Reaction RuleML 1.0 for Distributed Rule-Based Agents in Rule Responder

Adrian Paschke[1] and Harold Boley[2]

[1] Freie Universitaet Berlin, Germany
paschke AT inf.fu-berlin.de
[2] University of New Brunswick, Fredericton, NB, Canada
harold.boley AT unb.ca

Abstract. Rule Responder is a rule-based multi-agent framework in which agents run platform-specific rule engines as distributed inference services. They communicate with each other using Reaction RuleML as the common rule interchange format, e.g. for question answering or execution of mobile rule code in distributed problem solving, concurrent processing workflows and distributed event/action processing. In this paper we demonstrate the new capabilities of Reaction RuleML 1.0 for supporting the functionalities of Rule Responder such as knowledge interface declarations with signatures, modes, and scopes; distributed knowledge modules with static and dynamic scopes enabling imports and scoped reasoning within metadata-based scopes (closed constructive views) on the knowledge base; messaging reaction rules enabling conversation-scope based interactions between agents interchanging queries, answers, and rulebases; and evaluation and testing of interchanged knowledge bases with intended semantic profiles and self-validating test suites. We demonstrate these Reaction RuleML 1.0 capabilities with our proof-of-concept implementation Rule Responder agent architecture and Prova 3.0 rule engine.

1 Rule Responder

Rule Responder[1] [11, 14, 15] is a multi-agent system that supports distributed rule-based inference services on the Semantic-Pragmatic Web. It provides the infrastructure for using platform-specific rule engines for rule-based agents / inference services which can communicate with each other using Reaction RuleML as standardized rule interchange format.

RuleML is a knowledge representation language designed for the interchange of the major kinds of Web rules in an XML format that is uniform across various rule logics and platforms. It has broad coverage and is defined as an extensible family of sublanguages, whose modular system of schemas permits rule interchange with high precision. RuleML 1.0 encompasses both Deliberation RuleML 1.0 and Reaction RuleML 1.0[2] [1, 16, 10, 12, 13].

[1] http://responder.ruleml.org [14]
[2] http://reaction.ruleml.org

Reaction RuleML is a standardized rule markup language and semantic interchange format for reaction rules and rule-based event processing. Reaction rules include distributed Complex Event Processing (CEP), Knowledge Representation (KR) calculi, as well as Event-Condition-Action (ECA) rules, Production (CA) rules, and Trigger (EA) rules [13, 1, 16].

In this paper, we demonstrate the features of Reaction RuleML 1.0 for Rule Responder. Reaction RuleML defines a generic rule syntax distinguishing between metadata, interface, and implementation, enabling distributed and modularized rulebases and rules. It supports both programming-in-the-large with compositional import and message interchange mechanisms and programming-in-the-small with abstraction and scoping mechanisms. Semantic Profiles attach meaning to interchanged Reaction RuleML rulebases and messages and enable their semantic interpretation and interchange, e.g. in distributed rule-based agent system and rule-based Complex Event Processing (CEP) agent architectures [17].

As demonstration scenario we use a typical Semantic CEP scenario, namely the real-time monitoring of stock market events[19, 21, 20]. Multiple Rule Responder event processing agents monitor different stock market event streams which publish stock market ticker data, such as

```
{(Name, OPEL)(Price, 45)(Volume, 2000)(Time, 1) }
{(Name, SAP)(Price, 65)(Volume, 1000) (Time, 2)}
```

In contrast to standard CEP agents which apply syntactic event patterns, the Rule Responder agents support more expressive semantic event queries, such as "*stocks of companies, which have production facilities in Europe and produce products out of metal and have more than 10,000 employees*" and which require additional background knowledge bases (such as DBPedia), e.g.,

```
(OPEL, is_a, car_manufacturer),
(car_manufacturer, build, Cars),
(Cars, are_build_from, Metall),
(OPEL, has_production_facilities_in, Germany),
(Germany, is_in, Europe)
(OPEL, is_a, Major_corporation),
(Major_corporation, have, over_10,000_employees)
```

The paper is organizes as follows: Section 2 introduces semantic profiles which allow defining the intended evaluation semantics of Reaction RuleML knowledge. Section 3 distinguishes the knowledge interface from the knowledge implementation, which is the basis for describing the interfaces of distributed Rule Responder agents. Section 4 explains how Reaction RuleML supports modularization and distribution of knowledge locally, externally, and by import. Section 5 broadens the concept of scopes to the construction of dynamic views on the knowledge base by metadata based scopes and scoped reasoning. Section 6 uses this scoping mechanism for conversation based scopes which allow to maintain local (reasoning) context in messaging reaction rules with send and receive actions for interchanging Reaction RuleML based knowledge actions between Rule Responder agents. Finally, section 7 summaries the contribution of Reaction RuleML 1.0 for Rule Responder and their implementation in Prova 3[3].

[3] `http://prova.ws`

2 Dialects and Semantic Profiles

Dialects in Reaction RuleML provide a layer of general representation expressiveness by defining a dialect language, typically for a particular sort of reaction rules or a combination of different sorts. The main Reaction RuleML dialects with their core elements are:

- Derivation Reaction RuleML (if-then) - Time, Spatial, Interval, Situation (+ algebra operators)
- KR Reaction RuleML (if-then or on-if-do) - Happens(@type), Initiates, Terminates, Holds, fluent
- Production Reaction RuleML (if-do) - Assert, Retract, Update, Action
- ECA Reaction RuleML (on-if-do) - Event, Action, (+ event / action algebra operators)
- CEP Reaction RuleML (arbitrary combination of on, if, do) - Receive, Send, Message

Combinations of dialects are also possible, e.g., DR-PR Reaction RuleML, which combines derivation and production rules, or KR-CEP Reaction RuleML, which combines KR calculi with CEP reaction rules, enabling e.g. profiles with an interval-based Event Calculus semantics for complex (event/action) algebra operators [5, 18].

Semantic profiles in Reaction RuleML are used to define the *intended semantics* for knowledge interpretation (typically a model-theoretic semantics), reasoning (e.g., entailment regimes and proof-theoretic semantics), and for execution (e.g., operational semantics such as selection and consumption policies and windowing techniques in complex event processing). That is, they further detail the syntax and semantics of a dialect and provide necessary information about the intended semantics for Reaction RuleML knowledge representations as required for interchange, translation, inference, and verification and validation.

A dialect has a default semantic profile defining the *default semantics*, i.e., the semantics which by default is used for interpretation. Deviating semantic profiles (`<Profile>`) can be specified (`<evaluation>`) on all formulas and terms in Reaction RuleML giving them an interpretation and execution different from the default semantics.

Definition 1. *(**Semantic Profile**) A semantic profile, $SP = \langle S_{SP}, \Sigma_{SP}, I_{SP}, \Phi_{SP}, \tau_{SP} \rangle$, (partially) defines a profile signature S_{SP}, a language Σ_{SP}, an interpretation I_{SP}, a domain-independent theory Φ_{SP}, and a semantics-preserving translation function $\tau_{SP}(\cdot)$ which translates from Reaction RuleML to the profile's language / signature (and vice versa, with the inverse function τ_{SP}^{-1}).*

Semantic Profiles can be defined internally within a Reaction RuleML document (`<Profile>`) or externally. External semantic profiles can be referenced by their profile name (`@type`) and imported by their resource identifier (`@iri`). Their specification can be given in any XML format (`<content>`), including RuleML formulas (`<formula>`), as well as other formal and textual languages (which need not be directly machine processable). For non-Reaction RuleML profiles a semantics-preserving translation function τ needs be defined in order to allow interpretations of Reaction RuleML knowledge bases with the profile's semantics.

Multiple alternative semantic profiles are allowed with or without a priority ordering and their scope can be specified. For instance the following XML snippet uses two alternative profiles and gives the first profile preference by ordering them (`@index`).

```
<evaluation index="1">   <!-- WFS semantic profile -->
  <Profile type="ruleml:Well-Founded-Semantics" direction="backward"/>
</evaluation>
<evaluation index="2">   <!-- alternative ASS semantic profile -->
  <Profile type="ruleml:Answer-Set-Semantics"/>
</evaluation>
```

Semantic profiles might define further specialized or deviating structures (e.g., module composition with modular join semantics etc.), intended models (e.g., in terms of entailment regimes) and axioms and propositions (e.g., domain independent meta axioms of a theory, e.g. for calculi such as event calculus, situation calculus), as well as proof-theories and properties of operational semantics (e.g., process semantics and protocols, windowing techniques, selection and consumption policies in complex event processing and actions), etc. Also, they might specialize the language of a dialect, e.g., by limiting the dialect's signature to subsignatures.

Definition 2. *(**Subsignature**) A signature S_1 is a subsignature of S_2, i.e., $S_1 \subseteq S_2$ iff S_1 is a signature which consist only of symbols from S_2 without changing their sort and arity.*

For instance, the following example defines the signature of a global predicate "*planned*" as subsignature of the already existing `<Happens>` predicate in the KR dialect of Reaction RuleML, which takes an event as first argument and a time as second argument, with the mode attribute defining it as input predicate.

```
<signature>
  <Atom type="ruleml:Happens" arity="2" mode="+">
    <Rel scope="global">planned</Rel>
    <Var type="ruleml:Event" mode="+" scope="local"/>
    <Var type="ruleml:Time" mode="+" scope="local"/>
  </Atom>
</signature>
```

These explicitly defined signatures are introduced as further specialized sorts into the multi-sorted signature of a Reaction RuleML dialect and they are interpreted by the intended semantic profile, which defines a multi-sorted interpretation for the sort symbols. For further details about the core multi-sorted signatures and structures of Reaction RuleML see [10].

The following example (in Prova 3 syntax) filters received stock market ticks and sends the filtered events as new happens facts to another event processing agent, called "epa1".

```
Filter for stocks starting with "A" and price > 100
rcvMult(SID,stream,"S&P500", inform, tick(S,P,T)) :-
  filter(S,P), sendMsg(SID2,esb,"epa1", inform, happens(tick(S,P),T).
filter(Symbol,Price) :-
 Price > 100, Symbol = "A.*".
```

The transformed happens facts are interchanged to agent "epa1" as Reaction RuleML messages, e.g., specifying a reified event calculus profile as intended semantics.

```
<evaluation index="1">   <!-- Event Calculus semantic profile -->
  <Profile type="ruleml:ReifiedClassicalEventCalculus"/>
</evaluation>
```

3 Knowledge Interface Definition

A knowledge representation element in Reaction RuleML consists of the representational **knowledge object**, such as rulebases, rules, facts, queries, events, etc., and optional additional **meta knowledge**.

The meta knowledge comprises descriptive **metadata** (`<meta>`) and the **knowledge interface**, which contains information about the knowledge scope (`<scope>`), guard constraints (`<guard>`), intended semantics (`<evaluation>`), explicit signatures (`<signature>`), qualifying metadata (`<qualification>`), and quantifiers (`<quantification>`).

Moreover, several knowledge formulas can have further specialized meta knowledge, such as a truth/uncertainty degree (`<degree>`) for atomic formulas (`<Atom>`) and equations (`<Equal>`), conversation identifiers (`<cid>`), protocols (`<protocol>`), sender/recevier agents (`<sender>, <receiver>`) for messages (`<Message>`), etc. Several meta knowledge attributes specify additional information, e.g. about sort (`@type`), arity (`@arity`), cardinality of set values (`@card`, `@maxCard`, `@minCard`), relative weight (`@weight`), default quantification closure (`@closure`), inference direction (`@direction`), ordering (`@index`), internationalized (remote) resource locator (`@iri`), global node identifier (`@node`), internal key and key reference (`@key`, `@keyref`), input-output mode declaration (`@mode`), material implication (`@material`), equality orientation (`@oriented`), interpretation semantics of relations and functions (`@per`), prefix and vocabulary definition for "webized" IRI mappings (`@prefix`, `@vocab`), processing/execution safety (`@safety`), reasoning and execution style (`@style`), and indeterminism/determinism of functions and operators (`@val`).

The `knowledge implementation` is an instance (a knowledge object) of the knowledge interface, i.e. it needs to be well-formed according to the signature and it needs to be interpreted with the intended structures (Semantic Profiles) in the scope (quantification and qualification scope) defined by the interface.

The following example shows this distinction into metadata, interface, and implementation for a rule.

```
<Rule @key @keyref @style>
     <!-- descriptive rule metadata -->
     <meta>   <!- descriptive metadata of the rule -->        </meta>

     <!-- rule interface -->
     <scope>   <!- scope of the rule e.g. a rule module -->   </scope>
     <evaluation>    <!-- intended semantics -->           </evaluation>
     <signature> <! rule signature -->                        </signature>
     <qualification> <!- e.g. qualifying rule metadata, e.g.
                          priorities, validity, strategy -->   </qualification>
     <quantification>  <!-- quantifying rule declarations,
                           e.g. variable bindings -->   </quantification>

     <!-- rule implementation -->
     <oid>  <! object identifier -->               </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>
```

– The `<meta>` roles, contain the descriptive metadata, which is used for annotations describing the knowledge object. By default all knowledge is contextually anno-

tated by metadata about their source (@src([$Locator$])) and their name (@$label$([OID])), with *Locator* being the module's source location in which the knowledge object is implemented/defined and OID being the implicitly (i.e., automatically created) or explicitly defined object identifier.
- The `<scope>` role defines subsets of the universe as domain of discourse in which interpretation takes place. That is, from an operational point of view they define a static or dynamic constructive view on the knowledge base in which scoped reasoning takes place. The Reaction RuleML vocabulary predefines the scopes "*global*" (globally visible) "*local*" (visible with local interpretation) and "*private*" (hidden and not visible outside of the module). Reaction RuleML dialects might introduce further scopes, such as, e.g., "*supremum*" and "$infimum$", which expand the scope of nested submodules to its least upper bound or greatest lower bound. Further, named **metadata scopes** can be defined (in the `scope` role) and used as scoped submodule in scoped reasoning.
- The `<evaluation>` roles are used to specify semantic profiles, which define the intended semantics for knowledge interpretation (typically a model-theoretic semantics), reasoning (typically entailment regimes and proof-theoretic semantics), and for execution (e.g., operational semantics such as selection and consumption policies and windowing techniques in complex event processing). This includes, e.g., semantic properties and assumptions such as closed world, open world, closed (positively, negatively closed), etc.
- The `<signature>` role explicitly defines signatures, i.e., it introduces (specialized) signature definitions in the core multi-sorted signature of the used Reaction RuleML dialect. With the `@type` attribute locally defined signature sorts (e.g. frame types, event patterns etc.) as well as externally defined sorts from external (possibly order-sorted) type systems can be introduced as new sorts in the multi-sorted signature. Modes (`@mode`) further partition the universe into subsets having a different mode. Reaction RuleML predefines the modes (`@mode`) "+" (input mode), "−" (output mode) and "?" (open mode, i.e., input or output).
- The `<qualification>` role defines qualifying metadata. In contrast to descriptive metadata (meta), qualifying metadata has an impact on the interpretation, e.g. it is used for knowledge prioritization (e.g., conflict resolution strategies, defeasible reasoning etc.), or for defining validity times (e.g., for windowing techniques in event processing etc.).
- The `<quantification>` role explicitly defines the quantifiers. Note, there is a default quantifier scope assumed in a dialect; typically a universal closure, so that formulas are universally closed by default.

While these optional meta roles allow explicit definitions of meta knowledge, corresponding attributes on the knowledge formulas can point to these definitions. For instance, the `@scope` attribute can use the predefined terms[4] "*global*", "*local*" and "*private*", as well as scope names defined in the `<scope>` role. In following example (in Prova 3 syntax) the first public rule receives stock market ticks from the event stream "S&P500" and the second private selection rule compares the price with ticker information from other streams in the same event group in order to detect suspicious price information.

```
% Select stock ticker events from stream "S&P500"
```

[4] by using `@vocab`, an automated mapping of terms into IRIs is performed, i.e., the predefined terms are mapped into IRIs of the Reaction RuleML vocabulary.

```
% Each received event starts a new subconversation (CID) which further processes the selected
% event (select)
@group(g1) rcvMult(CID,stream,"S&P500", inform, tick(S,P,T)) :-
 @scope(private) select(CID,tick(S,P,T)).
% Indefinitely (count=-1) receive further ticker events from other streams that follow the
% previous selected event in event processing group (group=g1). If the price differs for
% the same stock at the same time [T1=T2, P1!=P2] then ...
@scope(private)
select(CID,tick(S,P1,T1)) :-
@group(g1) @count(-1)
rcvMsg(CID,stream, StreamID ,inform, tick(S,P2,T2)) [T1=T2, P1!=P2]
    println (["Suspicious:",StreamID, tick(S,P2,T2)]," ").
```

4 Knowledge Modularization and Distribution

Reaction RuleML supports knowledge modularization and distribution. A syntactic way to distribute knowledge locally within a KB is by separating the representation of a knowledge formula into several syntactic parts and connecting and conjoining them syntactically by key-keyref pairs (`@key`, `@keyref`). A key is a local ("webized" by using `@prefix` and `@vocab`) identifier, with a unique name assumption (UNA), which can be defined as meta knowledge on any Reaction RuleML language element. A key reference is a syntactic local reference (within a KB) using the key as locator to connect and conjoin the key element with the key reference element. Multiple references to a key element are possible ($1:m$ as well as $n:m$ by defining both key and keyref on pairwise conjoined elements). The resulting combined syntax elements need to be well-formed (according to their signature definitions) to allow meaningful interpretations, i.e. key-keyref pairs need to be on similar syntactic elements and for each key reference a matching unique key needs to be defined in a KB. A typical application of the key-keyref mechanism is the separation of the knowledge interface with signatures from the knowledge implementation, so that both can be represented and reused independently. For instance, the following example shows the separated implementation of a rule "$ruleimpl1$" which is referencing the rule interface "$ruleinterface1$".

```
<Rule key="ruleinterface1">
   <evaluation><Profile> ... </Profile></evaluation>
   <signature> ... </signature>
   ...
</Rule>
...
<Rule keyref="rulreinterface1"  key="ruleimpl1 ">
    <if>  ...   </if>
    <do> ... </do>
</Rule>
```

This enables, e.g., template definitions (e.g., abstracted signature patterns, knowledge templates, event pattern definitions, etc.), modularization and information hiding, e.g. by publishing the interface in a document distributed from the document with the (possibly private) implementation[5].

Furthermore, with the `@iri` attribute also remote resources can be referenced. For instance, in the following example an RDFS entailment regime is referenced as intended semantic profile for the order-sorted interpretation of external sorts defined in external RDFS ontologies (taxonomies).

[5] With XML Inclusion (`XInclude`) such distributed documents can be syntactically included into one KB enabling local key intra-references within it.

```
<evaluation><Profile type="rif:RDFS iri="http://www.w3.org/ns/entailment/RDFS"/></evaluation>
```

This profile can be used, e.g. for the type reasoning in the following example rule (in Prova syntax), which defines a semantic event query with variables typed by background knowledge bases (ontologies)[6].

```
rcvMult(SID,stream,S&P500, inform,
  tick(Name^^car:Major_corporation,P^^currency:Dollar,T^^time:Timepoint)) :- ...
```

Rule-based Data Access (RBDA) with optimizing techniques, such as enrichment, can be used for efficient processing. For instance, the following example defines a rule-based data access rule which selects with the SPARQL query built-in of Prova [7] all luxury car manufacturers from DBPedia.

```
luxuryCar(Manufacturer,Name,Car) :-
 Query="SELECT ?manufacturer ?name ?car   % SPARQL RDF Query
WHERE {?car <http://purl.org/dc/terms/subject>
         <http://dbpedia.org/resource/Category:Luxury_vehicles> .
     ?car foaf:name ?name .
     ?car dbo:manufacturer ?man .
     ?man foaf:name ?manufacturer. } ORDER by ?manufacturer ?name,
sparql_select(Query,manufacturer(Manufacturer),name(Name),car(Car)).
```

Rulebase formulas (`<Rulebase>`) introduce a (possibly nested) structuring of groups of knowledge formulas, called **modules**. External (`<RuleML>`) documents and messages (`<Message>`) can be consulted/imported (`<Consult>`) or received (`<Receive>`). They are treated as submodules in the importing KB.

Definition 3. *(**Import**) A document KB' is said to be an import to a document KB, if it is directly imported into KB (or it is imported into another document, which is directly imported into KB). An imported document KB' becomes a module of the KB.*

For instance, the following example consults (imports) the rule interface which has been published in a separated Reaction RuleML document.

```
<Consult iri="http://reaction.ruleml.org/1.0/exa/dr/DistributedDerivationRuleInterface.rrml"/>
```

The importing Reaction RuleML KB (i.e., `<RuleML>` document) is the super module of all modules. All asserted, imported, and received rulebases are submodules of this KB module.

Definition 4. *(**Module and Submodule**) A module Φ is a tuple $\langle \overline{@\phi}, \overline{\phi} \rangle$, where $\overline{\phi}$ is an ordered or unordered finite set of knowledge formulas $\phi_i \in \overline{\phi}$ (without or with duplicates) and $\overline{@\phi}$ is an ordered or unordered finite set of meta knowledge formulas $@\phi_i \in \overline{@\phi}$, called the **module interface**. A module Ψ is a submodule of Φ if $\Psi \subseteq \Phi$.*

Modes partition the symbols in the universe used for formulas (predicates, functions, and terms) into input, output, and open symbols. Also, **scopes** partition formulas (and terms) into global, local, and private symbols. The set of input formulas In and the set of output formulas Out are visible and can be imported and accessed by other

[6] RuleML and Prova support external types, such as object-oriented Java class hierarchies and ontologies [4]

[7] Prova has various built-ins for rule-based data access such as Java object access, file access, XML (DOM), SQL, RDF triples, XQuery, SPARQL

modules. Accordingly, their scope must be either *global* or *local*. The set of formulas with *private* scope are hidden internal formulas $Priv$ and hence not accessible by other modules. This is used for defining the semantics of imports (`<Consult>`, `XInclude`) and the composition semantics of modules.

Definition 5. *(**Modes and Scopes**) Let M be a module consisting of a tuple $\langle F, I, O, G, L, P\rangle$, where F is a finite set of all knowledge formulas in M; I, O are pairwise disjoint sets of input and output formulas, i.e., formulas with input or output mode; G, L, and P are disjoint sets of formulas with global, local, and private scope. The function $signature(F)$ gives the signature of F, $mode(F)$ gives the mode, and $scope(F)$ gives the scope. Let R be the set of rules in M, then $signature_{atom}(R)$ gives the atoms (atomic formulas) of a rule R, where*

- *$signature_{atom}(on)$, $signature_{atom}(if)$, $signature_{atom}(after)$ are in I, i.e. the parts of R which consist of input atoms*
- *$signature_{atom}(then)$, $signature_{atom}(else)$, $signature_{atom}(do)$ and $signature_{atom}(elseDo)$ are in O, i.e. output atoms which are not in I.*

Semantic profiles can define the composition semantics of modules.

The composition semantics must respect the declared modes and scopes. While global and local scopes are visible and hence can be used in input and output modes, a private scope is only visible within the module in which it is defined. For instance, the composition of imported modules removes the output atoms from the set of input atoms and enforces that KB' and KB are mutual independent and that private atoms from one module are not part of the signature of the other module in which they are not visible. Private and local symbols might be defined and interpreted differently in two imported modules, but their interpretation must coincide for global symbols.

Definition 6. *(**Module Composition**) Let KB' and KB be two modules; their composition is the union $KB' \sqcup KB = \langle F_{KB'} \cup F_{KB}, (I_{KB'} \cup I_{KB})\ (O_{KB'} \cup O_{KB}), O_{KB'} \cup O_{KB}, G_{KB'} \cup G_{KB}, L_{KB'} \cup L_{KB}, P_{KB'} \cup P_{KB}\rangle$, where $signature(KB') \cap P_1 = \emptyset$ and $signature(KB) \cap P_2 = \emptyset$ and there are no (positive) cyclic dependencies[8] in $F_{KB'} \cup F_{KB}$ through loops between input and output atoms.*

That is, semantic profiles define the intended interpretations for composed multi-module KBs. Semantic profiles might also specify syntactic translation and transformation mapping, e.g., to consistently map all imported local symbols into the local symbols of the importing KB or to do program transform such as renaming output formulas to guarantee compositionality. Furthermore, they can define concrete module composition semantics, e.g. as semantic join of models, as well as a mechanism to avoid cyclic imports[9].

Definition 7. *(**Modular Semantic Multi-Structure**) A modular semantic multi-structure $\overline{M} = \langle M_{KB_0}, M_{KB_1}, M_{KB_2}, ..\rangle$ is a set of semantic structures such that M_{KB} is the semantic structure of the importing KB and M_{KB_k} is a set of semantic structures*

[8] One approach to detect mutual positive or negative dependencies is by adding extra meta information in the models.

[9] see the Semantic Profiles for modular Reaction RuleML knowledge bases and the OntoMaven/RuleMaven dependency analysis and importation resolution algorithm for imports from distributed KB repositories.

of the imported modules. The semantic structures M_{KB} and all structures M_{KB_k} are required to coincide in the mappings of global symbols in all semantic structures. But they might differ for local and private symbols in their interpretation using the module scope to constrain and close the domain of discourse for deviating local interpretations in each M_{KB_k}.

A semantic profile can specify how to do the expansion of the modular semantic multi-structure. The default is that the semantics of imported modules expands to the semantics of the importing KB. But, other semantics can be define in a profile, e.g. as a conservative composition using renaming output transformations on the output formulas in the module composition and a semantic outer join operator for the joint interpretation.

Definition 8. ***(Join)*** *Let KB' and KB be two modules and $\overline{I'}$ and $\overline{I}$ be their sets of interpretations; then the natural outer join $\overline{I'} \bowtie \overline{I} = \{I' \cap signature(KB) = I \cap signature(KB')andI \cup I'\}$, where $I' \in \overline{I'}$ and $I \in \overline{I}$.*

The following example (in Prova) shows the rule of a manager agent which reads a Prova script from a file and uploads it to a contractor agent. The contractor consults and evaluates the receive mobile code.

```
% Manager
upload_mobile_code(Remote,File) :
    Writer = java.io.StringWriter(),
    fopen(File,Reader),
    copy(Reader,Writer),
    Text = Writer.toString(),
    SB = StringBuffer(Text),
    sendMsg(XID,esb,Remote,eval,consult(SB)).
% Service (Contractor)
rcvMsg(XID,esb,Sender,eval,[Predicate|Args]):- derive([Predicate|Args]).
```

5 Scoped Reasoning

A particular important feature of Reaction RuleML 1.0 feature for Rule Responder agents is that it allows constructing views dynamically on the KB. These views are defined by **metadata scopes** [14, 8] in which scoped reasoning can be performed.

Definition 9. ***(Metadata Scope)*** *Let KB be a KB. A metadata scope $KB_@$ (aka constructive view on KB), which is defined by one or more closed metadata (constrained) formulas $\{@(L_1), .., @(L_n)\}$ on the KB, is a submodule $KB_@ \subseteq KB$, where for every formula ϕ in $KB_@$ their metadata $@(\phi)$ satisfies the metadata constraints defined by the metadata scope. The scope's subsignature $S_@$ is said to be the* ***scoped domain of discourse****.*

Scoped reasoning can be performed on such metadata scopes (aka **constructive views** on the KB) by defining **scoped literals** in conditions, queries, and event patterns. Scope literals are interpreted in the scoped domain of discourse and by default have the scopes' closure[10]. The scope definition of a scope literal might contain variables. In addition to scopes, Reaction RuleML supports **guards** which act as additional

[10] like for module imports in the large semantic profiles can define different closure semantics

pre-conditional constraints on the literal. The following example defines a scoring rule which selects ticker events with a score value (scope) over 2 (guard). Accordingly, only the second ticker event with a score value over 2 is further analyzed.

```
% event instances with metadata
@score(1) @src(stream1) @label(e1) tick("Opel",10,t1).
@score(3) @src(stream2) @label(e2) tick("Opel",15,t1).

happens(tick(S,P),T):-
   @score(Value) tick(S,P,T) [Value>2].
```

By default, the scope of relations and functions is global and their arguments' scope is local. A global scope corresponds to a metadata scope defined over all knowledge qualified with the source of the KB (@*source*([*Locator*])), and the local scope corresponds to the metadata scope defined over all knowledge qualified with the name of the module (@*label*([*OID*])). The mode of formulas when used as conditions, constraints, queries, and event patterns is "+" (input), and the mode of conclusions, answers, and active actions is "−" (output); with a corresponding mode for their constant arguments, and by default for variables, the mode is "?" (open). The default quantification scope is universal (`<Forall>`). There is a nested submodule inheritance, i.e., meta knowledge defined on outer modules is automatically inherited to the inner modules.

In the following example defines a rating of events from only trusted sources.

```
% stream1 is trusted but stream2 is not, so one solution is found: X=e1
@src(stream1) event(e1).
@src(stream2) event(e2).

%only event from stream1 are trusted
@scope(private) @label(trust_db) trusted(stream1).

ratedEvent(X):-
  @src(Source) %scoped reasoning on @src
  event(X) [trusted(Source)]. %guard on trusted sources

:-solve(ratedEvent(X)). % => X=e1 (but not e2)
```

6 Messaging

Reaction RuleML supports actions for sending (`<Send>`) and receiving (`<Receive>`) knowledge via messages (`<Message>`) in **messaging reaction rules** (rule `@style="messaging"`). Messages interchange Reaction RuleML documents as their payload between agents (`<Agent>`), which are rule-based agents (aka inference services). A Message element that provides the syntax for inbound and outbound messages / notifications. Besides having the typical meta knowledge, a message consists of

- an `<oid>` (message object identifier) and a `<cid>` which is the conversation identifier (enabling also long-running asynchronous conversations and sub-conversations with new conversation identifiers),
- an optional `<protocol>`: protocol definition (e.g. high-level negotiation and coordination protocols, agent protocols and operational transport protocols)
- an optional `<sender>` and `<receiver>`: denotes the sender and the target of the message,
- a `directive`: pragmatic context defining the pragmatic interaction and interpretation context for the message payload, e.g. FIPA Agent Communication Language primitives such as "acl:query-ref",

- an optional `@type` defines the type of the message and an optional `@mode` attribute distinguishes "inbound" from "outbound" communication,
- a `payload` transporting any valid `<RuleML>` knowledge document (enclosing, e.g., queries (`<Query>`), answers (`<Answer>`), imports, and updates (`<Consult>`, `<Assert>`, `<Retract>`, `<Update>`), as well as general actions (`<Action>`)) or arbitrary XML `<content>`.

The following example shows the typical template of a message:

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

The knowledge of received RuleML documents can be used in the messaging reaction rules of the receiving agent. An important difference to "standard" imports, as described in the previous section, is that these knowledge updates are private to the **conversation scope** of the message interaction which takes place in the **execution scope** of the messaging reaction rules. A typical application of this conversation-based interactions is distributed question-answering (Q&A) between distributed agents (i.e., agents providing query interfaces to their KBs), where the send and receive actions in messaging reaction rules act as queries and answers to the external agents' KBs. For instance, the following messaging reaction rule starts two conversations, "$xid1$" and "$xid2$", sending two queries. The serial execution initially waits for the answers from the second conversation and then, after proving some conditions (e.g., conditions defined on the bound variables of the received answers), the execution waits for the answers to the first query in the first conversation.

```
<Rule style="messaging">
  ...
  <do><Send>
    <Message>
      <cid><Var>xid1</Var></cid>
      ... <payload> ... query1 ... </payload>
    </Message>
  </Send></do>
  <do><Send>
     <Message>
       <cid><Var>xid2</Var></cid>
       ... <payload> ... query2 ... </payload>
     </Message>
  </Send></do>
  <on><Receive>
    <Message>
      <cid><Var>xid2</Var></cid>
      ..<payload> ... answer2 ... </payload>
    </Message>
  </Receive></on>
  <if> ... conditions ... </if>
  <on><Receive>
    <Message>
      <cid><Var>xid2</Var></cid>
      ..<payload> ... answer1 ... </payload>
    </Message>
  </Receive></on>
   ...
</Rule>
```

Answers are given in terms of solved formulas, e.g. as a rulebase (`<Rulebase>`) that contains 'solved' equations with the variable bindings.

```
<Rulebase>
   <Equal><Var>x</Var><Ind>a</Ind></Equal>
   <Equal><Var>y</Var><Ind>b</Ind></Equal>
   <Equal><Var>z</Var><Ind>c</Ind></Equal>
 </Rulebase>
```

Alternatively, they are given one by one as ground literals (`<Atom>`) matching the sent query.

```
<Atom>
   <Rel>p</Rel>
   <Ind>a</Ind>
   <Ind>b</Ind>
   <Ind>c</Ind>
</Atom>
```

The semantics of sent queries interprets them as (sub-)goals which are proven by the external agent's knowledge and the variable bindings from the received answers are used to continue the local proof logic in the serial execution of the messaging reaction rule.

The following Prova example defines an event composition workflow, where first an event "A" is received which forks the process into two alternative branches for event "B" and "C".

```
rcvMsg(XID,Process,From,event,["A"]) :-
  fork_b_c(XID, Process).

fork_b_c(XID, Process) :-
 @group(p1) rcvMsg(XID,Process,From,event,["B"]),  .

fork_b_c(XID, Process) :-
 @group(p1) rcvMsg(XID,Process,From,event,["C"]),  .

fork_b_c(XID, Process) :-
  % OR reaction group "p1" waits for either of the two event message handlers "B" or "C"
  % and terminates the alternative reaction if one arrives
 @or(p1) rcvMsg(XID,Process,From,or,_).
```

Such reaction groups establish an additional **event processing scope** which is used to manage the event processing flow. This can be used, for instance, to define relative timer events in a reaction group within a time window, e.g., for the accumulation of events over a time window as in the following Prova example.

```
% This reaction operates indefinitely. When the timer elapses (after 25 ms),
% the group by map Counter is sent as part of the aggregation event and consumed in an
% or group, and the timer is reset back to the second argument of @timer.

groupby_rate() :-
  Counter = ws.prova.eventing.MapCounter(), % Aggr. Obj.
  @group(g1) @timer(25,25,Counter), % timer every 25 ms
  rcvMsg(XID,stream,From,inform,tick(S,P,T)) % event
   [IM=T,Counter.incrementAt(IM)]. % aggr. operation

groupby_rate() :-
% receive the aggregation counter in the or reaction
    @or(g1) rcvMsg(XID,self,From,or,[Counter]),
    ... % consume the Counter aggreation object.
```

Another important aspect in this distributed interaction is the interface declaration of knowledge base, in particular the signatures and their scope visibility which define which knowledge can be queried from external agents. Furthermore, the agent conversations might follow certain defined protocols (`<Protocol>`) and the knowledge interpretation might be given an additional pragmatic context (`<directive>`).

For the correct semantic interpretation, the intended semantic profiles can be interchanged together with **test suites**, which are special knowledge bases (`<TestSuite>`) with a test assertion base (typically ground facts), and test items (`<TestItem>`), consisting of test queries and predefined expected answers. [3, 8, 7, 2, 6] The following example shows a typical template for a test.

```
<Test>
 <TestSuite>
  <!-- semantic profiles which should be used for the test suite -->
  <evaluation><Profile></Profile></evaluation>
  <!-- test assertion base, such as ground test facts -->
  <testbase><Assert></Assert></testbase>
  <!-- one particular test item of a test suite -->
  <TestItem>
    <!-- test query -->
    <act><Query></Query></act>
    <!-- expected result -->
    <expectedResult>
     <Answer>
      <degree><Data>1</Data></degree> <!-- expected "1" = query succeeds -->
      <!-- resulting query variable bindings  -->
      <Equal></Equal>
      <Equal></Equal>
     </Answer>
    </expectedResult>
   </TestItem>
 </TestSuite>
</Test>
```

7 Conclusion - Reaction RuleML for Reaction Rules

In this paper we have described and exemplified several important features of Reaction RuleML 1.0 which provide support for Rule Responder.

- **Knowledge interface** declarations can be used to define signatures/patterns, intended semantic profiles, scopes, qualifications and quantifications of knowledge implementations with possible separation and distribution of the interface from its implementations by key-keyref references.
- **Modules** with declarations of **scopes**, **modes**, and **profiles** distinguish global, local, and private knowledge with possibly varying interpretations and visibility for external agents.
- **Knowledge imports** allow consultation of external knowledge as modules of the importing KB with module composition and interpreting (join) semantics defined by semantic profiles.
- **Scoped reasoning** can be used to close off the domain of discourse to a particular scope, including dynamic **metadata-based scopes** which act as constructive views on the (modular) knowledge base. [14, 8]
- **Messaging reaction rules** are used for serial execution of (conditional) **send and receive actions** which interchange knowledge between Rule Responder agents, such as queries, answers, and rulebases, via Reaction RuleML messages within

protocol conversation scopes and serial execution scopes of the messaging reaction rules in which the conversations take place.
- **Semantic profiles** and **test suites** are optionally interchanged together with the knowledge in order to test the Reaction RuleML knowledge with the intended semantics against pre-defined test items (test cases) leading to **self-validating rule bases**. [3, 8, 7, 2, 6]

The implementation of these advanced features of Reaction RuleML 1.0 and Rule Responder is demonstrated in the Prova 3.0 rule engine with use cases for **Q&A answering** on top of rule-based data access to Linked Open Data (LOD) knowledge bases such as DBPedia and design pattern implementations for Semantic Complex Event Processing (SCEP) functionalities [17, 9].

References

1. H. Boley, A. Paschke, and O. Shafiq. RuleML 1.0: The Overarching Specification of Web Rules. In *Semantic Web Rules - International Symposium, RuleML 2010, Washington, DC, USA, October 21-23, 2010. Proceedings*, volume 6403 of *Lecture Notes in Computer Science*, pages 162–178. Springer, 2010.
2. Jens Dietrich and Adrian Paschke. On the Test-Driven Development and Validation of Business Rules. In *Information Systems Technology and its Applications, 4th International Conference ISTA'2005, 23-25 May, 2005, Palmerston North, New Zealand*, volume 63 of *LNI*, pages 31–48. GI, 2005.
3. Adrian Paschke. The ContractLog Approach Towards Test-driven Verification and Validation of Rule Bases - A Homogeneous Integration of Test Cases and Integrity Constraints into Evolving Logic Programs and Rule Markup Languages (RuleML) . *International Journal of Interoperability in Business Information Systems*, 10, 2005.
4. Adrian Paschke. A Typed Hybrid Description Logic Programming Language with Polymorphic Order-Sorted DL-Typed Unification for Semantic Web Type Systems. In *OWLED*, 2006.
5. Adrian Paschke. ECA-RuleML: An Approach combining ECA Rules with temporal interval-based KR Event/Action Logics and Transactional Update Logics. *CoRR*, abs/cs/0610167, 2006.
6. Adrian Paschke. On Self-Validating Rule Bases. In *Proceedings of 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), Athens, Georgia, USA*, 2006.
7. Adrian Paschke. Verification, Validation, Integrity of Rule Based Policies and Contracts in the Semantic Web. In *2nd International Semantic Web Policy Workshop (SWPW'06), Nov. 5-9, 2006, Athens, GA, USA*, volume 207 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
8. Adrian Paschke. *Rule based service level agreements: RBSLA; knowledge representation for automated e-contract, SLA and policy management.* Idea Verlag GmbH, 2007.
9. Adrian Paschke. Semantic Complex Event Processing. http://www.slideshare.net/swadpasc/dem-aal-semanticceppaschke, May 2013. Tutorial at Dem@Care Summer School on Ambient Assisted Living, 16-20 September 2013, Chania, Crete, Greece, `http://www.slideshare.net/swadpasc/dem-aal-semanticceppaschke`.

10. Adrian Paschke. Reaction RuleML 1.0 for Rules, Events and Actions in Semantic Complex Event Processing. In *8th International Web Rule Symposium, Prague, Czech Republic, August 18-20, RuleML 2014 Proceedings*, Lecture Notes in Computer Science. Springer, 2014.
11. Adrian Paschke and Harold Boley. Rule Responder, October 2007. RuleML project, `http://responder.ruleml.org/`.
12. Adrian Paschke and Harold Boley. Rule Markup Languages and Semantic Web Rule Languages. In Adrian Giurca, Dragan Gasevic, and Kuldar Taveter, editors, *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 1–24. IGI Publishing, May 2009.
13. Adrian Paschke and Harold Boley. Rules Capturing Events and Reactivity. In Adrian Giurca, Dragan Gasevic, and Kuldar Taveter, editors, *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, pages 215–252. IGI Publishing, May 2009.
14. Adrian Paschke and Harold Boley. Rule Responder: Rule-Based Agents for the Semantic-Pragmatic Web. *International Journal on Artificial Intelligence Tools*, 20(6):1043–1081, 2011.
15. Adrian Paschke, Harold Boley, Alexander Kozlenkov, and Benjamin Larry Craig. Rule responder: RuleML-based agents for distributed collaboration on the pragmatic web. In *Proceedings of the 2nd International Conference on Pragmatic Web, ICPW 2007, Tilburg, The Netherlands, October 22-23, 2007*, volume 280 of *ACM International Conference Proceeding Series*, pages 17–28. ACM, 2007.
16. Adrian Paschke, Harold Boley, Zhili Zhao, Kia Teymourian, and Tara Athan. Reaction RuleML 1.0 : Standardized Semantic Reaction Rules. In *Rules on the Web: Research and Applications - 6th International Symposium, RuleML 2012, Montpellier, France, August 27-29, 2012. RuleML 2012 Proceedings*, volume 7438 of *Lecture Notes in Computer Science*, pages 100–119. Springer, 2012.
17. Adrian Paschke, Paul Vincent, Alexandre Alves, and Catherine Moxey. Tutorial on advanced design patterns in event processing. In *Proceedings of the Sixth ACM International Conference on Distributed Event-Based Systems, DEBS 2012, Berlin, Germany, July 16-20, 2012*, pages 324–334. ACM, 2012.
18. Kia Teymourian and Adrian Paschke. Semantic Rule-Based Complex Event Processing. In *Rule Interchange and Applications, International Symposium, RuleML 2009, Las Vegas, Nevada, USA, November 5-7, 2009. Proceedings*, volume 5858 of *Lecture Notes in Computer Science*, pages 82–92. Springer, 2009.
19. Kia Teymourian and Adrian Paschke. Plan-Based Semantic Enrichment of Event Streams. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*, volume 8465 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 2014.
20. Kia Teymourian, Malte Rohde, and Adrian Paschke. Processing of Complex Stock Market Events Using Background Knowledge. In *Proceedings of the 5th International RuleML2011@BRF Challenge, co-located with the 5th International Rule Symposium, Fort Lauderdale, Florida, USA, November 3-5, 2011*, volume 799 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.
21. Kia Teymourian, Malte Rohde, and Adrian Paschke. Knowledge-based processing of complex stock market events. In *15th International Conference on Extending Database Technology, EDBT '12, Berlin, Germany, March 27-30, 2012, Proceedings*, pages 594–597. ACM, 2012.