Invertible Bidirectional Metalogical Translation Between Prolog and RuleML for Knowledge Representation and Querying

Mark Thom[1] Harold Boley[2] Theodoros Mitsikas[3]

[1] RuleML, Canada
markjordanthom[AT]gmail[DOT]com

[2] University of New Brunswick, Canada
harold[DOT]boley[AT]unb[DOT]ca

[3] National Technical University of Athens, Greece
mitsikas[AT]central[DOT]ntua[DOT]gr

The 4th International Joint Conference on Rules and Reasoning
Virtual, June 29 - July 1 2020

Introduction and Motivation

- BiMetaTrans(Prolog, RuleML) is an invertible bidirectional metalogical translator across subsets of ISO Prolog and RuleML/XML 1.02 on the level of Negation-as-failure Horn logic with Equality (NafHornlogEq)
- BiMetaTrans, itself a Prolog program, introduces tighter integration between RuleML and Prolog, enabling reuse of, e.g., RuleML Knowledge Bases (KBs) and query engines, analysis tools, editors, APIs, and composed translators
- BiMetaTrans enables bidirectional translation between RuleML – a KB-interoperation hub – and Prolog – an actively refined and expanded ISO standard
- A metalogical encoding, along with the introduction of the *split translation pattern*, allows BiMetaTrans to build upon the abstraction of Definite Clause Grammars (DCGs), supporting invertible bi-translation

The Prolog/`'$V'` Metalogical Encoding

- KBs and queries should be represented as *ground terms* – Prolog compounds/trees containing no free variables
- Variable names should be preserved across translation boundaries – this information is lost if, e.g., RuleML/XML `<Var>` elements are directly translated as Prolog variables
- Invertibility means that a Prolog/`'$V'` encoding is entirely recoverable from its translated RuleML/XML document, and vice versa (modulo whitespace)
- The `'$V'` compound is introduced for this purpose:

`'$V'(myvar)` $\Longleftrightarrow$ `<Var>myvar</Var>`

The Prolog/'$V' Metalogical Encoding (Cont'd)

The translation is defined recursively in the below translation table (truncated from the paper):

$\chi_{\texttt{'\$V'}}$ maps from Prolog/'$V' to RuleML/XML.

Prolog/'$V' Syntax	RuleML/XML Serialization

[
  ***assertitem*$_1$**
, . . .
, ***assertitem*$_n$**
]
	<Assert mapClosure="universal">
	  $\chi_{\texttt{'\$V'}}$(***assertitem*$_1$**)
	  . . .
	  $\chi_{\texttt{'\$V'}}$(***assertitem*$_n$**)
	</Assert>

?- queryitem	<Query mapClosure="existential"> $\chi_{\texttt{'\$V'}}$(***queryitem***) </Query>

pred (
  ***argument*$_1$**,
  . . .
  ***argument*$_n$**
)
	<Atom>
	  <Rel>$\chi_{\texttt{'\$V'}}$(***pred***)</Rel>
	  $\chi_{\texttt{'\$V'}}$(***argument*$_1$**)
	  . . .
	  $\chi_{\texttt{'\$V'}}$(***argument*$_n$**)
	</Atom>

The Prolog/'$V' Metalogical Encoding (Cont'd)

Prolog/'$V' Syntax	RuleML/XML Serialization
left = ***right***	<Equal> $\chi_{\texttt{'\$V'}}$(***left***) $\chi_{\texttt{'\$V'}}$(***right***) </Equal>
\+ ***form***	<Naf> $\chi_{\texttt{'\$V'}}$(***form***) </Naf>
consequent :- ***antecedent***	<Implies> <then>$\chi_{\texttt{'\$V'}}$(***consequent***)</then> <if>$\chi_{\texttt{'\$V'}}$(***antecedent***)</if> </Implies>

$\chi_{\texttt{'\$V'}}$ is complemented by inverse translation function $\pi_{\texttt{xmL}}$ from RuleML/XML to Prolog/'$V', reading table right to left.

(For the rest of the table, see the paper!)

Strategy to Prove Invertibility

We outline a proof showing
the invertibility of the translation:

$$\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}} = id_{\texttt{'\$V'}} \ \wedge \ \chi_{\texttt{'\$V'}} \circ \pi_{\texttt{xmL}} = id_{\texttt{xmL}}$$

("$\circ$" denotes function composition and
id_X the identity function on set X)

Apply structural induction to
Prolog/`'$V'` and RuleML/XML trees.

Each row of the translation table is a proof case.

Property that we will prove:

$$\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}} = id_{\texttt{'\$V'}} \quad \text{(inverse direction is similar)}$$

Required Definitions

Definitions:

- The Prolog symbol set $\mathcal{S}$
- The reserved symbol set $\mathcal{R} = \{$ `=`, `:-`, `\+`, `'$V'`, `'.'` $\} \subset \mathcal{S}$
- A RuleML/XML document is $\mathcal{R}$-*valid* if its atoms never contain a member of $\mathcal{R}$ in their `<Rel>` elements

The Need for $\mathcal{R}$-validity

$\mathcal{R}$-validity is vital to invertibility. For example, suppose we naïvely translated this RuleML/XML atom:

```
<Atom><Rel>=</Rel><Var>x</Var><Var>y</Var></Atom>
```

BiMetaTrans would represent it as the Prolog compound specifiable in Prolog under equivalent ($\sim$) prefix and infix forms:

$$\texttt{'='('\$V'(x),'\$V'(y))} \sim \texttt{'\$V'(x) = '\$V'(y)}$$

Following the translation table in the opposite direction, BiMetaTrans would produce

```
<Equal><Var>x</Var><Var>y</Var></Equal>
```

which is not the original XML!

Proof Outline of Invertibility

We focus on the `<Implies>` element of the case analysis.

We want to show this equation:

$$\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$v'}}(\textbf{\textit{then}} :\!- \textbf{\textit{if}}) \;=\; \pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$v'}}(\textbf{\textit{then}}) :\!- \pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$v'}}(\textbf{\textit{if}})$$

The rest follows by induction.

The translation table has a row permitting this rewrite:

$$\chi_{\texttt{'\$v'}}(\textbf{\textit{then}} :\!- \textbf{\textit{if}}) \rightsquigarrow \begin{array}{l} \texttt{<Implies>} \\ \texttt{<then>}\chi_{\texttt{'\$v'}}(\textbf{\textit{then}})\texttt{</then>} \\ \texttt{<if>}\chi_{\texttt{'\$v'}}(\textbf{\textit{if}})\texttt{</if>} \\ \texttt{</Implies>} \end{array}$$

$\pi_{\texttt{xmL}}$ performs the inverse transformation, establishing the above equation.

DCGs: A DSL for Parsing

BiMetaTrans is written almost entirely in the Domain-Specific Language (DSL) of Definite Clause Grammars (DCGs).

DCGs undergo compile-time expansion into plain Prolog predicates, as in:

Before DCG Expansion	After DCG Expansion
`sign('-') --> "-".`	`sign('-',_A,_B) :- _A = ['-'|_B].`
`sign('+') --> "+".`	`sign('+',_A,_B) :- _A = ['+'|_B].`

Note how the variables `_A` and `_B` only appear in the expanded DCG. They help form a specialized Prolog data structure commonly known as the "difference list."

Difference lists are lists with variable tails. By keeping track of the tail variable, it becomes possible to concatenate terms to difference lists in constant time.

DCGs: A DSL for Parsing (Cont'd)

To see how difference list tail variables are threaded across multiple lists, we turn to the more elaborate example:

Before DCG Expansion	After DCG Expansion

`conditions([I | Is]) -->`
`  condition(I),`
`  conditions(Is).`
	`conditions([I | Is], _A, _B) :-`
	`  condition(I, _A, _C),`
	`  conditions(Is, _C, _B).`

The variable `_C` is bound to the tail of the difference list created by `condition` and to the head of the difference list next created by `conditions`.

The tail of the expanded DCG is always that of its final called grammar, here named `_B`. If the DCG succeeds, `_B` will be bound to either `[]` or the head of another difference list.

DCGs and the Split Translation Pattern

The difference list passed among DCGs of BiMetaTrans is always a RuleML/XML string that is either parsed or generated, depending on the translation direction.

In the `ruleml_atom` DCG, this leads to the following (parse/generate) reflection:

```
ruleml_atom(Item) -->
(    { var(Item) } ->
     list_ws("<Atom>"),
     list_ws("<Rel>"),
     prolog_symbol(Name),
     list_ws("</Rel>"),
     ruleml_items(Args),
     list_ws("</Atom>"),
     { Item =.. [Name | Args] }
;    { Item =.. [Name | Args] },
     list("<Atom>"),
     list("<Rel>"),
     prolog_symbol(Name),
     list("</Rel>"),
     ruleml_items(Args),
     list("</Atom>")
).
```

DCGs and the Split Translation Pattern

The difference list passed among DCGs of BiMetaTrans is always a RuleML/XML string that is either parsed or generated, depending on the translation direction.

In the `ruleml_atom` DCG, this leads to the following (parse/generate) reflection:

```
ruleml_atom(Item) -->
(    { var(Item) } ->
     list_ws("<Atom>"),
     list_ws("<Rel>"),
     prolog_symbol(Name),
     list_ws("</Rel>"),
     ruleml_items(Args),
     list_ws("</Atom>"),
     { Item =.. [Name | Args] }
;    { Item =.. [Name | Args] },
     list("<Atom>"),
     list("<Rel>"),
     prolog_symbol(Name),
     list("</Rel>"),
     ruleml_items(Args),
     list("</Atom>")
).
```

```
Item = '$V'(myvar)
```

(parsing)

```
{ Item =.. ['$V', myvar] }
{ '$V'(myvar) =.. [Name | Args] }
```

(generating)

```
Name = '$V', Args = [myvar]
```

BiMetaTrans Public Interface and Modes of Use

The sole public predicate of BiMetaTrans is `parse_ruleml/3`. It has these modes:

```
parse_ruleml(+AssertItems, +QueryItems, ?XML)   % χ'$V'
parse_ruleml(?AssertItems, ?QueryItems, +XML)   % π_xmL
```

The modes constrain the inputs to fit one of two patterns, each describing a translation direction.

An assumption of `parse_ruleml` not expressed by its modes is that its non-variable inputs are ground, meaning they do not contain free variables.

BiMetaTrans Use Case: An Air Traffic Control KB

We now consider two examples from the NafHornlogEq ATC KB, which we have translated into Prolog/`'$V'`.

The **first example** is a ground fact, one of hundreds describing aircraft `Char`acteristics:

```
aircraftChar(['B763', 186880.06, 156.08, 140.0]).
```

They are, from left to right:

- type
- weight (kilograms)
- wingspan (feet)
- approach speed (knots)

BiMetaTrans Use Case: An Air Traffic Control KB (Cont'd)

BiMetaTrans renders the fact as an `<Atom>` since the generated `<Rel>` element `aircraftChar` $\notin \mathcal{R}$ (recall $\mathcal{R}$-validity):

```
aircraftChar(['B763', 186880.06, 156.08, 140.0]).
```

```
<Atom>
    <Rel>aircraftChar</Rel>
    <Plex>
        <Ind>B763</Ind>
        <Data iso:type="number">186880.06</Data>
        <Data iso:type="number">156.08</Data>
        <Data iso:type="number">140.0</Data>
    </Plex>
</Atom>
```

BiMetaTrans Use Case: An Air Traffic Control KB (Cont'd)

The **second example** is of a rule (truncated from the paper), defining a weight turbulence category (`heavy`) of aircraft according to International Civil Aviation Organization (ICAO):

RuleML/XML

```
<Implies>
  <then>
    <Atom>
      <Rel>icaoCategory</Rel>
      <Var>aircraft</Var>
      <Data iso:type="symbol">heavy</Data>
    </Atom>
  </then>
  <if>
    <And>
      <Atom>
        <Rel>aircraftChar</Rel>
        <Plex>
          <Var>aircraft</Var>
          <Var>kg</Var>
          <repo>
            <Var>rest</Var>
          </repo>
        </Plex>
      </Atom>
      . . .
</Implies>
```

Prolog/'$V'

```
icaoCategory(
    '$V'(
       aircraft
    ),
    heavy
) :-
      aircraftChar(['$V'(aircraft),
                    '$V'(kg)
                    | '$V'(rest)]),
      . . .
```

BiMetaTrans Use Case: An Air Traffic Control KB (Cont'd)

The **third example** is of a query against the rule of the second example, based on the fact of the first example.

The presentation syntax of the query before and after reifying to Prolog/`'$V'`:

ISO Prolog	Prolog/`'$V'`
`?- icaoCategory('B763', Wtc).`	`?- icaoCategory('B763','$V'(wtc)).`

Its serialization syntax after transforming from Prolog/`'$V'` to RuleML/XML:

```
<Query>
  <Atom>
    <Rel>icaoCategory</Rel>
    <Ind>B763</Ind>
    <Var>wtc</Var>
  </Atom>
</Query>
```

Conclusions and Future Work

Two findings in the paper:

- DCGs along with Prolog's intrinsic bidirectionality, enabling split translation, are very effective!
- BiMetaTrans is realized in Scryer Prolog, a developing ISO Prolog system which compactly represents strings as lists of characters ("partial strings"), ideal for large XML strings

Two planned agenda items:

- Automation of the reflection step in split translation pattern
- Composition with PSOA RuleML, creating a translation chain from ISO Prolog via RuleML/XML to PSOA RuleML presentation syntax (for subsets of each)

BiMetaTrans 1.0 and example use/test cases (including those over the ATC KB) are available at:

https://github.com/mthom/scryer-prolog/tree/master/src/examples/bimetatrans

(cf. BiMetaTrans 1.0 Release news at http://ruleml.org)