I with a subset of the extended Herbrand base: $I \subseteq \overline{B}(P)$.

The assignment function in Prova is given as a query from the rule component to the type system, so that there is a separation between the inferences in a type system and the rule component. Moreover, explicit queries to a type system (Java or Semantic Web) defined in the body of a rule, e.g., procedural attachments, built-ins or ontology queries (special *rdf* query or free DL-typed facts) are based on this hybrid query mechanism.

Definition 5.11. (**Semantic Multi-Structure Model**) Let $\overline{KB} = \langle \Phi, \Psi \rangle$ be a combined KB of a typed Prova program P.
An interpretation I is a model of an untyped ground atom $A \in \overline{KB}$ or I satisfies A, denoted $I \models A$ iff $A \in I$.
I is a model for a ground typed atom $A : T \in \overline{KB}$, or I satisfies $A : T$, denoted $I \models A : T$, iff $A : T \in I$ and for every typed term $t_i : T_j$ in A the type query $T_j = sort(t_i)$, denoting the type check "is t_i of type T_i", is entailed in $\overline{KB}$, i.e., $\overline{KB} \models T_i = sort(t_i)$ (note, in an order sorted type system subtypes are considered, i.e., t_i is of the same or a subtype of T_j).
I is an interpretation of an ground explicit query/goal Q to the type system Ψ if $\Psi \models Q$.
I is a model of a ground rule $r : H \leftarrow B$ iff $I \models H(r)$ whenever $I \models B(r)$. I is a model a typed program P (respective a combined knowledge base $\overline{KB}$), denoted by $I \models P$, if $I \models r$ for all $r \in ground(P)$.

Informally, a typed Prova knowledge base consists of rules with logic programming literals which have typed terms and a set of external (order-sorted) type systems in which the types (sorts) are defined over their type alphabets. An external type system might possibly define a complete knowledge base with types/sorts (Java classes or T-Box in DL) and individuals associated with these types (Java object instances of the classes or A-box in DL). Restricted built-in and procedural attachment predicates or functions which construct or return individuals of a certain type (boolean or object-valued) are also considered to be part of the external type system(s), i.e., part of the external signature. The combined signature is then the union of the two (or more) signatures, i.e., the combination of the signature of the rule component and the signatures of the external type systems / knowledge bases combining their type alphabets, their functions and predicates and their individuals.

The operational semantics of typed Prova is implemented as hybrid polymorphic order-sorted unification. [Pas06b] In contrast to other hybrid (DL-typing) approaches which apply additional constraint literals as type guards in the rule body and leave the usual machinery of resolution and unification unchanged, the operational semantics for prescriptive types in Prova's typed logic is implemented by an order-sorted unification. Here the specific computations that are performed in the typed language are intimately related to the types attached to the atomic term symbols. The order-sorted unification yields the term of the two sorts (types) in the given sort hierarchy. This ensures that type checks apply directly during typed unification of terms at runtime enabling ad-hoc polymorphism of variables leading e.g., to different optimized rule variants and early constrained search trees. Thus, the order-sorted mechanism provides higher level of abstraction, providing more compact and complete solutions and avoiding possibly expensive backtracking.

Prova provides support for two external order-sorted type systems, namely *Java* class hierarchies and ontological type systems (e.g. OWL or RDFS ontologies) respectively *Description Logic* knowledge bases.

5.1.1. *Description Logic Type Systems / Ontologies*

External type systems supported by Prova are Semantic Web ontologies (Description Logic KBs) represented, e.g., in RDFS or OWL. That is, the combined signature $\overline{S}_{DL}$ consisting of the finite signature S of the rule component and the finite signature(s) S_i of the ontology language(s).

The type alphabet TS is a finite set of monomorphic type symbols built over the distinct set of terminological atomic concepts $\overline{T}$ in a Semantic Web ontology language Σ^{DL}, i.e., defined by the atomic classes in the T-Box model.

Note, that restricting types to atomic concepts is not a real restriction, because for any complex concept such as $(T_1 \sqcap T_2)$ or $(T_1 \sqcup T_2)$ one may introduce an atomic concept T_3 in the T-Box and use T_3 as atomic type instead of the complex concept. This approach is also reasonable from a practical point of view since dynamic type checking must be computationally efficient in order to be usable in an order-sorted typed logic with possible very large rule derivation trees and many typed unification steps, i.e., fast type checks are crucial during typed term unification. We assume that the type alphabet is fixed (but arbitrary), i.e., no new terminological concepts can be introduced in the T-Box by the rules at runtime. This ensure completeness of the domain and enables static type checking on the used DL-types in Prova programs at compile time (during parsing the Prova script).

The set of constants/individuals $\overline{c}$ is built over the set of individual names in

Σ^{DL}, but we do not fix the constant names and allow arbitrary fresh constants (individuals) (under default UNA) to be introduced in the head of rules and facts of the rule base. However, new individuals which are introduced in rules or facts apply locally within the scope of the rules in which they are defined, i.e., within a local reasoning chain; in contrast to the individuals defined in the A-box model of the type system which apply globally as individuals of a class. DL-typed terms in Prova are defined as follows:

Definition 5.12. (**DL-typed Terms**) A DL-type is a terminological concept/class defined in the DL-type system (T-Box model). A typed DL-typed Prova term is denoted by the relation $t : T$ where t is the term and T is the DL-type of term.

The type ontologies are typically provided as Web ontologies (RDFS or OWL) where types and individuals are represented as resources having an webized URI. Namespaces can be used to avoid name conflicts and namespace abbreviations facilitate a more readable language.

```
% A customer gets 10 percent discount, if the customer is a gold customer

discount(X^^business:Customer, 10^^math:Percentage) :-
    gold(X^^business:Customer).

% fact with free typed variable acts as instance query on the ontology A-box
gold(X^^business:Customer).
```

Free DL-typed variables are allowed in facts. They act as free instance queries on the ontology layer, i.e., they query all individuals of the given type and bind them to the typed variable.

5.1.2. *Java Type System, Procedural Attachments and Built-Ins*

For external Java type systems, the combined multi-sorted signature $\overline{S}_{Java}$ uses the fully qualified order-sorted Java class hierarchy as type symbols. In order to type a variable with a Java type the fully qualified name of the Java class to which the variable should belong must be specified as a prefix separated from the variable by a dot ".".

```
java.lang.Integer.X         variable X is of type Integer
java.util.Calendar.T        variable T is of type Calendar
java.sql.Types.STRUCT.S     variable S is of SQL type Struct
```

To sense the environment and trigger actions, query data from external sources such as databases, call external procedural code such as Enterprise Java Beans, and receive / send messages from / to other agents or external services, Prova provides a set of built-in functions and additionally can dynamically instantiate any Java object and call its API methods at runtime.

Java objects, as instances of Java classes, can be dynamically constructed by calling their constructors or static methods using extra logical procedural attachments. The returned objects, might then be used as individuals/constants that are bound by an equality relation (denoting typed unification equality) to appropriate variables, i.e., the variables must be of the same type or of a super type of the Java object.

A procedural attachment is a function that is implemented by an external procedure (i.e., a Java method). They are used in Prova to dynamically call external procedural methods during runtime, i.e., they enable the (re)use of procedural code and allow dynamic access to external data sources and tools using their programming interfaces (APIs). Hence, in particular for Rule Responder agents, they are a crucial extension to traditional logic programming, combining the benefits of object-oriented languages (Java) with declarative rule based programming, e.g., in order to externalize mathematical computations such as aggregations to highly optimized procedural code in Java or use query languages such as SQL by JDBC to select and aggregate facts from external data sources.

Definition 5.13. (**Procedural Attachments**) A procedural attachment is a function or predicate whose implementation is given by an external procedure. Two types of procedural attachments are distinguished:

- **Boolean-valued attachments** (or **predicate attachments**) which call methods which return a Boolean value, i.e., which are of Boolean sort (type).
- **Object-valued attachments** (or **functional attachments**) which are treated as functions that take arguments and return one or more objects, i.e., which are of a function sort.

Functional Java attachments have a left-hand side with which the results (the returned object(s)) of the call are unified by a unification equality relation =, e.g., $C = java.util.Calendar.getInstance()$. If the left-hand side is a free (unassigned) variable the latter stores the result of the invocation. If the left-hand side is a bound variable or a list pattern the unification can succeed or fail according to the typed unification and consequently the call itself can succeed or fail. List structures are used on the left-hand side to allow matching of sets of constructed/returned objects to specified list patterns. A predicate attachment is assumed to be a test in such a way that the call succeeds only if a true Boolean variable is returned. Static, instance and constructor calls are supported in both predicate and functional attachments depending on their return type. Constructor calls follow the Java syntax with the fully qualified name of the class and the constructor arguments, e.g., $X = java.lang.Long(123)$. Static method calls require fully qualified class names to appear before the name of the static method followed by arguments, e.g., $Z = java.lang.Math.min(X, Y)$. Instance methods are mapped to concrete classes dynamically based on the type of the variable, i.e., the method of a previously bound Java object is called. They require a variable before the name of an instance method followed by the arguments, e.g., $S = X.toString()$.

```
add(java.lang.Integer.In1,java.lang.Integer.In2,Result):-
   Result = java.lang.Integer.In1 + java.lang.Integer.In2.
```

The rule takes two Integer variables $In1$ and $In2$ as input and returns the result which is bound to the untyped variable $Result$. Accordingly, a query $add(1, 1, Result)$? succeeds with an Integer object 2 bound to the $Result$ variable, while a query $add(''abc'', ''def'', Result)$? will fail.

It is important to note, that Java objects can be bound to variables and their methods can be dynamically used as procedural attachment functions anywhere during the reasoning process, i.e., in other rules. This enables a tight and highly expressive integration of external object oriented functions into declarative agent's rules' execution.

Definition 5.14. (**Built-in Predicates or Functions**) Built-in predicates or functions (built-ins) are special restricted procedural attachment predicate respective function symbols in the Prova language for concrete domains, e.g., integers or strings, that may occur in the body of a rules.

Examples are $+$, $=$, $assert$, $bound$, $free$ etc. For instance, Prova provides a rich library of built-ins for query languages such as SQL, SPARQL, and XQuery:

```
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,...
```

The following rule uses a SPARQL query built-in to access an RDF Friend-of-a-Friend (FOAF) profile published on the Web. The selected data is assigned to variables which can be used within an agent's rule logic, e.g. to expose the agent's contact data.

```
exampleSPARQLQuery(URL,Type) :-
 QueryString = ' PREFIX foaf:
                 PREFIX rdf:
                 SELECT ?contributor ?url ?type
                 FROM
                 WHERE {
                         ?contributor foaf:name "Bob DuCharme" .
                         ?contributor foaf:weblog ?url .
                         ?contributor rdf:type ?type . } ',
 sparql_select(QueryString,url(URL),type(Type)).
```

Note, that the structures in Java type systems are usually not considered as interpretations in the strict model-theoretic definition, but are composite structures involving several different structures whose elements have a certain inner composition. However, transformations of composite structures into their flat model theoretic presentations is in the majority of cases possible. From a practical point of view, it is convenient to neglect the inner composition of the elements of the universe of a structure. These elements are just considered as "abstract" points devoid of any inherent meaning. This structural mapping between objects from their interpretations in the Java universe to their interpretation in the rule system ignoring finer-grained differences that might arise from the respective definitions is given by the following isomorphism.

Definition 5.15. (Isomorphism) Let I_1, I_2 be two interpretations of the combined signature $\overline{S} = \{T_1, .., T_n\}$, then $f_{\cong} : |M_1| \rightarrow |M_2|$ is an isomorphism of I_1 and I_2 if $f_{\cong}$ is a one-to-one mapping from the universe $|M_1|$ of I_1 onto the universe $|M_2|$ of I_2 such that:

(1) For every type T_i, $t \in T_i^{I_1}$, iff $f_{\cong}(t) \in T_i^{I_2}$
(2) For every constant c, $f_{\cong}(c^{I_1}) \cong c^{I_2}$
(3) For every n-ary predicate symbol p with n-tuple $t_1, .., t_n \in |M_1|$, $\langle t_1, .., t_n \rangle \in p^{I_1}$ iff $\langle f_{\cong}(t_1), .., f_{\cong}(t_n) \rangle \in p^{I_2}$
(4) For every n-ary function symbol f with n-tuple $t_1, .., t_n, \in |M_1|$, $f_{\cong}(f^{I_1}(t_1, .., t_n)) \cong f^{I_2}(f_{\cong}(t_1), .., f_{\cong}(t_n))$

For instance, in Prova an isomorphism between Boolean Java objects and their model-theoretic truth value is defined, which makes it possible to treat boolean-valued procedural attachments as conditional body literals in rules and establish a model-theoretic interpretation as defined above between the Java type system and the model-theoretic semantics of the typed logic of the rule component. Other examples are String objects which are treated as standard constants in rules, i.e., the Java String object maps with the untyped theory of logic programming. Primitive datatype values, from the ontology respective XML domain (XSD datatypes) can be mapped similarly.

5.1.3. *Example - Responsibility Assignment Matrix Ontology for Agent Coordination*

As one possible way for coordination in a virtual organization the Rule Responder framework uses a 'pluggable' Responsibility Assignment Matrix (RAM) ontology to support the OA in its selection of a PA and its optional participating profiles underneath. A RAM describes the responsibility of agent roles in completing certain tasks or deliverables in a virtual organization. A standard RAM is a RACI matrix (Responsible, Accountable, Consulted, and Informed), with

- *R*esponsible – agents who do the work to achieve the task. There is typically one role with a participation type of Responsible, although others can be delegated to assist in the work required. Typically, the PAs are the responsible roles.
- *A*ccountable (also Approver or final Approving authority) – agent who is ultimately accountable for the correct and thorough completion of the deliverable or task, and the one to whom Responsible is accountable. Typically, this is

the OA which receives the answer from the PA and further processes it before forwarding it to the EA.
- Informed – the agent who is kept up-to-date on progress, often only on completion of the task or deliverable; and with whom there is just one-way communication. Typically, this is the EA who is informed about the result by the OA.

In a simple star-like Rule Responder agent topology, a single RAI matrix can be used in the OA to map an incoming query to the PA whose local knowledge base is deemed to be best suited for answering it. The RAI matrix is represented as an OWL ontology (OWL Lite) and can be used by a Rule Responder agent via querying it with the Semantic Web built-ins of Prova, binding the respective roles and their responsibilities to typed variables in the agent's rule logic. For instance, the following returns an agent's responsibility and role in a virtual organization by querying the concrete values from the imported RAM matrix (an OWL document) using the special Prova RDF query built-in (a simple RDF triple query instead of the more complex Prova SPARQL query built-in).

```
assigned(Agent,Responsibility,Role) :-
  rdf("http://www.csw.inf.fu-berlin.de/agents/ram.owl","owl",Agent,Role,Responsibility).
```

Many variants of the RAM with different role distinctions are possible such as RACI (with Consulted agents), RASCI (with Supporting agents) etc. - see, e.g., table 1.

Table 1. Responsibility Assignment Matrix

	General Chair	Program Chair	Publicity Chair
Symposium	responsible	consulted	supportive
Website	accountable	responsible	
Sponsoring	informed, signs	verifies	responsible
Submission	informed	responsible	
...	...	...	...

While RAMs (RACI matrix, Linear Responsibility Charts, etc.) are often used to represent stable semiotic structures of virtual organizations, where responsibilities are clearly defined for each role, it should be noted that Prova OAs can also implement other well-known agent coordination and negotiation mechanisms.

5.2. *Modularization, Scopes and Guards*

Modularization is another important concept for a virtual collaboration of independent agents, since each agents might play multiple roles in the same organization. Prova supports modularization of the agents knowledge base in order to implement the several different roles an agent might play in the same agent instance. Each role has its own set of reaction rules to autonomously react (potentially proactive) on detected situations (complex events) and its own set of decision rules to interpret goals and derive decisions according to conditional proofs. Moreover, modularization of the agent's KB makes it easier to maintain.

5.2.1. *Metadata Based Modularization and Module Imports/Updates*

Prova has a flexible approach towards modularization of the knowledge base which allows constructing metadata based views on the knowledge base, so called *scopes*. Therefore, Prova extends the rule language to a labelled logic programming rule language (LLP) with metadata annotations such as rule labels, module (rule sets in rule bases) labels and arbitrary other (Semantic Web) annotations (e.g., Dublin Core author, date etc). These metadata annotations are used to manage the rules and facts in the knowledge base.

In analogy to the multi-sorted extension for types, the meta-data extension of the Prova language is defined over a combined signature $\overline{S}$ which is the union of the signature of the rule language and the signatures of the used metadata vocabularies (e.g. Dublin Core).

Definition 5.16. (**Combined Signature with Metadata Annotations**) The combined metadata annotated signature $\overline{S}$ is defined as a tuple $\langle \overline{T}, \overline{P}, \overline{F}, arity, \overline{c}$, $sort$, $meta\rangle$ where $\overline{P}$ is the union of the predicate symbols defined in the signature of the core Prova rule language and the metadata predicate symbols (denoting metadata key properties) defined in the signature(s) of the metadata vocabularie(s) and $\overline{c}$ is the union of constant symbols defined in the rule signature and in the metadata signature(s) (denoting metadata values). *meta* is a special unary function which returns the assigned metadata.

To explicitly annotate clauses in a Prova program P with an additional set of metadata labels a general 1-ary built-in function @ is introduced in the Prova language.

Definition 5.17. (**Metadata Annotation Labels**) The special 1-ary built-in function @ is a partial injective labelling function that assigns a set of metadata annotations m (property-value pairs) to a clause cl in P, e.g.

$@(L_1), .., @(L_n)\ H : -B$

where L_i are a finite set of unary positive literals (positive metadata literals) which denote an arbitrary metadata $property(value)$ pair, e.g., $@label(rule1)$.

The implicit form $@(L_1), .., @(L_n)\ H : -B$ of the metadata function expresses that $@(H : -B) = L_1, .., L_n$. The explicit @() annotation is optional, i.e., a Prova program P without metadata annotated clauses coincides with a standard unlabelled logic program.

Clauses in Prova are treated as objects in KB having an unique *object id* (*oid*) which might be user-defined, i.e., explicitly defined by a metadata annotation $@label(oid)\ H : -B$ or system-defined i.e., all rules are automatically "labelled" with an auto-incremented *oid* (an increasing natural number) provided by the system at compile time. Rules and facts might be bundled to clause sets, so called *modules*, which also have an object id, the module oid. By default the module oid is the URI or full document name of the Prova script which defines the module. But the module oid might also be user-defined $@src(moduleoid)$. All clauses (rules and facts) defined in a module are automatically annotated with the module oid $@src(moduleoid)\ H:-B$. The oids are used to manage the knowledge in the (distributed) knowledge base, e.g., to import a rule set from an URI which is then used as the module oid or remove a module from the KB by its oid. Beside oids arbitrary other semantic annotations such as Dublin Core data might be specified in the @ annotation function.

```
@label(r1) @dc:author("Adrian") @dc:date(2006-11-12)
    p(X):-q(X).
@label(f1)
    q(1).
```

The example shows a rule with rule label $r1$ and two additional Dublin Core annotations $dc:author(''Adrian'')$ and $dc:date(2006-11-12)$ and a fact with fact label $f1$. Since there is no explicitly user-defined module oid in the meta-data labels, the default module oid for both clauses is the URI or document name of the Prova script in which they are defined, e.g. $@src(''http://prova.ws/example1.prova'')$.

In Prova it is possible to consult (import/load) distributed rulebases from local files, a Web address, or from incoming messages transporting a rulebase. Furthermore, Prova supports update built-ins such as *assert* and *retract*.

```
%load from a local file
:- eval(consult("organization2009.prova")).
% import from a Web address
:- eval(consult("http://ruleml.org/organization2010.prova")).
```

The imported rulebases are managed as modules in the knowledge base, which are uniquely identified by their source object id $src(moduleOID)$. Since multiple nested imports are possible, modules might be nested, i.e. a module denoting a rule base (e.g. a Prova script) might consist of several nested submodules (e.g. sets of rules and facts).

Similar to imports of external type systems and built-ins (procedural attachments) which query and compute external data, the semantics for modules in Prova is defined over the combined knowledge base of the modules, an extended state based Herbrand Base and semantic multi-structures.

Definition 5.18. (**Combined Knowledge Base**) The combined knowledge base of a modular Prova $\overline{KB} = \langle \Phi, \Psi \rangle$ consists of a finite set of modules $\Psi = \{\Psi_1 \cap .. \cap \Psi_n\}$ and an initial primary Prova KB Φ.

Prova supports knowledge updates which import modules (*consult*) and add or remove clauses (*assert*, *retract*). Each update leads to a new knowledge state of the combined KB.

Definition 5.19. (**Knowledge State**) A knowledge state represents the combined knowledge base KB_k at this particular state, where $k \in \aleph$.

Note that according to the modularized logic in Prova a state, i.e., a combined knowledge base KB_k, might consist of nested submodules, each having an unique ID (the module oid). Intuitively, a state represents the union of all clauses stored in all modules in the combined knowledge base.

An update is then a transition which adds or removes facts and/or rules and changes the knowledge base. That is, the KB transits from the initial state KB_1 to a new state KB_2. We define the following notion of positive (assert) and negative(retract) transition:

Definition 5.20. (**Positive Update Transition**) A positive update transition, or simply positive update, to a knowledge state KB_k is defined as a finite set $U^{pos}_{oid} := \{r_N : H : -B, fact_M : A\}$ with A an atom denoting a fact, $H : -B$ a rule, $N = 0, .., n$ and $M = 0, ..m$ and oid being the update oid which is also used as module oid to manage the knowledge as a new module in the KB. Applying U^{pos}_{oid} to KB_k leads to the extended state $KB_{k+1} = \{KB_k \cup U^{pos}_{oid}\}$. Applying several positive updates as an increasing finite sequence $U^{pos}_{oid_j}$ with $j = 0, .., k$ and $U^{pos}_{oid_0} := \emptyset$ to KB_0 leads to a state $KB_k = \{KB_0 \cup U^{pos}_{oid_0} \cup U^{pos}_{oid_1} \cup ... \cup U^{pos}_{oid_k}\}$.

That is a state KB_k is decomposable in the previous knowledge state $k - 1$ plus the update: $KB_k = \{KB_{k-1} \cup U^{pos}_k\}$. We define $KB_0 = \{\emptyset \cup U^{pos}_{oid_0}\}$ and $U^{pos}_{oid_0} = \{KB : \text{the set of rules and facts defined in the program } P\}$, i.e., importing the initial Prova program P from a Prova script document is the first update leading to the knowledge state KB_1.

Likewise, We define a *negative update transition* as follows:

Definition 5.21. (**Negative Update Transition**) A negative update transition, or for short a negative update, to a knowledge state KB_k is a finite set $U^{neg}_{oid} := \{r_N : H : -B, fact_M : A\}$ with $A \in KB_k$, $H : -B \in P$, $N = 0, .., n$ and $M = 0, ..m$, which is removed from KB_k, leading to the reduced program $KB_{k+1} = \{KB_k \setminus U^{neg}_{oid}\}$.

Applying arbitrary sequences of positive and negative updates leads to a sequence of KB states $KB_0, .., KB_k$ where each state KB_i is defined by either $KB_i = KB_{i-1} \cup U^{pos}_{oid_i}$ or $KB_i = KB_{i-1} \setminus U^{neg}_{oid_i}$. In other words, KB_i, i.e., the set of all clauses in the KB at a particular knowledge state i, is decomposable in the previous knowledge state plus/minus an update, whereas the previous state consists of the state $i - 2$ plus/minus an update and so on. Hence, each particular knowledge state can be decomposed in the initial state KB_0 and a sequence of updates. Although an update might insert more than one rule or fact, i.e., insert or remove a complete module, it is nevertheless treated as an elementary update, a so called bulk update, which transits the current knowledge state to the next state in an elementary transition: $\langle KB_i, U^{pos/neg}_{oid}, KB_{k+1}\rangle$. Intuitively, one might think of it as a complex atomic update action which performs all knowledge inserts respective removes simultaneously.

Elementary updates have both a truth value, i.e. they may succeed or fail, and a side effect on the knowledge base leading to the transition of the knowledge state. The extended Herbrand Base is defined on the notion of knowledge states and transitions from one state to another.

Definition 5.22. (**Extended State-Based Herbrand Base**) Let P be the combined KB at a particular knowledge state KB_k. The extended Herbrand base of P, denoted $\overline{B}(P)$, is the set of all ground literals which can be formed by using the predicate/function symbols in the combined signature with the ground typed terms in the combined universe $\overline{U}(P)$, which is the set of all ground typed terms which can be formed out of the constants, type and function symbols of the combined signature of KB_k.

Definition 5.23. (**Modular semantic multi-structure**) A modular multi-structure I is the model of a modular program P (respective the knowledge state KB_k of the combined knowledge base $\overline{KB}$), denoted by $I \models P$, if $I \models c$ for all clauses $c \in ground(P)$, where $I \models c$ is a usual multi-sorted model for providing the interpretation of Prova clauses.

Accordingly, all queries to a Prova program apply on the extended respective reduced transition knowledge state of the program, i.e., the truth valuation of a goal G depends on its model at the current knowledge state KB_k, denoted by $TVal_{KB_k \models G}(G)$.

Based on this modular knowledge state transition semantics and the metadata based control of the knowledge state updates which are treated as modules in the combined KB, Prova provides supports for transactional updates, where failing sequences of knowledge updates can be rolled back by removing the associated modules from the combined Prova KB. [Pas06a] In the non-transactional style update action within (serial) Prova rules are not rolled-back to the original state if the derivation fails and the system backtracks. Typically this "weak" non-transactional semantics is intended when external Prova scripts are imported (consult) or new rule sets are added (assert) as modules. That is, independently, of whether the particular derivation in which the update is performed fails from some reason the update transition to the next knowledge state subsists and is not rolled back in case of failures.

5.2.2. *Scoped Reasoning*

The metadata annotation of rules/facts and rule sets (modules) enables scoped (meta) reasoning with the semantic annotations. The metadata can act as an explicit scope for constructive queries (creating a view) on the knowledge base. For instance, the metadata annotations might be used to constrain the level of generality of a scoped goal literal to a particular module, i.e., to consider only the set of rules and facts which belong to the specified module.

Definition 5.24. (**Scoped Literal**) A *scoped literal* is of the form $@\overline{C}\ L$ where L is a positive or negative literal and $@\overline{C}$ is the scope definition which is a set of one or more metadata constraints. Scoped literals are only allowed in the body of a rule.

Informally, the semantics of scoped literals allows to explicitly close the domain of discourse to certain parts of the KB.

Definition 5.25. (**Metadata-based Scope**) Let $\overline{KB}$ be a combined KB consisting of a set of submodules $\overline{KB} = \{KB_1 \cup .. \cup KB_k\}$. The scope KB' of a scoped literal $@\overline{C}\ L$ is the set of clauses $KB' = \{m'_1 cl_1, .., m'_n cl_n\} \in \overline{KB}$, where for all clauses $cl_i(m'_i) \in \overline{KB'}$ its set of metadata annotations m'_i satisfy the scope constraints $\overline{C}$ of the scoped literal L, i.e., $m'_i \models \overline{C}$.

Accordingly, a scope (aka constructive view) is constructed by one or more metadata constraints, e.g., the module oid $@src(URI/Filename)$ or Dublin Core values $@dc : author(...)$.

Definition 5.26. (**Closure**) Let $\overline{KB}$ be a combined KB. The closure of $\overline{KB}$, denoted $Cl(\overline{KB})$, is defined by $\overline{KB}$ plus all modules KB_k which are in the scope of any scoped literal in $\overline{KB}$.

A scoped literal $@\overline{C}\ L$ is closed if each rule in $\overline{KB}$ which unifies with the literal L is also closed, i.e., its body literals are closed in $Cl(\overline{KB})$.

Intuitively, this means that the closure of a Prova program depends on the scopes of the literals in the bodies of its rules. Obviously, if one of the subsequently used goal literals in a proof attempt is open, i.e., without a scope, the closure expands to the open KB.

Definition 5.27. (**Scoped Semantics**) Given a scoped $\overline{KB}'$, where all literals are scoped with closure $Cl(\overline{KB}')$, the truth value of a scoped literal $@\overline{C}\ L$ depends on the partial model of the clauses of $\overline{KB}'$ wrt the scope definition $\overline{C}$, i.e., $I_{partial^{\overline{C}}}(\overline{KB}') \models L$.

Syntactically the scope definitions use the syntax of Prova metadata annotations.

```
@label(rule1) r1(X):-q(X).
@label(rule2) r1(X):-q(X).
@label(rule3) p1(X):-
              @label(rule1) r1(X). % scoped goal literal
q(1).

:-solve(p1(Y)).
```

The example shows three metadata annotated rules. They query $p1(Y)$ will return only one solution with $Y = 1$, since the subgoal $r1(X)$ of $rule3$ applies only in the scope of the rule with label $rule1$, but not on $rule1$ and $rule2$, which would be the case if there would be no scope constraint defined for the subgoal.

Prova allows variables in the scope definitions which are bound to the annotated metadata values. The following example shows the definition of a scope, that constraints the application of the subgoal $r2(X)$ on the rule with label $rule3$ and on the module with source name $AgentRole1.prova$.

```
% get module label
r1(X,Y):-
    @src(Y) @label(rule3)
    r2(X).
:-solve(r1(X,"AgentRole1.prova")).
```

5.2.3. *Guards*

In addition to scopes Prova supports literal *guards* which act as additional precondition constraints.

Guards in Prova are syntactically specified in the Prova rule language using brackets after the goal literal. The model-theoretic semantics of guards is like for goal literals, however in the proof-theoretic semantics guards act like pre-conditions before the proofs of the standard goal literals starts.

For instance, the following rule makes decisions on the basis of rules which haven been authored by different persons and only applies those rules from trusted authors.

```
%simplified decision rules of an agent
@author(dev22) r2(X):-q(X).
@author(dev32) r2(X):-s(X).
q(2).
s(-2).

% for simplicity this is a fact, but could be also a complex rule
% which computes the trust value from the reputation value of dev22
trusted(dev22).

% Author dev22 is trusted but dev32 is not, so one solution is found: X=2
p1(X):-
 @author(A)
 r2(X) [trusted(A)].

% for all query
:-solve(p1(X1)).
```

This example uses metadata annotations on rules for the head literals $r2$ and a scope on the literal $r2(X)$ in the body of the rule for $p1(X)$. Since variable A in $@author(A)$ is initially free, it gets instantiated from the matching target rule(s). Once A is instantiated to the target rule's $@author$ annotation's value ($dev22$, for the first $r2$ rule), the body of the target rule is dynamically non-destructively modified to include all the literals in the additional guard $trusted(A)$ before the body start, after which the processing continues. Since $trusted(dev22)$ is true but $trusted(dev32)$ is not, only the first rule for predicate r2 is used and so one solution $X1 = 2$ is returned by $solve(p1(X1))$.

5.3. *Prova Serial Horn Rules for Messaging*

For communication between distributed agents Prova supports special built-ins for asynchronously sending and receiving event messages within serial Horn rules. The main language constructs of messaging reaction rules are: *sendMsg* predicates to send messages, reaction *rcvMsg* rules which react to inbound messages, and *rcvMsg* or *rcvMult* inline reactions in the body of messaging reaction rules to receive one or more context-dependent multiple inbound event messages:

```
sendMsg(XID,Protocol,Agent,Performative,Payload |Context)
rcvMsg(XID,Protocol,From,Performative,Paylod|Context)
rcvMult(XID,Protocol,From,Performative,Paylod|Context)
```

Here, *XID* is the conversation identifier (conversation-id) of the conversation to which the message will belong. *Protocol* defines the transport protocol. *Agent* denotes the target party of the message. *Performative* describes the pragmatic envelope for the message content. A standard nomenclature of performatives is, e.g., the FIPA Agents Communication Language (ACL). *Payload* represents the message content sent in the message envelope. It can be a specific query or answer or a complex interchanged rule base (set of rules and facts). For instance, the following rule snippet shows how a query is sent to an agent via the ESB and then an answer is received from this agent.

```
...
sendMsg(Sub_CID,esb,Agent,acl:query-ref, Query),
rcvMsg(Sub_CID,esb,Agent,acl:inform-ref, Answer),
...
```

Prova does not define a specific set of mentalistic notions as first-class programming constructs. Instead, interchanged messages besides the conversation's metadata and payload also carry the pragmatic context of the conversation such as communicative situations / acts, mentalistic notions, organizational and individual norms, purposes or individual goals and values. The payload of incoming event messages is interpreted with respect to the local conversation state, which is denoted by the conversation id, and the pragmatic context, which is given by a pragmatic performative. For instance, a standard nomenclature of pragmatic performatives, which can be integrated as external (semantic) vocabulary/ontology, is e.g., defined by the Knowledge Query Manipulation Language (KQML) [FLM97], by the FIPA Agent Communication Language (ACL) [?], which gives several speech act theory-based communicative acts, or by the Deontic Logic with its normative concepts for obligations, permissions, and prohibitions. Depending on the pragmatic context, the message payload is used, e.g. to update the internal knowledge of the agent (e.g., add new facts or rulebases), add new tasks (goals), or detect a complex event pattern (from the internal event instance sequence). For instance, the following example shows a reaction rule that sends a complete rule base, which is loaded from a local *File* to an agent service *Remote* using JMS as transport protocol.

Example 5.1.

```
% Upload a rule base read from File to the host
% at address Remote via JMS
upload_mobile_code(Remote,File) :-
    % Opening a file returns an instance
    % of java.io.BufferedReader in Reader
    fopen(File,Reader),
    Writer = java.io.StringWriter(),
    copy(Reader,Writer),
    Text = Writer.toString(),
    % variable SB will encapsulate the whole content of File
    SB = StringBuffer(Text),
    % send the complete rule base to the receiver agent "Remote"
    sendMsg(XID,jms,Remote,acl:inform,consult(SB)).
```

The corresponding receiving reaction rule of the remote agent is:

```
% wait for incoming messages with pragmatic context $acl:inform$
rcvMsg(XID,jms,Sender,acl:inform,[Predicate|Args]):-
    % derive the message payload, i.e. consult the received rule set to the internal KB
    derive([Predicate|Args]).
```

This rule receives incoming JMS-based messages with the pragmatic context $acl : inform$ and derives the message content, i.e. consults the received rule base to the local knowledge base of the remote agent. It is important to note that via the conversation id several reaction rule reasoning processes might run in parallel, local to their conversation flows. Inactive reactions (conversation partitions) are removed from the system, e.g. by timeouts. Self-activations by sending a message to the receiver "self" are possible. With the pragmatic performatives it is possible to implement different coordination and negotiation protocols. For instance, if an agent does not understand the semantics of the interchanged message payload, it can inform the sender about this, using, e.g., the $acl : not - understood$ performative, so that the sender can additionally send the semantic information, e.g. a pointer to the ontology that defines the concepts of the payload, and the receiving agent can import this ontology to its internal knowledge base.

For implementing the Rule Responder communication flows in the OAs, Prova messaging reaction rules are used. A typical coordination pattern implemented in a Rule Responder OA is the following messaging reaction rule (Prova variables start with an upper-case letter), which waits for an incoming query from an EA and delegates this query to an internal responsible PA.

```
% receive query and delegate it to another party
rcvMsg(CID,esb, Requester, acl:query-ref, Query) :-
  responsibleRole(Agent, Query),
  sendMsg(Sub-CID,esb,Agent,acl:query-ref, Query),
  rcvMsg(Sub-CID,esb,Agent,acl:inform-ref, Answer),
  ... (other goals)...
  sendMsg(CID,esb,Requester,acl:inform-ref,Answer).
```

When activated by an incoming request from an EA, e.g. an HTTP request coming from a Web form, this messaging reaction rule first selects the responsible role for the query. Then the rule sends the query in a new sub-conversation to the selected party and waits for the answer to the query. That is, the rule execution waits until an answer event message is received in the inlined sub-conversation, which activates the process flow again, e.g. to prove further 'standard' goals, e.g. with information from the received answer, which is unified with variables in the normal logic programming way, including also backtracking to other variable assignments. Finally, in this example, the rule sends back the answer to the original requesting EA.

Remarkably, Prova's messaging reaction rules do not require separate threads for handling multiple conversation situations simultaneously. Hence, new subconversations can be started with other PAs in parallel. This can be used for implementing, e.g., a Contract Net coordination protocol, where PAs bid for the task offered by the OA and the OA selects the best PA according to the received bids, or a publish-subscribe protocol, where PAs are selected according to their subscriptions with the OA.

By using messaging reaction rules Prova can be deployed as a distributed rule inference service in a Rule Responder agent, or e.g. as an OSGi [i] component enabling massive parallelization of Prova agent nodes in grid/cloud environments and (smart) devices (e.g. RFID networks) which communicate via event messages.

Several other expressive logic formalisms are supported by Prova [Pas07], e.g., for updating the knowledge base (transactional update logic), defining and detecting complex events (complex event algebra), handling situations/states (event calculus), as well as for reasoning (e.g., deontic logic for normative reasoning on permissions, prohibitions, obligations) and planning (abductive reasoning on plans and goals).

In summary, Prova agents can interchange event information, rules (tasks), and queries/answers in agent conversations, including information about the semantics and pragmatics of the interchanged information.

6. Recent Rule Responder Instantiations

Early instantiations of Rule Responder include the Health Care and Life Sciences eScience infrastructure [Pas08], the Rule-based IT Service Level Managment, and Semantic BPM system [PB08,PK08]. Recent instantiations include multiple versions of the deployed SymposiumPlanner system [CB08], two versions of the WellnessRules prototype [BOC09], and PatientSupporter. We will here highlight the principles of Rule Responder instantiations with an emphasis on the recent ones.

6.1. *SymposiumPlanner*

SymposiumPlanner is a series of deployed applications created with Rule Responder for the Q&A parts of the official websites of the RuleML Symposia.

Rule Responder started to support the organizing committee of the RuleML-2007 Symposium [Cra07] and was further developed to assist the yearly RuleML Symposia since. These applications embody responsibility assignment, automated first-level contacts for information regarding the symposium, helping the publicity

---

[i] Open Services Gateway initiative standard

chair with sponsoring correspondence, helping the panel chair with managing panel participants, and the liason chair with coordinating organization partners.

SymposiumPlanner utilizes a single organizational agent to handle the filtering and delegation of incoming queries. Each committee chair has a personal agent that acts in a rule-governed manner on behalf of the committee member. Each agent manages personal information, such as a FOAF-like profile containing a layer of facts about the committee member as well as FOAF-extending rules. These rules allow the PA to automatically respond to requests concerning the RuleML Symposium. Task responsibility for the organization is currently managed through a responsibility matrix, which defines the tasks committee members are responsible for. The matrix and the roles assigned within the virtual organization are defined by an OWL (Ontology Web Language) Lite Ontology.

External agents and the RuleML-2008 agents can communicate by sending messages that transport queries, answers, or complete rulebases through the public interface of the OA (e.g., an EA can use an HTTP port to which `post` and `get` requests can be sent from a Web form).

The Rule Responder instantation to SymposiumPlanner is further described and demonstrated online.[j]

6.2. *WellnessRules*

WellnessRules is a system supporting the management of wellness practices within a community based on rules plus ontologies. The idea is the following. As in Friend of a Friend (FOAF)[k], people can choose a (community-unique) nickname and create semantic profiles about themselves, here about their wellness practices, for their own planning and to network with other people supported by a system that 'understands' those profiles. As in FindXpRT [LBBM06], such FOAF-like fact-only profiles are extended with rules to capture conditional person-centered knowledge such as each person's wellness activity depending on the season, the time-of-day, the weather, etc. People can use rules of various refinement levels and rule languages ranging from pure Prolog to N3, which will be interoperated through RuleML/XML [Bol07].

Interoperating with translators, WellnessRules thus frees participants from using any single rule language. In particular, it bridges between Prolog as the main Logic Programming rule paradigm and N3 as the main Semantic Web rule paradigm. The distributed nature of Rule Responder profiles, each queried by its own (copy of an) engine, permits scalable knowledge representation and processing.

WellnessRules has recently been developed to WellnessRules2, using a new kind of agents, a Service Agent (SA), for accessing Google weather data. From the point of the OA, an SA can be queried similarly to a PA. However, while a PA is a personal

---

[j] *http://ruleml.org/SymposiumPlanner*

[k] *http://www.foaf-project.org/*

assistant to a human owner, an SA is just a machine agent, in WellnessRules2 acting as a wrapper for a (Web) service.

The Rule Responder instantations to WellnessRules are further described and demoed online.[1]

6.3. *PatientSupporter*

Patients are increasingly seeking interaction in support groups, which provide shared information and experience about diagnoses, treatment, etc. We present a Social Semantic Web prototype, PatientSupporter, that will enable such networking between patients within a virtual organization. PatientSupporter is an instantiation of Rule Responder that permits each patient to query other patients' profiles for finding or initiating a matching group.

Rule Responder's External Agent (EA) is a Web-based patient-organization interface that passes queries to the Organizational Agent (OA). The OA represents the common knowledge of the virtual patient organization, delegates queries to relevant Personal Agents (PAs), and hands validated PA answers back to the EA. Each PA represents the medical subarea of primary interest to a corresponding patient group. The PA assists its patients by advertising their interest profiles employing rules about diagnoses and treatments as well as interaction constraints such as time, location, age range, gender, and number of participants.

PAs can be distributed across different rule engines using different rule languages (e.g., Prolog and N3), where rules, queries, and answers are interchanged via translation to and from RuleML/XML. We discuss the implementation of PatientSupporter in a use case where the PA's medical subareas are defined through sports injuries structured by a partonomy of affected body parts.

PatientSupporter uses ontologies and rules for organizing geographically distributed patients – here, suffering from sports injuries – into virtual support groups around classes of an ontology of injuries – here, a sports-injury partonomy. The prototype is designed to help patients with a similar sports injury to interact with a virtual support group having that common interest. Patients in an online PatientSupporter virtual organization create their semantic profile referring to classes in a disease ontology – here a partonomy of body parts affected by sports injuries. Profiles contain rules about diagnoses and treatments as well as interaction constraints such as time, location, age range, gender, and number of participants. A patient can pose queries against the semantic profiles of other patients in his or her virtual organization to find or initiate a matching group.

PatientSupporter allows patients to have their profiles expressed in either Pure Prolog (Logic Programming rules) or N3 (Semantic Web rules). Providing these quite different rule language paradigms permit patients to choose the language

---

[1] *http://ruleml.org/WellnessRules and http://ruleml.org/WellnessRules2*

that best suits them. Rule Responder handles the interoperation between the rule languages of different patients using translators to and from RuleML/XML as the interchange format [BTW01,Bol07].

Patients using the PatientSupporter Social Semantic Web portal are able to initiate the virtual support group about their sports injury on a global scale. They also benefit from PatientSupporter's interoperation facility in the background – to transform patient profiles between Pure Prolog and N3 through RuleML/XML. The system employs a partonomy of sports-injury-affected body parts (a 'body partonomy'), which makes it easy for patients to navigate hierarchically up or down to increase recall or precision, respectively. A patient's queries invoke other patients' interaction rules, allowing him or her to narrow down the search in a step-wise fashion. All of this saves a patient from browsing through a large set of irrelevant patient profiles and permits him or her to efficiently converge on a first Skype call.

The Rule Responder instantation to PatientSupporter is further described and demoed online.[m]

6.4. *Reputation Management System*

The Rule Responder reputation management system [APM10] is based on distributed Rule Responder rule agents, which use rules for implementing the reputation management functionalities as rule agents, and which use Semantic Web ontologies for representing simple or complex multi-dimensional reputation objects. This Semantic Web reputation ontology model enables reputation portability, eases the management of reputation data, mitigates risks in open environments, and enhances the decision making process in the reputation processing agents. The reputation management system computes, manages, and provides reputation about entities which act on the Web. It is implemented as a *Reputation Processing Network (RPN)* consisting of *Reputation Processing Agents (RPAs)* that have two different roles:

1. *Reputation Authority Agents (RAAs)*: Act as reputation scoring services for the reputee entities whose Reputation Objects (ROs) are being considered or calculated in the agents' rule-based Reputation Computation Services (RCSs). An RCS runs a rule engine which accesses different sources of reputation (input) data from the reputors about an entity and evaluates an RO based on its declarative rule-based computational algorithms and contextual information available at the time of computation.
2. *Reputation Management Agents (RMAs)*: Act as a reputation trust center offering reputation management functionalities. An RMA manages the local RAAs providing control of their life-cycle in particular, and also ensuring goals such as fairness. It might act as a Reputation Service Provider (RSP) which aggregates reputations from the reputation scores of local RAAs. Based on the final calcu-

---

[m] *http://ruleml.org/PatientSupporter*

lated reputation, it might also perform actions, e.g. compute trust-worthiness, make automated decisions, or trigger reactions. It also manages the communication with the reputors, collecting data about entities from them, generates reputation data inputs for the reputation scoring, and distributes the data to the RAAs. It might also act as central point of communication for the real reputee entities (e.g., persons) giving them legitimate control over their reputation and allowing entities the governance of their reputations.

The agent-based approach for an online reputation management ensures efficient automation, semantic interpretability and interaction, openness in ownership, fine-grained privacy and security protection, and easy management of semantic reputation data on the Web.

6.5. *Semantic Complex Event Processing Agent Network*

The Event Processing Network (EPN) [KJP09] consists of Semantic Event Processing Agents (EPA) implemented as distributed Prova inference services which detect complex events using Prova's rule-based Semantic Complex Event Processing (SCEP) logic. [TP09]. The multi-agent approach allows for a highly-available distributed implementation with redundant Event-Calculus based state processing where events are processed concurrently in the EPN.

7. Related Work

Closely related to Rule Responder are agent architectures which directly use expressive rule languages and rule engines as basis for the agent behavior control. Using this kind of architecture basically requires that the rule base is properly connected with the agent's sensors and effectors in order to allow an agent to receive percepts and execute actions. To be able to exhibit reactive behavior and process incoming messages it is necessary that 1) incoming messages will trigger the execution of processing rules and 2) the external knowledge representation fits to the internal one or is mapped accordingly. Examples of this domain are e.g. JADE/Jess agent [Car07], Vivid Agents [SW00], OPAL Agents [WPN05], Jason [BWH07], and Emerald [KKB10]. In JADE/Jess an agent is constructed in a way that allows access to the production rule engine (JESS). Vivid agents are controlled systems whose state comprises the mental components of knowledge, perceptions, tasks, and intentions. Their behavior is represented by means of action and reaction rules, which follow the event-condition-action (ECA) paradigm and are underpinned by formal transition semantics for concurrent action planning. In Emerald reasoning engines are employed as reasoning services that are implemented as reasoner agents, which intercommunicate via FIPA-based communication protocols. In the OPAL agents various reasoning engines are employed as plug-in components of the agents which are deployed on the OPAL platform and intercommunicate via FIPA-based communication protocols. Jason is an extension of the AgentSpeak agent-oriented programming language used for rule-based programming of the behaviour of individual agents.

Another related category contains specialized rule engine based agent architectures, which have specialized rule engines for executing the agent logic. Nonetheless, the offered programming concepts have not been changed, i.e. the agent behavior specification is mainly done by programming traditional rules. Two typical representatives for this category are RC++ ^{WM03} and SOAR ^{LLR96}. The motivations behind those approaches are quite different. RC++ is an extension to the C++ programming language, which incorporates rules directly into the base language. It therefore realizes a conservative extension approach, which aims at a tight integration with the underlying procedural core language. The RC++ language is a general purpose language, but has been designed with a clear application focus in mind. It should facilitate the programming of game AI for the game console. In contrast, SOAR has been developed to be a general problem solver for arbitrary complex problems. Hence, the SOAR architecture has primarily been used for knowledge-rich intelligent agent applications. Examples include intelligent control, natural language understanding, human behavior experiments and simulation ^{WJ05}.

A third category of related rule-based agent architectures encompasses approaches that aim at introducing abstract mentalistic notions as agent programming language constructs. Each of these approaches proposes a specific concerted set of mental state components and introduces a language with specific types of rules to operate on these components (e.g. commitment rules, which operate on commitments). As a natural way of controlling the relation between different types of mental components and rules respectively, these approaches are not implemented as a general rule base, but use the specific rules only as part of an extended interpreter architecture. All approaches in this category are intended to be specific agent programming languages. Due to the rules operating directly on the mental state of the agent, these approaches are best suited for agents operating in dynamic environments, where quick reactions to environmental changes are advantageous. Such environments are common, e.g. for agents operating on mobile devices such as PDAs or cell phones and for controlling autonomous robots. Prominent representatives of this category are the AOP (AOP, cf. ^{Sho93})and the 3APL/2APL language families. 3APL (”An Abstract Agent Programming Language”) and its successor 2APL (”A Practical Agent Programming Language”) are developed at the University of Utrecht ^{HDBVDHCM99}. In addition to implementing agent applications, these languages are also used for teaching purposes. In contrast to 3APL/2APL, AOP languages are developed by different groups for different purposes including academic and commercial projects. Agent-0 ^{Sho93} and other AOP languages introduce additional constructs for making agent oriented programming more convenient. In the second generation of AOP, PLACA ^{Tho95} extends Agent-0 with planning features, while Agent-K ^{DE94} supports the KQML as a standardized agent communication language. Agent-K has been further extended to GOAL ^{BE96}, which incorporates planning similar to PLACA, but also on a multi-agent level (e.g. group goals, which comprise commitments of several agents). Successors of PLACA are RADL, which is the language of the commercial AgentBuilder toolkit [n] and AF-APL, belonging to the open source AgentFactory framework [RCO05]. Both languages try to advance the practical usability of the language, e.g., by providing convenient mechanisms for integration with Java.

8. Conclusion

Rule Responder is a framework for specifying virtual organizations as semantic multi-agent systems. The software is available open source in Sourceforge[o]. Characteristics of Rule Responder include

- the coverage of the distributed processing spectrum from Web Services to agents in one framework
- the recursive (holonic) modeling of a virtual organization of services and agents as a single agent,
- the use of ESBs, especially Mule, as a Semantic and Pragmatic Web infrastructure,
- the use of Semantic-Pragmatic Web rules as the main knowledge representation, complemented by ontologies,
- the introduction of PAs as human-assisting agents within a virtual organization, complementing the usual self-contained CAs,
- the design of a 'pluggable' agent-finding mechanism from role assignment to Semantic Service discovery.

The Rule Responder framework, with its increasing number of users and engines (Prova, OO jDREW, DR-Device, Euler, and Drools), is thus being proposed as a reference architecture for distributed rule-based knowledge representation and processing on the Semantic-Pragmatic Web.

9. Acknowledgement

The international Rule Responder initiative has greatly helped us with work leading to this article. In particular, we want ot thank Alexander Kozlenkov, Benjamin Craig, Taylor Osmun, Derek Smith, Omair Shafiq, Mahsa Kiani, Kia Teymourian, Rehab Alnemr, Irfan ul Haq, Nick Bassiliades, Stratos Kontopoulos, and Kalliopi Kravari.

References

APM10. Rehab Alnemr, Adrian Paschke, and Christoph Meinel. Enabling reputation interoperability through semantic technologies. In *ACM International Conference on Semantic Systems*. ACM, 2010.

---

[n] *http://www.agentbuilder.com/*

[o] http://mandarax.svn.sourceforge.net/viewvc/mandarax/PragmaticAgentWeb

BE96. Ciara Byrne and Peter Edwards. Refinement in agent groups. In *Proceedings of the Workshop on Adaption and Learning in Multi-Agent Systems*, IJCAI '95, pages 22–39, London, UK, 1996. Springer-Verlag.

BOC09. Harold Boley, Taylor Michael Osmun, and Benjamin Larry Craig. Social Semantic Rule Sharing and Querying in Wellness Communities. In Asunción Gómez-Pérez, Yong Yu, and Ying Ding, editors, *ASWC*, volume 5926 of *Lecture Notes in Computer Science*, pages 347–361. Springer, 2009.

Bol07. Harold Boley. Are Your Rules Online? Four Web Rule Essentials. In A. Paschke and Y. Biletskiy, editors, *Proc. Advances in Rule Interchange and Applications, International Symposium (RuleML-2007), Orlando, Florida*, volume 4824 of *LNCS*, pages 7–24. Springer, 2007.

BP07. Harold Boley and Adrian Paschke. Expert Querying and Redirection with Rule Responder. In Anna V. Zhdanova, Lyndon J. B. Nixon, Malgorzata Mochol, and John G. Breslin, editors, *FEWS*, volume 290 of *CEUR Workshop Proceedings*, pages 9–22. CEUR-WS.org, 2007.

BTW01. Harold Boley, Said Tabet, and Gerd Wagner. Design Rationale of RuleML: A Markup Language for Semantic Web Rules. In *Proc. Semantic Web Working Symposium (SWWS'01)*, pages 381–401. Stanford University, July/August 2001.

BWH07. Rafael H. Bordini, Michael Wooldridge, and Jomi Fred Hübner. *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*. John Wiley & Sons, 2007.

Car07. H. L. Cardoso. Integrating jade and jess, 2007.

CB08. Benjamin Larry Craig and Harold Boley. Personal Agents in the Rule Responder Architecture. In Nick Bassiliades, Guido Governatori, and Adrian Paschke, editors, *RuleML*, volume 5321 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2008.

Cra07. Benjamin Craig. The OO jDREW Engine of Rule Responder: Naf Hornlog RuleML Query Answering. In Adrian Paschke and Yevgen Biletskiy, editors, *RuleML-2007*, volume 4824 of *Lecture Notes in Computer Science*. Springer, 2007.

DE94. W. H. E. Davies and P. Edwards. Agent-K: An Integration of AOP and KQML. In T. Finin and Y. Labrou, editors, *Proceedings of the CIKM'94 Workshop on Intelligent Agents*, 1994.

DGKK98. P. Doherty, J. Gustafsson, L. Karlsson, and J. Kvarnstroem. Tal: Temporal action logics language specification and tutorial. *Linkoeping Electronic Articles in Computer and Information Science*, 3(015), 1998.

FLM97. Tim Finin, Yanis Labrou, and James Mayfield. *KQML as an agent communication language*. MIT Press, Cambridge, MA, USA, 1997.

HDBVDHCM99. Koen V. Hindriks, Frank S. De Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2:357–401, November 1999.

KJP09. Alexander Kozlenkov, David Jeffery, and Adrian Paschke. State management and concurrency in event processing. In *DEBS*, 2009.

KKB10. Kalliopi Kravari, Efstratios Kontopoulos, and Nick Bassiliades. Emerald: A multi-agent system for knowledge-based reasoning interoperability in the semantic web. In *SETN*, pages 173–182, 2010.

KS86. R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

LBBM06. Jie Li, Harold Boley, Virendrakumar C. Bhavsar, and Jing Mei. Expert Finding for eCollaboration Using FOAF with RuleML Rules. In *Montreal Conference of eTechnologies 2006*, pages 53–65, 2006.

LLR96. Jill Fain Lehman, John Laird, and Paul Rosenbloom. A gentle introduction to soar, an architecture for human cognition. In *In S. Sternberg & D. Scarborough (Eds), Invitation to Cognitive Science*. MIT Press, 1996.

MH69. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

Pas06a. Adrian Paschke. Eca-lp / eca-ruleml: A homogeneous event-condition-action logic programming language. *CoRR*, abs/cs/0609143, 2006.

Pas06b. Adrian Paschke. A typed hybrid description logic programming language with polymorphic order-sorted dl-typed unification for semantic web type systems. *CoRR*, abs/cs/0610006, 2006.

Pas07. A. Paschke. *Rule-Based Service Level Agreements - Knowledge Representation for Automated e-Contract, SLA and Policy Management*. Idea Verlag GmbH, Munich, 2007.

Pas08. Adrian Paschke. Rule responder hcls escience infrastructure. In *ICPW '08: Proceedings of the 3rd International Conference on the Pragmatic Web*, pages 59–67, New York, NY, USA, 2008. ACM.

PB08. Adrian Paschke and Martin Bichler. Knowledge representation concepts for automated sla management. *Decis. Support Syst.*, 46(1):187–205, 2008.

PK08. Adrian Paschke and Alexander Kozlenkov. A rule-based middleware for business process execution. In *Multikonferenz Wirtschaftsinformatik*, 2008.

RCO05. Robert Ross, Rem Collier, and G. OHare. Af-apl bridging principles and practice in agent oriented languages. In Rafael Bordini, Mehdi Dastani, Jrgen Dix, and Amal El Fallah Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3346 of *Lecture Notes in Computer Science*, pages 66–88. Springer Berlin / Heidelberg, 2005.

SG06. Geoff Sutcliffe and Randy Goebel, editors. *Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference, Melbourne Beach, Florida, USA, May 11-13, 2006*. AAAI Press, 2006.

Sho93. Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.

SW00. Michael Schroeder and Gerd Wagner. Vivid agents: Theory, architecture, and applications. *Applied Artificial Intelligence*, 14(7):645–675, 2000.

Tho95. S. Thomas. The PLACA agent programming language. In *Intelligent Agents*, pages 355–370. 1995.

TP09. Kia Teymourian and Adrian Paschke. Towards semantic event processing. In *DEBS*, 2009.

WCB01. M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well Conditioned, Scalable Internet Services. In *Proceedings of Eighteeth Symposium on Operating Systems (SOSP-18)*, Chateau Lake Louise, Canada, 2001.

WJ05. R. E. Wray and R. M. Jones. Considering Soar as an Agent Architecture. In Ron Sun, editor, *Cognition and Multi-agent Interaction*, pages 53–78.

Cambridge University Press, 2005.

WM03. I. Wright and J. Marshall. The execution kernel of rc++: Rete*, a faster rete with treat as a special case. *International Journal of Intelligent Games and Simulation*, 2, 2003.

Woo01. M. Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, 2001.

WPN05. Mengqiu Wang, Martin Purvis, and Mariusz Nowostawski. An internal agent architecture incorporating standard reasoning components and standards-based agent communication. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, IAT '05, pages 58–64, Washington, DC, USA, 2005. IEEE Computer Society.