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

Mark Thom[1], Harold Boley[2], and 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

Abstract. The paper presents BiMetaTrans(Prolog, RuleML), 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. BiMetaTrans, itself written in Prolog, introduces a tighter integration between RuleML and Prolog, which enables the reuse of, e.g., RuleML Knowledge Bases (KBs) and query engines. A Prolog/`'$V'` encoding is defined as the BiMetaTrans translation source-and-target counterpart to RuleML/XML. This 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 BiMetaTrans DCG is explored and an invertibility proof is outlined. BiMetaTrans is exemplified for knowledge representation and querying applied to an Air Traffic Control KB.

1 Introduction

In decentralized systems such as peer-to-peer [1] or multi-agent [2] architectures, peers or agents should be equipped with invertible translators for information exchange so that they can send/receive data and knowledge to/from other peers or agents, on-the-fly, in a round-trippable manner. This paper presents a RuleML-Prolog translator and outlines its invertibility proof.

In Logic Programming (LP) [3], multidirectional algorithms such as the one for `append` have often been used to exemplify the declarative advantages of specifying multiple functions as a single predicate. While most of LP's multidirectional predicates constitute "programming-in-the-small", few Knowledge Bases (KBs) of predicate definitions exist for multidirectional "programming-in-the-large." This paper contributes to filling the gap by presenting an (extensible) 'mid-size' invertible bidirectional translation algorithm, BiMetaTrans(Prolog, RuleML), applicable to (extensible) subsets of ISO Prolog and RuleML. The

source of BiMetaTrans is available[1], consisting of about 1200 lines of code including comprehensive documentation, and can be easily downloaded and run in any ISO Prolog system. The source directory also includes a test suite featuring an ISO Prolog variant of an Air Traffic Control (ATC) KB [4–6] as a use case.

A common sublanguage of RuleML 1.02 (particularly, of NafFologEq, one of its anchor languages)[2] and of ISO Prolog is proposed to become an anchor language, *Negation-as-failure (Naf) Horn logic with Equality* (***NafHornlogEq***), which is motivated by combining the following:

- A practical LP language – often restricted to NafHornlog (and further to NafDatalog) or to HornlogEq – with many implementations and applications.
- A target language for a Naf extension of the graph-relational PSOATransRun [7].
- The interchange language for BiMetaTrans(Prolog/`'$V'`, RuleML/XML) of this paper, where the Eq component is restricted to syntactic equality.

Made bidirectionally translatable between RuleML – a KB-interoperation hub [8] – and Prolog – an actively refined and expanded ISO standard [9] – NafHornlogEq can become the de facto standard for its expressivity level of knowledge representation. For the current NafHornlogEq subset, BiMetaTrans already provides ISO Prolog with the XML format of RuleML, and RuleML/XML with the common presentation syntax of Prolog. This enables the reuse of, e.g., KB libraries, query engines, analysis tools, editors/IDEs, APIs, and composed translators. Subsequently, for increasing subsets, bidirectional, semantics-preserving translations between the two languages will facilitate further interoperation and reuse.

Regarding the metalogical aspect, BiMetaTrans(Prolog, RuleML) is itself written entirely in Prolog, and translates RuleML/XML from and to equivalent Prolog *compounds* (tree-shaped Prolog terms with a *function symbol*, at their root). BiMetaTrans uses a specialized Prolog encoding, Prolog/`'$V'`, a shorthand for "Metalogic Prolog with variable-as-`'$V'`-compound-reification," which is described in Section 3. Its purpose is to permit processing of KBs and queries as ground terms. A rigorous exploration of the concept of metalogical encoding and its applications is found in Section 1.2 of [10].

Once the encoding and the mapping between it and RuleML/XML are given, we will restrict our attention to their bi-translation. This approach sacrifices no expressivity, as programs written in pure Prolog (or more accurately, Hornlog) can be encoded as Prolog/`'$V'` (and vice versa) by replacing variable names with a Prolog compound (resp. the inverse) headed by the function symbol `'$V'`, whose only subterm is the text of the variable name rendered as a symbol (see the translation table of Section 3.1 for details).

[1] `https://github.com/mthom/scryer-prolog/blob/master/src/examples/bimetatrans/`

[2] See penultimate row of `http://deliberation.ruleml.org/1.02/relaxng/#anchor` table. Also note row for NafNegHornlogEq on the sublanguage path to NafFologEq. Our initial version of BiMetaTrans – like Prolog – assumes ordered conjunctions, disjunctions, and rulebases, although the corresponding RuleML/XML `formula` edges with `index="1"`, `index="2"`, ... attributes (`http://wiki.ruleml.org/index.php/Glossary_of_Deliberation_RuleML_1.02#.40index`) are omitted for simplicity.

On the RuleML/XML side, BiMetaTrans targets: (1) the *ifthen-compact form*[3], using `<then>` and `<if>` edges – in this Prolog-aligned order – on `<Implies>` nodes as shown in Figure 1a; (2) the *mapClosure form*[4], omitting – for Prolog alignment – `<Forall>` and `<Exists>` wrappers; (3) a *minified (xmL) rendering* where the XML is stripped of extraneous whitespace, which is not a limitation since RuleML/XML can be rendered as RuleML/xmL without content loss.

The rest of this paper is organized thus: We begin with a primer on Definite Clause Grammars (DCGs), a domain-specific language for context-free parsing and generation in Prolog. Next we explain our Prolog/`'$V'` encoding and its correspondence to various elements of RuleML/XML, restricted to NafHornlogEq. An outline of a formalized proof of the translator's bidirectionality and termination properties under appropriate term groundedness assumptions follows. We then give an overview of the specification of BiMetaTrans(Prolog, RuleML) as a DCG, with examples of bi-translating parts of a core ATC KB, and descriptions of the interoperation achieved. The conclusions also discuss future work.

2 DCGs and Bidirectional Translation

BiMetaTrans is written almost entirely in the domain-specific language of DCGs [11]. To convey how BiMetaTrans achieves its bidirectionality, we describe how DCGs work by detailing how they might be implemented in raw Prolog. Because of their modular compositionality, DCGs can capture entire grammars, groups of their productions, as well as their individual productions.

Prolog source code shares the same internal representation as Prolog data, allowing abstract syntax trees to be directly manipulated at compile time using specially designated Prolog predicates known as *term expansions*. Many Prolog systems support term expansion and often implement DCGs using it.[5] Here is an example of a DCG of BiMetaTrans, before and after term expansion:

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 are component to a specialized Prolog data structure commonly known as the "difference list" [3]. Difference lists are lists with uninstantiated tails. By keeping track of a logical variable bound to the uninstantiated tail, it becomes possible to concatenate terms to difference lists in constant time.

DCGs maintain an underlying difference list of grammar items through two additional arguments in the heads of grammar clauses: one to the head, and one

[3] `http://wiki.ruleml.org/index.php/Specification_of_Deliberation_RuleML_1.02#XSLT-Based_Compactifiers`

[4] `https://wiki.ruleml.org/index.php/Glossary_of_Deliberation_RuleML_1.02#.40mapClosure`

[5] `https://www.metalevel.at/prolog/dcg#implementation`

to the as-yet uninstantiated tail. The variable `_A` is bound to the difference list starting with the character `'-'` and terminating at the tail variable `_B`.

To see how 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).`

In the post-expansion body of `conditions`, `condition` and `conditions` are both expanded as calls to the named grammar rules. 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 caller 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.

The underlying list passed among the DCGs of BiMetaTrans is always a RuleML/XML string that is either parsed or generated, depending on the translation direction. The bi-translatability of Prolog/`'$V'` terms is addressed via the *split translation pattern*, a technique of reflecting a parsing strategy across translation boundaries. The reflected strategy is expected to perform the inverse translation of the counterpart element. Small changes to the reflected strategy are sometimes needed, but in general, the pattern is highly effective.

The pattern is applied by first writing a uni-translation of a RuleML/XML element, after checking that `Item` is unbound. Here, we give the first half of the `ruleml_atom` DCG production, before applying split translation, with an 'if' part to the left of "`->`" and a 'then' part in the lines below it:

```
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] }
    ;   . . .
    ).
```

The variable `Item` stands for a Prolog/`'$V'` term, since it is specified as an explicit grammar argument. `var(Item)` succeeds when `Item` is a free variable, which BiMetaTrans takes as proof that it is translating from RuleML/XML to Prolog/`'$V'`. The atom's name and arguments are bound to the variables `Name` and `Args`. The extraction of those data from the implicit XML string will be explained in later sections. Central to the split translation pattern is the observation that the predicate "`=..`", as with many other ISO Prolog built-ins, is itself bidirectional. It presupposes that one of its arguments contains enough information to determine the other. In one direction, the list has the function symbol of the compound at its head, followed by a list of its arguments at its tail: the `Name` and `Args` of the structure, in this context. Conversely, if `Name`

and `Args` are specified, as they are in the first half of `ruleml_atom`, the Prolog compound just described is bound to `Item` on the left-hand side.

Therefore, to apply the pattern in the second half, we reflect the translation of the 'then' part from the first half across the boundary of translation, here (and everywhere in BiMetaTrans) signified by a "`;`" for the 'else' part:

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

In the above reflected (inverse) direction of translation, `Name` and `Args` are extracted from the specified Prolog/`'$V'` term `Item`, and control is threaded to the bidirectional DCG productions `prolog_symbol` and `ruleml_items`. Some small differences are present in the reflected translation: `list_ws` is replaced by `list`, which doesn't match (or, for that matter, generate) trailing whitespace as `list_ws` does. Also, the order of the XML element matching was not changed in the reflected translation.

3 The Prolog/'$V' Bidirectional Metalogical Translation

Prolog/`'$V'` KBs are encoded according to a convention that reifies RuleML variables by name. Initially, it may seem simpler to translate RuleML variables directly, to and from Prolog variables, but we soon discover a problem. In many popular Prolog implementations, including those in which BiMetaTrans was tested, the variable names are never stored. In fact, once variables have been rendered to the Prolog heap, their textual, source-level names are promptly discarded and forgotten by the Prolog engine.

Therefore, if we want fully invertible translation, we must store variable names in terms, so they can be recovered by an inverse translation, thus completing a round-trip. As detailed in a row of the translation table of Section 3.1, BiMetaTrans *reifies* RuleML variable names to Prolog symbols wrapping them each in a Prolog compound with the function symbol `'$V'`. The leading "`$`" was so chosen because atoms with leading "`$`"'s are seldom used by Prolog programmers, making `'$V'` an unlikely source of name clash.

3.1 The BiMetaTrans Translation Table and Syntax Graph

We define the mapping $\chi_{\texttt{'\$V'}}$ as translating from Prolog/'$V' to the equivalent RuleML/XML elements and the mapping $\pi_{\texttt{xmL}}$ as translating in the inverse direction. The aim of this section is to outline a proof showing that $\chi_{\texttt{'\$V'}} \circ \pi_{\texttt{xmL}} = id_{\texttt{xmL}}$ and $\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}} = id_{\texttt{'\$V'}}$; in words, both compositions yield the identity function.

The recursive definition of $\chi_{\texttt{'\$V'}}$ is given in the table below, similar in style to the tables in [12]. We should stress that, while the translator uses minified XML

(or xmL, as in the mapping subscripts), the second column of the table uses indented XML for readability (see Section 1 for a definition of minified XML).

Prolog/'$V' Syntax	**RuleML/xmL**
[*assertitem*$_1$, . . . , *assertitem*$_n$]	<Assert mapClosure="universal"> $\chi_{\text{'\$V'}}$(*assertitem*$_1$) . . . $\chi_{\text{'\$V'}}$(*assertitem*$_n$) </Assert>
?- queryitem	<Query mapClosure="existential"> $\chi_{\text{'\$V'}}$(*queryitem*) </Query>
(*conjunct*$_1$, . . . , *conjunct*$_n$)	<And> $\chi_{\text{'\$V'}}$(*conjunct*$_1$) . . . $\chi_{\text{'\$V'}}$(*conjunct*$_n$) </And>
(*disjunct*$_1$; . . . ; *disjunct*$_n$)	<Or> $\chi_{\text{'\$V'}}$(*disjunct*$_1$) . . . $\chi_{\text{'\$V'}}$(*disjunct*$_n$) </Or>
consequent :- antecedent	<Implies> <then>$\chi_{\text{'\$V'}}$(*consequent*)</then> <if>$\chi_{\text{'\$V'}}$(*antecedent*)</if> </Implies>
pred (*argument*$_1$, . . . *argument*$_n$)	<Atom> <Rel>$\chi_{\text{'\$V'}}$(*pred*)</Rel> $\chi_{\text{'\$V'}}$(*argument*$_1$) . . . $\chi_{\text{'\$V'}}$(*argument*$_n$) </Atom>
func (*argument*$_1$, . . . *argument*$_n$)	<Expr> <Fun>$\chi_{\text{'\$V'}}$(*func*)</Fun> $\chi_{\text{'\$V'}}$(*argument*$_1$) . . . $\chi_{\text{'\$V'}}$(*argument*$_n$) </Expr>
left = right	<Equal> $\chi_{\text{'\$V'}}$(*left*) $\chi_{\text{'\$V'}}$(*right*) </Equal>

continued on next page

continued from previous page	
`\+` *form*	`<Naf>` $\chi_{\texttt{'\$V'}}$(*form*) `</Naf>`
`"`*prologstring*`"`	`<Data` `iso:type="`*string*`">`*prologstring*`</Data>`
prologstring	`<Data` `iso:type="`*number*`">`*prologstring*`</Data>`
prologcharseries	`<Data` `iso:type="`*symbol*`">`*prologcharseries*`</Data>`
`'`*capitalizedprologcharseries*`'`	`<Ind>`*capitalizedprologcharseries*`</Ind>`
`'$V'(`*prologcharseries*`)`	`<Var>`*prologcharseries*`</Var>`
`[`*argument*$_1$`, ..., `*argument*$_n$`]`	`<Plex>` $\chi_{\texttt{'\$V'}}$(*argument*$_1$) . . . $\chi_{\texttt{'\$V'}}$(*argument*$_n$) `</Plex>`
`[`*argument*$_1$`, ..., `*argument*$_n$ `|` *repo*`]`	`<Plex>` $\chi_{\texttt{'\$V'}}$(*argument*$_1$) . . . $\chi_{\texttt{'\$V'}}$(*argument*$_n$) `<repo>` $\chi_{\texttt{'\$V'}}$(*repo*) `</repo>` `</Plex>`

prologstring and ***prologstring*** are expected to conform to the lexical grammar of numbers and strings as defined in the ISO Prolog standard [9]. As such, $\chi_{\texttt{'\$V'}}$ is not applied to them. A similar constraint applies to ***prologcharseries***, which is assumed to start with a lower case letter followed by a sequence of alphanumeric characters. ***capitalizedprologcharseries*** differs only in that the first letter must be upper case. As shown in the table, Prolog's convention of distinguishing variables from symbols by capitalizing the former, in BiMetaTrans becomes a convention of RuleML distinguishing `Ind`ividuals from `<Data>`[6]. A ***capitalizedprologcharseries*** may also represent a symbol, in which case it is printed between single quotes so as not to be mistaken for a variable.

Most elements of RuleML/XML below the root element `<RuleML>`[7] are constrained to appear as children of certain container elements. The two top-most container elements are the performatives `<Assert>` and `<Query>`. `<Assert>` issues new facts and rules to the underlying RuleML KB while `<Query>` posts queries to its query engine. `<Assert>` makes an implicit `<Rulebase>` assumption, containing `<Assert>`'s children.

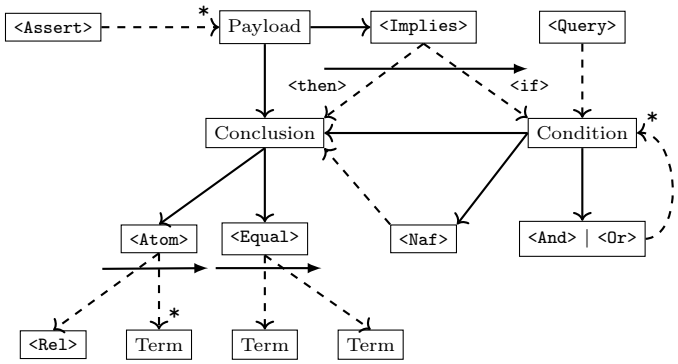
To indicate the main RuleML/XML elements' parent-child relations, and as an overview of how the translator DCG is built up from smaller DCGs, we present the grammar of the NafHornlogEq dialect of RuleML/XML as a syntax graph

[6] Complementing RuleML's `xsi:type`, we introduce `iso:type` for ISO Prolog types.

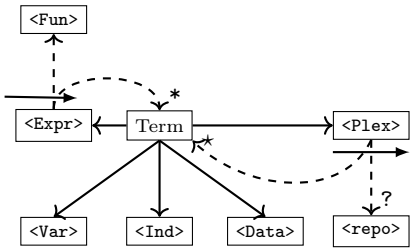
[7] The `<RuleML>` root is assumed by BiMetaTrans(Prolog/`'$V'`, RuleML/XML), thus simplifying its various (KB-only, query-only, KB&query) translation uses.

in Figure 1a. The supplemental syntax graph in Figure 1b elaborates Figure 1a's Term box. Taken together, they comprise a single graph describing the grammar as a whole.

In each graph, boxes labeled by XML elements map one-to-one to the rows of the translation table. Solid arrows visualize the choice operator in DCGs and EBNF, indicating that an origin box chooses any of its destination boxes. Dashed arrows originate from boxes with angular bracketed text, which are always XML elements; consequently, their destination boxes are the children of their origin box's parent XML element. The children are said to be sequenced if they are meant to appear in a prescribed left-to-right order. Sequencing is denoted in the syntax graphs by sequencers, horizontal arrows with rectangular heads overlaying each of the dashed arrows fanning out from an origin.



(a) The top-level of the syntax graph.



(b) The term syntax subgraph.

Fig. 1: RuleML/XML elements and their children.

Other boxes are labeled by intermediate nonterminals. In Figure 1a, a solid (choice) arrow points from the intermediate Condition box to the intermediate Conclusion box. It entails that the `<Atom>` and `<Equal>` elements, the children of the Conclusion box, are also admissible as conditions in the grammar.

Choice arrowheads are in some cases marked by a "`?`" or "`*`" modifier, meaning that the corresponding XML parent can have, respectively, an optional child or zero or more occurrences of its destination as children. Sequenced arrows may also be marked by the "`?`" or "`*`" modifiers.

Lastly, there are two cases of labeled choice arrows in the composite syntax graph. In RuleML/XML, the consequent and antecedent children of `<Implies>` must be wrapped by `<then>` and `<if>` elements, in that order. The ordering reflects the Prolog convention for writing implications / inference rules using the infix operator "`:-`", which places the consequent on its left-hand side and the antecedent on the right.

3.2 An Invertibility Proof Outline

We now consider the invertibility property: $\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}} = id_{\texttt{'\$V'}} \wedge \chi_{\texttt{'\$V'}} \circ \pi_{\texttt{xmL}} = id_{\texttt{xmL}}$ ("$\circ$" denotes function composition and id_X the identity function on set X).

The following definitions will be needed. Let $\mathcal{R} = \{$ `=`, `:-`, `\+`, `'$V'`, `'.'` $\}$ be the set of *reserved symbols* and $\mathcal{S} \supset \mathcal{R}$ the set of *ISO Prolog symbols*. A RuleML/XML document is $\mathcal{R}$-*valid* if its atoms never contain a member of $\mathcal{R}$ in their `<Rel>` elements. Note that $\mathcal{R}$-validity applies to all document parts, e.g. to RuleML/XML KBs (rooted in `<Assert>`) and queries (rooted in `<Query>`).

With these definitions in place, we outline the invertibility proof, having established the translation table and syntax graph to help us present the proof as a case analysis. We will partly show that $\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}} = id_{\texttt{'\$V'}}$, leading to the analysis of an exemplary case.

A valid RuleML/XML KB or query can be viewed as a traversal tree of the syntax graph, all of whose leaves correspond to grammar terminals. The traversal tree must have either the `<Assert>` or `<Query>` box as its root. $\chi_{\texttt{'\$V'}}$ is defined on the set of subtrees of all valid traversal trees, and so, we outline an inductive proof on the height of a given traversal tree. Since each traversal tree is finite, the proof will implicitly show that BiMetaTrans terminates on valid inputs.

$\mathcal{R}$-validity requires that RuleML/XML atoms named under the `<Rel>` elements of `<Atom>` elements are not members of $\mathcal{R}$. This constraint is vital to the invertibility of BiMetaTrans. For example, suppose we naïvely translated this (minified) RuleML/xmL atom:

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

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

```
'='('$V'(x),'$V'(y))  ~  '$V'(x) = '$V'(y)
```

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

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

which is not the original XML. The mistranslation occurs because plain Prolog compounds are the targets of both the `Atom` and `Equal` elements, leaving $\chi_{\texttt{'\$V'}}$ unable to tell which of the two was the source. Therefore, we see that disallowing RuleML atoms with a `Rel` element value of "`=`" is necessary to avoid ambiguity. In effect, the Prolog symbol "`'='`" can only be used for the translation of `Equal` elements.[8]

[8] A conclusion compound with function symbol `'='` is not permissible in an ISO Prolog KB, but we are concerned only with Prolog/`'$V'` encodings as ground terms.

The leaves `<Var>`, `<Ind>`, and `<Data>` are the subjects of the induction basis. We have already seen an example translation sending `<Var>` to a "`'$V'`" compound and back. Since `'$V'` $\in \mathcal{R}$, this can be done at no information loss. `<Ind>` is more subtle in that on the RuleML/XML side, we suppose that all `Ind`ividuals are capitalized strings. We can represent capitalized strings as Prolog symbols under the requirement that they be printed in single quotes to avoid confusion with variable names, which are always unquoted, capitalized strings. Similarly, the contents of `<Data>` are recognized on both translation sides by unique lexical characteristics that need no special effort to be distinguished from other forms.

For the induction hypothesis (IH), we assume that invertibility holds for all trees of height bounded by some $h \geq 1$. BiMetaTrans uses two recursive patterns based on the outgoing arrow types of the tree's root box. We outline the IH by demonstrating the invertibility of a single instance of each.

We focus on the `<Implies>` element of the case analysis. `<Implies>` has two outgoing sequenced arrows to its child edge elements `<then>` and `<if>`. Referring to the translation table, we see that `<Implies>` elements are translated as Prolog compounds headed by `:-` $\in \mathcal{R}$. First, we want to show this equation:

$$\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}}(\boldsymbol{then} \texttt{ :- } \boldsymbol{if}) \;=\; \pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}}(\boldsymbol{then}) \texttt{ :- } \pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}}(\boldsymbol{if}) \tag{1}$$

Consulting the translation table, we find a row permitting this rewrite:

$$\begin{aligned} \chi_{\texttt{'\$V'}}(\boldsymbol{then} \texttt{ :- } \boldsymbol{if}) \;\rightsquigarrow\; & \texttt{<Implies>} \\ & \texttt{<then>}\chi_{\texttt{'\$V'}}(\boldsymbol{then})\texttt{</then>} \\ & \texttt{<if>}\chi_{\texttt{'\$V'}}(\boldsymbol{if})\texttt{</if>} \\ & \texttt{</Implies>} \end{aligned}$$

Then, $\pi_{\texttt{xmL}}$ performs the inverse transformation, establishing equation (1). The IH entails

$$\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}}(\boldsymbol{then}) \;=\; \boldsymbol{then}$$

and similarly by swapping ***if*** for ***then***, completing the argument for this case.

For recursion based on arrows of the second type, we consider the case of the Condition box. The Condition box in Figure 1a does not directly pertain to a RuleML/XML element, but chooses one among `<Atom>`, `<Equal>`, and `<Naf>`. Any of the three descends into a subtree of height $< h$, so we use the IH.

Proofs of the invertibility of other elements can be similarly obtained, with some modifications for variable numbers of children or an optional child, completing the $\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}}$ direction. The proof of $\chi_{\texttt{'\$V'}} \circ \pi_{\texttt{xmL}} = id_{\texttt{xmL}}$ is symmetric.

4 A Guided Tour of BiMetaTrans via ATC KB Examples

In this section we walk through the implementation of BiMetaTrans as a Prolog DCG. We will follow the exposition with examples of translating an ISO Prolog variant[9] of a POSL ATC KB to RuleML/XML.

[9] `http://users.ntua.gr/mitsikas/ATC_KB/ATC_KB_ISO_PL.pl`

The sole public predicate of BiMetaTrans is `parse_ruleml/3`. Its first two arguments are Prolog-content lists; the third is a string containing a RuleML/XML serialization (parsed by BiMetaTrans as RuleML/xmL 'on-the-fly') consisting of an (optional) assertion followed by (optional) queries. It has these modes:

```
parse_ruleml(+AssertItems, +QueryItems, ?XML)        %  Realizes χ_'$V' of Section 3
parse_ruleml(?AssertItems, ?QueryItems, +XML)        %  Realizes π_xmL of Section 3
```

The modes constrain the inputs to fit one of two patterns, each describing a translation direction. The first, where `AssertItems` and `QueryItems` are instantiated and `XML` is possibly a free variable, describes translation from Prolog/`'$V'` to RuleML/xmL. The other direction is captured by the second mode, where the (un)instantiation conditions are swapped: a RuleML/xmL string is assigned to `XML` while `AssertItems` and `QueryItems` may be unbound. A third assumption of `parse_ruleml` not expressed by its modes is that its inputs, when instantiated, are ground, meaning they do not contain free variables. These assumptions help ensure that BiMetaTrans is deterministic and will terminate on all valid inputs.

The possible children of `<Assert>` elements are disjunctively connected by

```
ruleml_assert_item(Item) -->
  ruleml_implies(Item) | ruleml_equal(Item) | ruleml_atom(Item).
```

using the "`|`" operator, which realizes the *choice operator* in the EBNF metasyntax: In DCGs, "`|`" causes its operands to be backtracked over from left to right until one succeeds. Thus, the `<Implies>`, `<Equal>`, and `<Atom>` elements are valid children of the `<Assert>` element, as seen in the translation table.

The remaining boxes of the syntax graph/rows of the translation table are similarly represented by other DCGs. For container elements with one or more children, the "`*`"s of the syntax graph, we have – for the greedy consumption of children – patterns such as the following:

```
ruleml_assert_items([Item | Items]) -->
    ruleml_assert_item(Item),
    ruleml_assert_items(Items).
ruleml_assert_items([]) --> [].
```

Here, children of an `<Assert>` fill the argument list until an `</Assert>` is met.

We now turn to the NafHornlogEq ATC KB[9], which we have translated into Prolog/`'$V'` (see the end of this section). Its purpose is to help determine the separation minima of pairs of aircraft according to various ATC regulations.

The first clause is a ground fact describing aircraft `Char`acteristics, from left to right – type, weight (kilograms), wingspan (feet), and approach speed (knots):

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

BiMetaTrans renders the fact as an atom because its head `aircraftChar` $\notin \mathcal{R}$:

```
<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>
```

The aircraft `Char`'s are collected in a single `<Plex>`, specialized to a RuleML/XML counterpart for Prolog's list data type. Each `<Data>` element is matched by the `ruleml_data` DCG. The contents of `<Data>` elements are matched and generated by the following `ruleml_data_contents` grammar with three disjoint cases:

```
ruleml_data_contents(number, Cs) -->
    ruleml_number(Cs).
ruleml_data_contents(symbol, Cs) -->
    ruleml_symbol(Cs).
ruleml_data_contents(string, Cs) -->
    ruleml_string(Cs).
```

The `ruleml_data_contents` DCG notably breaks from the "`|`" convention of other choice-driven DCGs: the first argument is used to pass or receive type information to/from the caller, which is (resp.) generated as the "`iso:type`" attribute or used to index the clauses of `ruleml_data_contents`, depending on the translation direction.

The preceding grammars match disjoint character sequences. Consequently, numbers, strings, and symbols are inlined into Prolog/`'$V'` compounds without annotation. "`'B763'`" is a ***capitalizedprologcharseries*** that is not wrapped in a "`$V`", and so is considered an `Ind`ividual.

A more advanced example is found in a rule for categorizing an aircraft in its wake turbulence category, according to regulations set by the International Civil Aviation Organization (ICAO). The original NafHornlogEq rule in ISO Prolog is given on the left, and the encoded Prolog/`'$V'` source is on the right:[10]

ISO Prolog	Prolog/'$V'
`icaoCategory(Aircraft, heavy) :-` `  aircraftChar([Aircraft, Kg | Rest]),` `  greaterThanOrEqual(Kg, 136000.0),` `  \+ icaoCategory(Aircraft, super).`	`icaoCategory('$V'(aircraft), heavy) :-` `  aircraftChar(['$V'(aircraft), '$V'(kg)` `  \| '$V'(rest)]),` `  greaterThanOrEqual('$V'(kg), 136000.0),` `  \+ icaoCategory('$V'(aircraft), super).`

In the following "co-alignment" and "DCG" tables, circle superscripts co-reference between the children of `Naf` ("1") and `Var` ("2") as well as the source ("black") and target ("white"). While in the co-alignment table, the source is on the right of the target, in the DCG table, the source is below the target.

For the co-alignment table, recalling the `<Implies>` box in the syntax graph of Figure 1a, we see that `<then>` precedes `<if>`. The child of `<if>` is an `<And>` containing, as its first child, an `aircraftChar` `<Atom>` with a single `<Plex>` argument. This example's use of `<Plex>` contains an optional (EBNF's "`?`") child `<repo>` (Prolog's "`|`"). A `<repo>` must be either a `<Var>` or another `<Plex>`. This distinction is realized by a structural check. The second `<And>` child is a `greaterThanOrEqual` atom. The third conjunct is a Naf that contains an `icaoCategory` atom.

For the DCG table, notice that the `<Naf>` and `<Var>` elements each contain one element, albeit with different children.

[10] In the Emacs text editor, the reification can be done by the interactive command `M-x query-replace-regexp` with basically `\([A-Z][a-z]*\)` as the matching expression and `'$V'(\,(downcase \1))` as the transform. Such a command could also be named as an Emacs Lisp function.

The targeted RuleML/XML is on the left while the Prolog/`'$V'` rule has been co-aligned on the right, where its tree structure is made more explicit:

RuleML/XML	Prolog/'$V'
`<Implies>` `  <then>` `  <Atom>` `  <Rel>icaoCategory</Rel>` `  <Var>aircraft`②`</Var>` `  <Data iso:type="symbol">heavy</Data>` `  </Atom>` `  </then>` `  <if>` `  <And>` `  <Atom>` `  <Rel>aircraftChar</Rel>` `  <Plex>` `  <Var>aircraft`②`</Var>` `  <Var>kg`②`</Var>` `  <repo>` `  <Var>rest`②`</Var>` `  </repo>` `  </Plex>` `  </Atom>` `  <Atom>` `  <Rel>greaterThanOrEqual</Rel>` `  <Var>kg`②`</Var>` `  <Data iso:type="number">136000.0</Data>` `  </Atom>` `  <Naf>` `  <Atom>`① `  <Rel>icaoCategory</Rel>` `  <Var>aircraft`②`</Var>` `  <Data iso:type="symbol">super</Data>` `  </Atom>` `  </Naf>` `  </And>` `  </if>` `</Implies>`	`icaoCategory(` `  '$V'(` `  aircraft`② `  ),` `  heavy` `) :- aircraftChar([` `  '$V'(` `  aircraft`② `  ),` `  '$V'(` `  kg`② `  )` `  \|` `  '$V'(` `  rest`② `  )]),` `  greaterThanOrEqual(` `  '$V'(` `  kg`② `  ),` `  136000.0,` `  ),` `  \+ icaoCategory(`① `  '$V'(` `  aircraft`② `  ),` `  super` `).

For the DCGs, circle-superscripted parts of `ruleml_naf` and `ruleml_var` translate the circle-superscripted data of the co-alignment table:

DCG production for Naf	DCG production for Var
`ruleml_naf(I) -->` `  ( { var(I) } ->` `  list_ws("<Naf>"),` `  ruleml_condition(NI)`①`,` `  { Item = (\+ NI) },` `  list_ws("</Naf>")` `  ; "<Naf>",` `  { Item = (\+ NI) },` `  ruleml_condition(NI)`①`,` `  "</Naf>"` `).	`ruleml_var(V) -->` `  ( { var(V) } ->` `  list_ws("<Var>"),` `  ruleml_var_contents(VCs)`②`,` `  { atom_chars(VN, VCs) },` `  { V = '$V'(VN) },` `  list_ws("</Var>")` `  ; "<Var>",` `  { V = '$V'(VN) },` `  { atom_chars(VN, VCs) },` `  ruleml_var_contents(VCs)`②`,` `  "</Var>"` `).

As a final example, we consider a query against the ATC KB in ISO Prolog[9]. Here is its presentation syntax before and after reifying[10] to Prolog/`'$V'`:

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

This is the translation of the second column to (minified) RuleML/xmL:

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

The only new element introduced by this case is `<Query>`. Note that, unlike the `<Assert>` performative, `<Query>` does not implicitly contain a `<Rulebase>`.

When posed to either a RuleML/XML or ISO Prolog engine with a KB of just the sample fact and rule, this query will succeed, binding its variable `<Var>wtc</Var>` or `Wtc` to the symbol `heavy`.

5 Conclusions

This paper presents BiMetaTrans, an invertible bidirectional metalogical translator between RuleML/XML and Prolog/`'$V'`. The latter is an encoding used to represent the complete contents of RuleML/XML KBs/queries as Prolog terms.

The case for writing invertible translators in DCGs is bolstered by comparing the simplicity and compactness of BiMetaTrans to recent approaches of the functional programming community in this area [13–15]. DCGs natively harness the declarativity of Prolog, making them at once simple, efficient, and effective.

The grammar of the NafHornlogEq RuleML language, and its translation to Prolog/`'$V'`, were explained in a syntax graph and translation table, respectively. We gave a proof outline of BiMetaTrans' invertibility, focusing on the $\pi_{\texttt{xmL}} \circ \chi_{\texttt{'\$V'}}$ direction, and examined its operation on a NafHornlogEq ATC KB.

Future work includes extending the RuleML/XML use of `<Equal>` from Prolog's "=" for syntactic unification to its `is` primitive for (arithmetic) functional built-ins, e.g. as in PSOATransRun's Prolog conversion. BiMetaTrans could also be extended to NafHornlogEq RuleML superlanguages, e.g. adding strong Negation for the anchor NafNegHornlogEq and Disjunctive conclusions for a NafDisHornlogEq. Moreover, we intend to explore ways of automating the reflection step of the split translation pattern.

BiMetaTrans was created and is being developed in Scryer Prolog[11], an ISO Prolog system under development by the first author. Its unique features include "partial strings"[12], which provide a 24-fold reduction in memory usage over how strings are typically represented in Prolog systems. Partial strings pack characters in UTF-8 format, but act as difference lists of characters. This allows their use in DCGs, which, combined with their compact representation, makes them well-suited to generating/parsing large (RuleML/)XML KBs/queries as strings.

BiMetaTrans could be composed with the PSOA RuleML API [16], creating a translation chain from ISO Prolog via RuleML/XML to PSOA RuleML presentation syntax (for subsets of each).

[11] `https://github.com/mthom/scryer-prolog`

[12] `https://github.com/mthom/scryer-prolog#strings-and-partial-strings`

References

1. Iamnitchi, A., Trunfio, P., Ledlie, J., Schintke, F.: Peer-to-peer computing. In: Euro-Par 2010 - Parallel Processing, 16th Int'l Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I. Volume 6271 of Lecture Notes in Computer Science., Springer (2010) 444–445
2. Kravari, K., Bassiliades, N., Boley, H.: Cross-Community Interoperation Between Knowledge-Based Multi-Agent Systems: A Study on EMERALD and Rule Responder. Expert Syst. Appl **39**(10) (2012) 9571–9587
3. Sterling, L., Shapiro, E.Y.: The art of Prolog: advanced programming techniques. (1994)
4. Mitsikas, T., Stefaneas, P., Ouranos, I.: A Rule-Based Approach for Air Traffic Control in the Vicinity of the Airport. In: Algebraic Modeling of Topological and Computational Structures and Applications, Springer Int'l Publishing (2017) 423–438
5. Mitsikas, T., Almpani, S., Stefaneas, P., Frangos, P., Ouranos, I.: Formalizing Air Traffic Control regulations in PSOA RuleML. In: Proc. Doctoral Consortium and Challenge@ RuleML+ RR 2018 hosted by 2nd Int'l Joint Conference on Rules and Reasoning. Volume 2204., CEUR Workshop Proceedings (2018)
6. Deryck, M., Mitsikas, T., Almpani, S., Stefaneas, P., Frangos, P., Ouranos, I., Boley, H., Vennekens, J.: Aligning, interoperating, and co-executing air traffic control rules across PSOA RuleML and IDP. In Fodor, P., Montali, M., Calvanese, D., Roman, D., eds.: Rules and Reasoning, Springer Int'l Publishing (2019) 52–66
7. Boley, H., Zou, G.: Perspectival Knowledge in PSOA RuleML: Representation, Model Theory, and Translation. CoRR **abs/1712.02869, v3** (2019)
8. Boley, H.: The RuleML Knowledge-Interoperation Hub. In Alferes, J.J., Bertossi, L.E., Governatori, G., Fodor, P., Roman, D., eds.: Rule Technologies. Research, Tools, and Applications - 10th Int'l Symposium, RuleML 2016, Stony Brook, NY, USA, July 6-9, 2016. Proceedings. Volume 9718 of Lecture Notes in Computer Science., Springer (2016) 19–33
9. ISO: ISO/IEC 13211-1:1995: Information technology — Programming languages — Prolog — Part 1: General core. (1995)
10. Basin, D.A., Constable, R.L.: Metalogical frameworks. In: Proceedings of the Second Annual Workshop on Logical Frameworks, Edinburgh, UK (June 1991)
11. Pereira, F., Warren, D.H.D.: Definite clause grammars for language analysis – a survey of the formalism and a comparison with augmented transition networks. Artificial Intelligence **13** (1980) 231–278
12. Boley, H.: RIF RuleML Rosetta Ring: Round-Tripping the Dlex Subset of Datalog RuleML and RIF-Core. In Governatori, G., Hall, J., Paschke, A., eds.: RuleML. Volume 5858 of Lecture Notes in Computer Science., Springer (2009) 29–42
13. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. SIGPLAN Not. **45**(11) (September 2010) 1–12
14. Duregård, J., Jansson, P.: Embedded parser generators. Volume 46. (12 2011) 107–117
15. Matsuda, K., Wang, M.: Flippr: A prettier invertible printing system. Volume 7792. (03 2013) 101–120
16. Al Manir, M.S., Riazanov, A., Boley, H., Baker, C.J.O.: PSOA RuleML API: A Tool for Processing Abstract and Concrete Syntaxes. In Bikakis, A., Giurca, A., eds.: Rules on the Web: Research and Applications, Proc. 6th International Symposium, RuleML 2012, Montpellier, France. Volume 7438 of Lecture Notes in Computer Science., Springer (August 2012) 280–288