The RuleML Family of Web Rule Languages*

Harold Boley

Institute for Information Technology – e-Business,
National Research Council of Canada,
Fredericton, NB, E3B 9W4, Canada
harold.boley AT nrc DOT gc DOT ca

Abstract. The RuleML family of Web rule languages contains derivation (deduction) rule languages, which themselves have a webized Datalog language as their inner core. Datalog RuleML's atomic formulas can be (un)keyed and (un)ordered. Inheriting the Datalog features, Hornlog RuleML adds functional expressions as terms. In Hornlog with equality, such uninterpreted (constructor-like) functions are complemented by interpreted (equation-defined) functions. These are described by further orthogonal dimensions "single- vs. set-valued" and "first- vs. higher-order". Combined modal logics apply special relations as operators to atoms with an uninterpreted relation, complementing the usual interpreted ones.

1 Introduction

Efforts in Web rules have steadily increased since they were brought into focus by the RuleML Initiative [http://ruleml.org] in 2000, including DARPA's DAML Rules [http://www.daml.org/rules], IST's REWERSE [http://rewerse.net], ISO's Common Logic [http://cl.tamu.edu], OMG's Production Rule Representation (PRR) [http://www.omg.org/docs/bmi/06-02-08.pdf] as well as Semantics of Business Vocabulary and Business Rules (SBVR) [http://www.businessrulesgroup.org/sbvr.shtml], and W3C's Rule Interchange Format (RIF) [http://www.w3.org/2005/rules]. RuleML has co-evolved with some of these other efforts as well as with the Semantic Web Rule Language (SWRL) [http://www.w3.org/Submission/SWRL], the Semantic Web Services Language (SWSL) [http://www.w3.org/Submission/SWSF-SWSL], and the Web Rule Language (WRL) [http://www.w3.org/Submission/WRL]. This has been supported by, and influenced, RuleML's modular design.

The specification of RuleML constitutes a modular **family** of Web sublanguages, whose root accesses the language as a whole and whose members identify *customized*, *combinable* subsets of the language. Each of the family's sublanguages has an XML Schema definition, Web-addressed by a URI, which permits inheritance between sublanguage schemas and precise reference to the required

* Thanks to David Hirtle for creating the family's XML Schemas, and the RuleML Steering Committee for guidance. This research was partially supported by NSERC.

expressiveness. The family structure provides an expressive inclusion hierarchy for the sublanguages, and their URIs are the subjects of (model-theoretic) semantic characterization. The modular system of XML Schema definitions [BBH+05] is currently in version 0.9 [http://www.ruleml.org/modularization].

The RuleML family's top-level distinctions are derivation rules, queries, and integrity constraints as well as production and reaction rules. The most developed branch groups derivation (deduction) rule languages, which themselves have a webized Datalog language as their inner core. Hornlog RuleML adds functional expressions as terms. In Hornlog with equality, such uninterpreted (constructor-like) functions are complemented by interpreted (equation-defined) functions. This derivation rule branch is extended upward towards First Order Logic, has subbranches for negation-as-failure, strong-negation, or combined languages, and languages with 'pluggable' built-ins.

This paper takes a fresh look at the family from the perspectives of three orthogonally combinable branches: the generalized Object-Oriented RuleML (section 2) as well as the new Functional RuleML (section 3) and the preliminary Modal RuleML (section 4).

2 Rules in the Key-Order Matrix

This section will propose extensions to OO RuleML [Bol03]. RuleML's global markup conventions provide common principles for the family. XML elements are used for representing trees while XML attributes are used for distinguishing variations of a given element and, as in RDF, for webizing. Variation can thus be achieved by different attribute values rather than requiring different elements. Since the same attribute can occur in different elements, a two-dimensional classification accrues, which has the potential of quadratic tag reduction.

The data model of RuleML accommodates XML's arc-ordered, node-labeled trees and RDF's arc-labeled ('keyed'), node-labeled graphs [http://www.dfki.uni-kl.de/~boley/xmlrdf.html]. For this, RuleML complements XML-like elements – upper-cased *type* tags, as in Java classes – by RDF-like properties – lower-cased *role* tags, as in Java methods. Both kinds of tag are again serialized as XML elements, but case information makes the difference. This model with unkeyed, ordered child elements (subsection 2.1) and keyed, unordered children (subsection 2.2) has recently been generalized to a 'key-order' matrix also permitting keyed, ordered as well as unkeyed, unordered children (subsection 2.3).

As a running example, we will consider RuleML versions of the business rule "A customer is premium if their spending has been min 5000 euro in the previous year." This can be serialized using various equivalent concrete syntaxes, all corresponding to the same abstract syntax that reflects the data model.

2.1 Arguments in Order

In RuleML's most RDF-like, fully 'striped' syntax (with alternating type and role tags), the example can, e.g., be serialized interchangeably as follows:

```
<Implies>
  <head>
    <Atom>
      <op><Rel>premium</Rel></op>
      <arg index="1">
        <Var>customer</Var>
      </arg>
    </Atom>
  </head>
  <body>
    <Atom>
      <arg index="1">
        <Var>customer</Var>
      </arg>
      <arg index="3">
        <Ind>previous year</Ind>
      </arg>
      <arg index="2">
        <Ind>min 5000 euro</Ind>
      </arg>
      <op><Rel>spending</Rel></op>
    </Atom>
  </body>
</Implies>
```

```
<Implies>
  <body>
    <Atom>
      <op><Rel>spending</Rel></op>
      <arg index="1">
        <Var>customer</Var>
      </arg>
      <arg index="2">
        <Ind>min 5000 euro</Ind>
      </arg>
      <arg index="3">
        <Ind>previous year</Ind>
      </arg>
    </Atom>
  </body>
  <head>
    <Atom>
      <op><Rel>premium</Rel></op>
      <arg index="1">
        <Var>customer</Var>
      </arg>
    </Atom>
  </head>
</Implies>
```

The right-hand serialization is in `<Implies>` **normal form**, with the `<body>` role tag before the `<head>` role tag, the `<op>` role before all `<arg>` roles, and the `<arg>` roles ordered according to increasing `<index>` attribute values.

Once in `<Implies>` normal form, all `<op>` and `<arg>` roles can be omitted (left), and the `<body>` and `<head>` roles, too (right):

```
<Implies>
  <body>
    <Atom>
      <Rel>spending</Rel>
      <Var>customer</Var>
      <Ind>min 5000 euro</Ind>
      <Ind>previous year</Ind>
    </Atom>
  </body>
  <head>
    <Atom>
      <Rel>premium</Rel>
      <Var>customer</Var>
    </Atom>
  </head>
</Implies>
```

```
<Implies>

  <Atom>
    <Rel>spending</Rel>
    <Var>customer</Var>
    <Ind>min 5000 euro</Ind>
    <Ind>previous year</Ind>
  </Atom>


  <Atom>
    <Rel>premium</Rel>
    <Var>customer</Var>
  </Atom>

</Implies>
```

The right-hand serialization shows RuleML's most XML-like, fully 'stripe-skipped' syntax [http://esw.w3.org/topic/StripeSkipping]. Notice that in all of these syntaxes the three argument positions of the ternary `spending` relation

carry information that must be known, e.g. via a signature declaration, for correct interpretation.

2.2 Slots are Key

There is an alternative to signature declarations for determining the roles of children in atomic formulas. In Object-Oriented RuleML [Bol03], the earlier *positional* representation style is complemented by a *slotted* style: the 'system-level' data model with type and role tags is also made available on the 'user-level', permitting F-logic-like role→filler pairs.

For this, a single (system-level) metarole `<slot>` with two children is employed, the first naming different (user-level) roles, and the second containing their fillers.

For example, the fully stripe-skipped positional `<Implies>` rule above can be made slotted with user-level roles `<spender>` etc.:

```
<Implies>
  <Atom>
    <Rel>spending</Rel>
    <slot><Ind>spender</Ind><Var>customer</Var></slot>
    <slot><Ind>amount</Ind><Ind>min 5000 euro</Ind></slot>
    <slot><Ind>period</Ind><Ind>previous year</Ind></slot>
  </Atom>
  <Atom>
    <Rel>premium</Rel>
    <slot><Ind>client</Ind><Var>customer</Var></slot>
  </Atom>
</Implies>
```

The correct interpretation of the three `spending` arguments is no longer position-dependent and additional arguments such as `region` can be added without affecting any existing interpretation. A child element, rather than an attribute, was decided upon for naming the role to provide an extension path towards (e.g., F-logic's) schema-querying options. Although problematic in general [http://www.daml.org/listarchive/joint-committee/1376.html], we did not want to exclude the possibility in RuleML to query a role constant like `<Ind>period</Ind>` above through a role variable like `<Var>time</Var>`.

2.3 Making Independent Distinctions

Recent work on the Positional-Slotted Language [http://www.ruleml.org/#POSL] led to orthogonal dimensions extending the RuleML 0.9 roles `<arg . . .>` and `<slot>`. So far, the *unkeyed* `<arg index="...">` was always *ordered*, as indicated by the mandatory `index` attribute, and the *keyed* `<slot>` was always *unordered*, as indicated by the lack of an `index` attribute. This can be generalized by allowing an optional `index` attribute for both roles, as shown by the independent distinctions in the following **key-order matrix**:

	ordered	*unordered*
keyed	`<slot index="...">`	`<slot>`
unkeyed	`<arg index="...">`	`<arg>`

Two extra orthogonal combinations are obtained from this system.

First, *keyed*, *ordered* children permit positionalized slots, as in this `cost` fact:

```
<Atom>
  <Rel>cost</Rel>
  <slot index="1"><Ind>item</Ind><Ind>jewel</Ind></slot>
  <slot index="2"><Ind>price</Ind><Data>6000</Data></slot>
  <slot index="3"><Ind>taxes</Ind><Data>2000</Data></slot>
</Atom>
```

Here, slot names `item`, `price`, and `taxes` are provided, e.g. for readability, as well as index positions `1-3`, e.g. for efficiency.

Second, *unkeyed*, *unordered* children permit elements acting like those in a bag (finite multiset), as in this `transport` fact:

```
<Atom>
  <Rel>transport</Rel>
  <arg><Ind>chair</Ind></arg>
  <arg><Ind>chair</Ind></arg>
  <arg><Ind>table</Ind></arg>
</Atom>
```

Here, the arguments are specified to be commutative and 'non-idempotent' (duplicates are kept). Ground bags can be normalized using some canonical (e.g., lexicographic) order, and then linearly compared for equality. Results in (non-ground) bag unification are also available (e.g., [DV99]).

The RuleML 0.9 rest terms (normally variables) can be correspondingly generalized by allowing a role `<ordertail>` to unify with `index`-attributed rest elements, `<arg index="...">` and `<slot index="...">`, as well as a role `<commutail>` to unify with `index`-less rest elements, `<arg>` and `<slot>`.

The unkeyed RuleML case can be compared with Xcerpt [SB04] in that both distinguish ordered/unordered and total/partial term specifications, where the latter in RuleML is notated as the absence/presence of an `<orderest>` role with a fresh (e.g., anonymous) variable. However, following our XML-RDF-unifying data model [http://www.dfki.uni-kl.de/~boley/xmlrdf.html], in RuleML these distinctions are made for term normalization and unification; in Xcerpt, for matching query terms to data terms.

3 Equality for Functions

While section 2 dealt with RuleML for logic programming (LP) on the Semantic Web, functional programming (FP) [BKPS03] is also playing an increasing Web role, with XSLT and XQuery [FRSV05] being prominent examples. We present here the design of Functional RuleML, developed via orthogonal notions and

freely combinable with the previous Relational RuleML, including OO RuleML [Bol03], discussed in section 2. This branch of the family will also allow for FP/LP-integrated programming (FLP), including OO FLP, on the Semantic Web. Some background on FLP markup languages was given in [Bol00a].

Since its beginning in 2000, with RFML [http://www.relfun.org/rfml] as one of its inputs, RuleML has permitted the markup of oriented (or directed) equations for defining the value(s) of a function applied to arguments, optionally conditional on a body as in Horn rules. Later, this was extended to logics with symmetric (or undirected) equality for the various sublanguages of RuleML, but the `Equal` element has still often exploited the left-to-right orientation of its (abridged) textual syntax.

It has been a RuleML issue that the constructor (`Ctor`) of a complex term (`Cterm`) is disjoined, as an XML element, from the user-defined function (`Fun`) of a call expression (`Nano`), although these can be unified by proceeding to a logic with equality. For example, while currently call patterns can contain `Cterm`s but not `Nano`s, obeying the "constructor discipline" [O'D85], the latter should also be permitted to legalize 'optimization' rules like `reverse(reverse(?L)) = ?L`.

This section thus conceives both `Cterm`s and `Nano`s as expression (`<Expr>`) elements and distinguishes 'uninterpreted' (constructor) vs. 'interpreted' (user-defined) functions just via an XML attribute; another attribute likewise distinguishes the (single- vs. set-)valuedness of functions (subsection 3.1). We then proceed to the nesting of all of these (subsection 3.2). Next, for defining (interpreted) functions, unconditional (oriented) equations are introduced (subsection 3.3). These are then extended to conditional equations, i.e. Horn logic implications with an equation as the head and possible equations in the body (subsection 3.4). Higher-order functions are finally added, both named ones such as `Compose` and λ-defined ones (subsection 3.5).

3.1 Interpretedness And Valuedness

The different notions of 'function' in LP and FP have been a continuing design issue:

LP: *Uninterpreted functions* **denote** unspecified values when applied to arguments, not using function definitions.
FP: *Interpreted functions* **compute** specified returned values when applied to arguments, using function definitions.

Uninterpreted function are also called 'constructors' since the values denoted by their application to arguments will be regarded as the syntactic data structure of these applications themselves.

For example, the function `first-born`: $Man \times Woman \rightarrow Human$ can be uninterpreted, so that `first-born(John, Mary)` just denotes the first-born child; or, interpreted, e.g. using definition `first-born(John, Mary)` = `Jory`, so the application returns `Jory`.

The distinction of uninterpreted vs. interpreted functions in RuleML 0.9 is marked up using different elements, `<Ctor>` vs. `<Fun>`. Proceeding to the

increased generality of logic with equality (cf. introductory discussion), this should be changed to a single element name, `<Fun>`, with different attribute values, `<Fun in="no">` vs. `<Fun in="yes">`, respectively: The use of a Function's interpreted attribute with values `"no"` vs. `"yes"` directly reflects uninterpreted vs. interpreted functions (those for which, `in` the rulebase, `no` definitions are expected vs. those for which they are). Functions' respective RuleML 0.9 [http://www.ruleml.org/0.9] applications with `Cterm` vs. `Nano` can then uniformly become Expressions for either interpretedness.

The two versions of the example can thus be marked up as follows (where `"u"` stands for `"no"` or `"yes"`):

```
<Expr>
  <Fun in="u">first-born</Fun>
  <Ind>John</Ind>
  <Ind>Mary</Ind>
</Expr>
```

In RuleML 0.9 as well as in RFML and its human-oriented Relfun syntax [Bol99] this distinction is made on the level of expressions, the latter using square brackets vs. round parentheses for applications. Making the distinction through an attribute in the `<Fun>` rather than `<Expr>` element will permit higher-order functions (cf. subsection 3.5) to return, and use as arguments, functions that include interpretedness markup.

A third value, `"semi"`, is proposed for the interpreted attribute: *Semi-interpreted functions* **compute** an application if a definition exists and **denote** unspecified values else (via the syntactic data structure of the application, which we now write with Relfun-like square brackets). For example, when `"u"` stands here for `"semi"`, the above application returns `Jory` if definition `first-born(John, Mary)` = `Jory` exists and denotes `first-born[John, Mary]` itself if no definition exists for it. Because of its neutrality, `in="semi"` is proposed as the default value.

In both XML and UML processing, functions (like relations in LP) are often *set-valued* (*non-deterministic*). This is accommodated by introducing a `val`ued attribute with values including `"1"` (deterministic: exactly one) and `"0.."` (set-valued: zero or more). Our `val` specifications can be viewed as transferring to functions, and generalizing, the cardinality restrictions for (binary) properties (i.e., unary functions) in description logic and the determinism declarations for (moded) relations in Mercury [SHC96].

For example, the set-valued function `children`: $Man \times Woman \rightarrow 2^{Human}$ can be interpreted and set-`val`ued, using definition `children(John, Mary)` = {`Jory, Mahn`}, so that the application `children(John, Mary)` returns {`Jory, Mahn`}.

The example is then marked up thus (other legal `val` values here would be `"0..3"`, `"1..2"`, and `"2"`):

```
<Expr>
  <Fun in="yes"
       val="0..">children</Fun>
  <Ind>John</Ind>
  <Ind>Mary</Ind>
</Expr>
```

Because of its highest generality, `val="0.."` is proposed as the default.

While uninterpreted functions usually correspond to `<Fun in="no" val="1">`, attribute combinations of `in="no"` with a `val` unequal to `"1"` will be useful when uninterpreted functions are later to be refined into interpreted set-valued functions (which along the way can lead to semi-interpreted ones).

Interpretedness and valuedness constitute orthogonal dimensions in our design space, and are also orthogonal to the dimensions of the subsequent subsections, although space limitations prevent the discussion of all of their combinations in this section.

3.2 Nestings

One of the advantages of interpreted functions as compared to relations is that the returned values of their applications permit nestings, avoiding flat relational conjunctions with shared logic variables.

For example, the function `age` can be defined for `Jory` as `age(Jory) = 12`, so the nesting `age(first-born(John, Mary))`, using the `first-born` definition of subsection 3.1, gives `age(Jory)`, then returns `12`.

Alternatively, the function `age` can be defined for the uninterpreted `first-born` application as `age(first-born[John, Mary]) = 12`, so the nesting `age(first-born[John, Mary])` immediately returns `12`.

Conversely, the function `age` can be left uninterpreted over the returned value of the `first-born` application, so the nesting `age[first-born(John, Mary)]` denotes `age[Jory]`.

Finally, both the functions `age` and `first-born` can be left uninterpreted, so the nesting `age[first-born[John, Mary]]` just denotes itself.

The four versions of the example can now be marked up thus (where `"u"` and `"v"` can independently assume `"no"` or `"yes"`):

```
<Expr>
  <Fun in="u">age</Fun>
  <Expr>
    <Fun in="v">first-born</Fun>
    <Ind>John</Ind>
    <Ind>Mary</Ind>
  </Expr>
</Expr>
```

Nestings are permitted for set-valued functions, where an (interpreted or uninterpreted) outer function is automatically mapped over all elements of a set returned by an inner (interpreted) function.

For example, the element-valued function `age` can be extended for `Mahn` with `age(Mahn)` = 9, and nested, interpreted, over the set-valued interpreted function `children` of subsection 3.1: `age(children(John, Mary))` via `age({Jory, Mahn})` returns {`12, 9`}.

Similarly, `age` can be nested uninterpreted over the interpreted `children`: `age[children(John, Mary)]` via `age[{Jory, Mahn}]` returns {`age[Jory], age[Mahn]`}.

The examples can be marked up thus (only `"u"` is left open for `"no"` or `"yes"`):

```
<Expr>
  <Fun in="u">age</Fun>
  <Expr>
    <Fun in="yes"
         val="0..">children</Fun>
    <Ind>John</Ind>
    <Ind>Mary</Ind>
  </Expr>
</Expr>
```

3.3 Unconditional Equations

In subsections 3.1 and 3.2 we have employed expression-defining equations without giving their actual markup. Let us consider these in more detail here, starting with *unconditional equations*.

For this, we introduce a modified RuleML 0.9 `<Equal>` element, permitting both symmetric (or undirected) and oriented (or directed) equations via an `oriented` attribute with respective `"no"` and `"yes"` values. Since it is more general, `oriented="no"` is proposed as the default.

Because of the potential orientedness of equations, the RuleML 0.9 `<side>` role tag within the `<Equal>` type tag will be refined into `<lhs>` and `<rhs>` for an equation's **l**eft-**h**and **s**ide and **r**ight-**h**and **s**ide, respectively.

For example, the subsection 3.1 equation `first-born(John, Mary)` = `Jory` can now be marked up thus:

```
<Equal oriented="yes">
  <lhs>
    <Expr>
      <Fun in="yes">first-born</Fun>
      <Ind>John</Ind>
      <Ind>Mary</Ind>
    </Expr>
  </lhs>
  <rhs>
    <Ind>Jory</Ind>
  </rhs>
</Equal>
```

While the explicit `<lhs>` and `<rhs>` role tags emphasize the orientation, and are used as RDF properties when mapping this markup to RDF graphs, they can be omitted via stripe-skipping [http://esw.w3.org/topic/StripeSkipping]: the `<lhs>` and `<rhs>` roles of `<Equal>`'s respective first and second subelements can still be uniquely recognized.

This, then, is the stripe-skipped example:

```
<Equal oriented="yes">
  <Expr>
    <Fun in="yes">first-born</Fun>
    <Ind>John</Ind>
    <Ind>Mary</Ind>
  </Expr>
  <Ind>Jory</Ind>
</Equal>
```

Equations can also have nested left-hand sides, where often the following restrictions apply: The `<Expr>` directly in the left-hand side must use an interpreted function. Any `<Expr>` nested into it must use an uninterpreted function to fulfill the so-called "constructor discipline" [O'D85]; same for deeper nesting levels. If we want to obey it, we use `in="no"` within these nestings. An equation's right-hand side `<Expr>` can use uninterpreted or interpreted functions on any level of nesting, anyway.

For example, employing binary `subtract` and nullary `this-year` functions, the equation `age(first-born[John, Mary])` = `subtract(this-year(),1993)` leads to this stripe-skipped 'disciplined' markup:

```
<Equal oriented="yes">
  <Expr>
    <Fun in="yes">age</Fun>
    <Expr>
      <Fun in="no">first-born</Fun>
      <Ind>John</Ind>
      <Ind>Mary</Ind>
    </Expr>
  </Expr>
  <Expr>
    <Fun in="yes">subtract</Fun>
    <Expr>
      <Fun in="yes">this-year</Fun>
    </Expr>
    <Data>1993</Data>
  </Expr>
</Equal>
```

3.4 Conditional Equations

Let us now proceed to oriented *conditional equations*, which use a (defining, oriented) `<Equal>` element as the conclusion of an `<Implies>` element, whose condition may employ other (testing, symmetric) equations. An equational condition may also bind auxiliary variables. While condition and conclusion can be marked up with explicit `<body>` and `<head>` roles, respectively, also allowing the conclusion as the first subelement, we will use a stripe-skipped markup where the condition must be the first subelement.

For example, using a unary `birth-year` function in the condition, and two ("`?`"-prefixed) variables, the conditional equation (written with a top-level "⇒") `?B` = `birth-year(?P)` ⇒ `age(?P)` = `subtract(this-year(),?B)` employs an equational condition to test whether the `birth-year` of a person `?P` is known, assigning it to `?B` for use within the conclusion. This leads to the following stripe-skipped markup:

```
<Implies>
  <Equal oriented="no">
    <Var>B</Var>
    <Expr>
      <Fun in="yes">birth-year</Fun>
      <Var>P</Var>
    </Expr>
  </Equal>
  <Equal oriented="yes">
    <Expr>
      <Fun in="yes">age</Fun>
      <Var>P</Var>
    </Expr>
    <Expr>
      <Fun in="yes">subtract</Fun>
      <Expr>
        <Fun in="yes">this-year</Fun>
      </Expr>
      <Var>B</Var>
    </Expr>
  </Equal>
</Implies>
```

Within conditional equations, relational conditions can be used besides equational ones.

Thus, using a binary `lessThanOrEqual` relation in the condition, the conditional equation `lessThanOrEqual(age(?P),15)` ⇒ `discount(?P,?F)` = `30` with a free variable `?F` (flight) and a data constant `30` (percent), gives this markup:

```
<Implies>
  <Atom>
    <Rel>lessThanOrEqual</Rel>
    <Expr>
      <Fun in="yes">age</Fun>
      <Var>P</Var>
    </Expr>
    <Data>15</Data>
  </Atom>
  <Equal oriented="yes">
    <Expr>
      <Fun in="yes">discount</Fun>
      <Var>P</Var>
      <Var>F</Var>
    </Expr>
    <Data>30</Data>
  </Equal>
</Implies>
```

Notice the following interleaving of FP and LP (as characteristic for FLP): The function `discount` is defined using the relation `lessThanOrEqual` in the condition. The `<Atom>` element for the `lessThanOrEqual` relation itself contains a nested `<Expr>` element for the `age` function.

For conditional equations of Horn logic with equality in general [Pad88], the condition is a conjunction of `<Atom>` and `<Equal>` elements.

3.5 Higher-Order Functions

Higher-order functions are characteristic for FP and thus should be supported by Functional RuleML. A *higher-order function* permits functions to be passed to it as (actual) parameters and to be returned from it as values.

Perhaps the most well-known higher-order function is `Compose`, taking two functions as parameters and returning as its value a function performing their sequential composition.

For example, the composition of the `age` and `first-born` functions of subsection 3.1 is performed by `Compose(age,first-born)`. Here is the markup for the interpreted and uninterpreted use of both of the parameter functions (where we use the default `in="semi"` for the higher-order function and let `"u"` and `"v"` independently assume `"no"` or `"yes"` for the first-order functions):

```
<Expr>
  <Fun>Compose</Fun>
  <Fun in="u">age</Fun>
  <Fun in="v">first-born</Fun>
</Expr>
```

The application of a parameterized `Compose` expression to arguments is equivalent to the nested application of its parameter functions.

For example, when interpreted with the definitions of subsection 3.1, `Compose(age,first-born)(John, Mary)` via age(first-born(John, Mary)) returns `12`.

All four versions of this sample application can be marked up thus (with the usual `"u"` and `"v"`):

```
<Expr>
  <Expr>
    <Fun>Compose</Fun>
    <Fun in="u">age</Fun>
    <Fun in="v">first-born</Fun>
  </Expr>
  <Ind>John</Ind>
  <Ind>Mary</Ind>
</Expr>
```

Besides being applied in this way, a `Compose` expression can also be used as a parameter or returned value of another higher-order function.

To allow the general construction of anonymous functions, `Lambda` formulas from λ-calculus [Bar97] are introduced. A λ-*formula* quantifies variables that occur free in a functional expression much like a $\forall$-*formula* does for a relational atom. So we can extend principles developed for explicit-quantifier markup in FOL RuleML [http://www.w3.org/Submission/FOL-RuleML], where quantifiers are allowed on all levels of rulebase elements.

For example, the function returned by `Compose(age,first-born)` can now be explicitly given as λ`(?X, ?Y)age(first-born(?X, ?Y))`. Here is the markup for its interpreted and uninterpreted use (with the usual `"u"` and `"v"`):

```
<Lambda>
  <Var>X</Var>
  <Var>Y</Var>
  <Expr>
    <Fun in="u">age</Fun>
    <Expr>
      <Fun in="v">first-born</Fun>
      <Var>X</Var>
      <Var>Y</Var>
    </Expr>
  </Expr>
</Lambda>
```

This `Lambda` formula can be applied as the `Compose` expression was above. The advantage of `Lambda` formulas is that they allow the direct λ-*abstraction* of arbitrary expressions, not

just for (sequential or parallel) composition etc. An example is λ(?X, ?Y)plex(age(?X), xy, age(?Y), fxy, age(first-born(?X, ?Y))), whose markup should be obvious if we note that plex is the interpreted analog to RuleML's uninterpreted built-in function for n-ary complex-term (e.g., tuple) construction.

By also abstracting the parameter functions, age and first-born, Compose can be defined generally via a Lambda formula as Compose(?F, ?G) = λ(?X, ?Y) ?F(?G(?X, ?Y)). Its markup can distinguish object (first-order) Variables like ?X vs. function (higher-order) ones like ?F via attribute values ord="1" vs. ord="h".

4 Combining Modal Logics

This section introduces a preliminary extension of RuleML for modal logics, whose relevance to the Semantic Web has been known for quite some time [Bol00b]. Modal operators can be represented *generically* as special relations at least one of whose arguments is a proposition represented as an embedded atom that has an uninterpreted relation (including another modal operator), complementing the usual main atoms that have interpreted relations:

- **Alethic** operators: The relations necessary (□) and possible (◇) represent the operators of modal logic in the narrow sense.
- **Deontic** operators: The relations must and may are used to express obligations and permissions (e.g., in business rules).
- Further modal operators can be introduced as relations for temporal (e.g., to plan/diagnose reactive rules), epistemic (e.g., in authentication rules), and other modalities.

These logics are based on Kripke-style possible worlds semantics, usually focusing one pair of operators at a time. However, recently logicians have begun to develop *many-dimensional modal* [GKWZ03] and *multi-modal* [Hen06] systems in which, e.g., alethic, epistemic, and temporal operators can be combined in unified formal frameworks. Motivated by the co-occurrence of several modalities in RuleML application (e.g., business) domains and enabled by the combination principle of sublanguages in the RuleML family, such combined modal logics are proposed for the Modal RuleML extension.

Modal Relations are indicated by a modal="yes" attribute setting, where the default value, for non-modal Relations, is modal="no". Furthermore, we regard the Atoms in the contexts created by modal relations as counterparts to the Expressions with an in="no" function used by relations beyond Datalog: the interpretedness attribute for Functions is also allowed for Relations. While in="semi" is the proposed default value for functions, in="yes" is proposed as the default value for relations to keep their interpretation for non-modal sublanguages unchanged when no in attribute is given. The use of Atoms with an uninterpreted Relation for embedded propositions can be regarded as a customized form of the universally usable Reify element introduced for RuleML's

SWSL-Rules sublanguage [http://www.w3.org/Submission/SWSF-SWSL/#ruleml-reification]. These modal-logic characteristics will be further explored through the examples below.

The unary alethic fact $\Box prime(1)$ is serialized thus (the main `Rel` by default obtains `in="yes"`; the embedded `Rel` gets `modal="no"`):

```
<Atom>
  <Rel modal="yes">necessary</Rel>
  <Atom>
    <Rel in="no">prime</Rel>
    <Data>1</Data>
  </Atom>
</Atom>
```

The binary epistemic fact $knows(Mary, material(moon, rock))$ likewise becomes this serialization:

```
<Atom>
  <Rel modal="yes">knows</Rel>
  <Ind>Mary</Ind>
  <Atom>
    <Rel in="no">material</Rel>
    <Ind>moon</Ind>
    <Ind>rock</Ind>
  </Atom>
</Atom>
```

With the *veridicality* axiom $Knows_{Agent} proposition \rightarrow proposition$, here in a unary notation, the non-modal fact $material(moon, rock)$ can be derived, which is serialized thus (using defaults `<Rel modal="no" in="yes">`):

```
<Atom>
  <Rel>material</Rel>
  <Ind>moon</Ind>
  <Ind>rock</Ind>
</Atom>
```

The nested epistemic-alethic fact $knows(Mary, \Box prime(1))$ of a combined modal logic can then be serialized as follows:

```
<Atom>
  <Rel modal="yes">knows</Rel>
  <Ind>Mary</Ind>
  <Atom>
    <Rel modal="yes" in="no">necessary</Rel>
    <Atom>
      <Rel in="no">prime</Rel>
      <Data>1</Data>
    </Atom>
  </Atom>
</Atom>
```

Here, the knows and necessary Relations are modal, hence are attributed with modal="yes". However, the necessary relation is furthermore attributed as uninterpreted, since it occurs in the context of the knows relation, which itself, by default, is interpreted.

5 Conclusions

The key-order matrix of the generalized Object-Oriented RuleML presented in this paper – when used for expressions with uninterpreted functions rather than for atoms – makes four data containers available in a systematic manner: (keyed) positionalized records and ordinary records as well as (unkeyed) tuples and bags.

The *unordered* column of the matrix could be extended by a column for 'idempotent' slots and arguments (duplicates are merged), leading to data containers for unique-key records and sets. Unification algorithms could be based on earlier work (e.g., [DV99]).

The design of Functional RuleML as presented here also benefits other sublanguages of RuleML, e.g. because of the more 'logical' complex terms. Functional RuleML, as a development of FOL RuleML, could furthermore benefit all of SWRL FOL [http://www.w3.org/Submission/2005/01]. However, there are some open issues, two of which will be discussed below.

Certain constraints on the values of our attributes cannot be enforced with DTDs and are hard to enforce with XSDs, e.g. in="no" on functions in call patterns in case we wanted to always enforce the constructor discipline (cf. subsection 3.3). However, a semantics-oriented validation tool will be required for future attributes anyway, e.g. for testing whether a rulebase is stratified. Thus we propose that such a static-analysis tool should be developed to make fine-grained distinctions for all 'semantic' attributes.

The proposed defaults for some of our attributes may require a future revision. It might be argued that the default in="semi" for functions is a problem since equations could be invoked inadvertently for functions that are applied without an explicit in attribute. However, notice that, intuitively speaking, the default oriented="no" for equations permits to 'revert' any function call, using the same equation in the other direction. Together, those defaults thus constitute a kind of 'vanilla' logic with equality, which can (only) be changed via our explicit attribute values.

While our logical design does not specify any evaluation strategy for nested Expressions with interpreted Functions, we have preferred 'call-by-value' in implementations [Bol00a]. A reference interpreter for Functional RuleML is planned as an extension of OO jDREW [BBH+05]; an initial step has been taken by implementing oriented ground equality via an EqualTable data structure for equivalence classes [http://www.w3.org/2004/12/rules-ws/paper/49].

The preliminary design of a Modal RuleML with combined modalities addresses modeling needs of the business rules community. As Hornlog RuleML uses relations over embedded expressions with an uninterpreted function, modal

operators are special relations applied to embedded atoms with an uninterpreted relation. Efficiently implementing combined modal logics is a current challenge.

References

[Bar97] Henk Barendregt. The Impact of the Lambda Calculus in Logic and Computer Science. *The Bulletin of Symbolic Logic*, 3(2):181–215, 1997.

[BBH+05] Marcel Ball, Harold Boley, David Hirtle, Jing Mei, and Bruce Spencer. The OO jDREW Reference Implementation of RuleML. In *Proc. Rules and Rule Markup Languages for the Semantic Web (RuleML-2005)*. LNCS 3791, Springer-Verlag, November 2005.

[BKPS03] Paul A. Bailes, Colin J. M. Kemp, Ian Peake, and Sean Seefried. Why Functional Programming Really Matters. In *Applied Informatics*, pages 919–926, 2003.

[Bol99] Harold Boley. Functional-Logic Integration via Minimal Reciprocal Extensions. *Theoretical Computer Science*, 212:77–99, 1999.

[Bol00a] Harold Boley. Markup Languages for Functional-Logic Programming. In *9th International Workshop on Functional and Logic Programming, Benicassim, Spain*, pages 391–403. UPV University Press, Valencia, publication 2000/2039, September 2000.

[Bol00b] Harold Boley. Relationships Between Logic Programming and RDF. In *Proc. 1st Pacific Rim International Workshop on Intelligent Information Agents (PRIIA 2000)*. University of Melbourne, Australia; LNCS 2112, August 2000.

[Bol03] Harold Boley. Object-Oriented RuleML: User-Level Roles, URI-Grounded Clauses, and Order-Sorted Terms. In *Proc. Rules and Rule Markup Languages for the Semantic Web (RuleML-2003)*. LNCS 2876, Springer-Verlag, October 2003.

[DV99] E. Dantsin and A. Voronkov. A Nondeterministic Polynomial-Time Unification Algorithm for Bags, Sets and Trees. *Lecture Notes in Computer Science*, 1578:180–196, 1999.

[FRSV05] Achille Fokoue, Kristoffer Rose, Jérôme Siméon, and Lionel Villard. Compiling XSLT 2.0 into XQuery 1.0. In *Proceedings of the Fourteenth International World Wide Web Conference*, pages 682–691, Chiba, Japan, May 2005. ACM Press.

[GKWZ03] Dov M. Gabbay, Ágnes Kurucz, Frank Wolter, and Michael Zakharyaschev. *Many-Dimensional Modal Logics: Theory and Applications*. Elsevier, Amsterdam, 2003.

[Hen06] Vincent F. Hendricks. *Mainstream and Formal Epistemology*. Cambridge University Press, New York, 2006.

[O'D85] M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, Cambridge, Mass., 1985.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*. EATCS Monographs on Theoretical Computer Science, Vol. 16. Springer, 1988.

[SB04] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.

[SHC96] Z. Somogy, F. Henderson, and T. Conway. The Execution Algorithm of Mercury, An Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.