On Interchange between Drools and Jess

Oana Nicolae, Adrian Giurca and Gerd Wagner
Brandenburg University of Technology, Germany
E-mail: {nicolae, giurca, G.Wagner}@tu-cottbus.de

Keywords: Drools (aka JBossRules), Jess, RuleML, R2ML, RIF, Rete, ReteOO, business rules, interchange, standardisation

Received: March 15, 2008

There is a growing demand for research in order to provide insights into challenges and solutions based on business rules, related to target PSMs (Platform Specific Model in OMG's MDA terms - Implementation Model). As an answer to these needs, the paper argues on the relevance of business rules target platforms for the actual IT and business context, by emphasising the important role of business rules interchange initiatives. Therefore, the rule-system developers can do their work without any concern about a vendor-specific format, and in particular without any concern about the compatibility between the technologies. The paper provides a description of the business rules translation from a particular object oriented rule-system such as Drools, to another rule-system as Jess coming from the AI area, using R2ML as interchange language. The transformation preserves the semantic equivalence for a given rule set, taking also into account the rules vocabulary.

Povzetek: Prispevk opisuje prenos pravil iz objektnega sistema Drools v AI sistem Jess.

1 Introduction

There is a growing request for business rules technology standardisation from both UML and ontology architects communities. Due to these reasons, business rules aim to express rules in a platform independent syntax.

A number of initiatives on rules interchange have been started. They include the RuleML (2), OMG Production Rules Representation (PRR) (8), RIF (1), and the REWERSE I1 Rule Markup Language (R2ML[1]) (10). We mention here the efforts to establish some standards for expressing business rules and their vocabularies in natural language such as OMG's SBVR (9) and Attempto Controlled English (ACE) (4). SBVR, this human readable format of business rules comes under OMG's Model Driven Architecture (MDA[2]) standards and is defined as Computation-Independent Model (CIM[3]). CIM is most frequently used in the context of the Model Driven Architecture (MDA) approach which corresponds the Object Management Group (OMG) vision of Model Driven Engineering (MDE). The Meta-Object Facility (MOF), is the OMG standard for Model Driven Engineering.

The second layer in OMG's MDA is Platform-Independent Model (PIM)[4] where rule interchange formats (i.e. RuleML, RIF, R2ML) try to accomplish their general purpose: a PSM to PSM business rules migration through the PIM level. The third MDA level is Platform-Specific Model (PSM[5]) containing rule specific languages together with their specific engines/platforms like: F-Logic (5), JRules(ILOG[6]), Jess[7] or Drools[8].

The main purpose of an interchanging approach is to provide means for reusing, publication and interchange of rules between different systems and tools. Actually, it also plays an important role in facilitating business-to-customer (B2C) and business-to-business (B2B) interactions over the Internet. Moreover, an interchange approach always supposes less transformations than PSM-to-PSM translations.

Our rule interchange work addresses Drools as source platform and Jess as a target platform, using the approach suggested by OMG's MDA, because these languages are actually in business market interest as popular business logic frameworks, used by Java developers to create complex rule-based applications by combining Java platform and business rule technology. Another reason for choosing these two rule systems is their efficiency in "pattern" matching, especially to handle updates to its working set of facts, as both Drools and Jess use an algorithm known as the Rete (i.e. Latin for "net") algorithm. Computational complexity per iteration of this algorithm is linear in the size of the fact base.

The main standardisation communities, OMG[9] and W3C[10] focus their work on providing business rules specification languages for all MDA layers of models in order

[1] R2ML - `http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/6`

[2] MDA - Model Driver Architecture is a framework for distinguishing different abstraction levels defined by the Object Management Group.

[3] CIM - Computational Independent Model

[4] PIM - Platform Independent Model

[5] PSM - Platform Specific Model

[6] *ILog*, `http://www.ilog.com`

[7] *Jess*, `http://herzberg.ca.sandia.gov/jess/`

[8] *JBossRules*, `http://labs.jboss.com/jbossrules/`

[9] OMG - http://www.omg.org/

[10] W3C - http://www.w3.org/

to obtain rules interchange. Their standards are not sustained by most of business rules management system tools, as they implement proprietary rule languages. The reasons for this situation imply the existence of only a few interchange works in the academia i.e. RIF (1) language still has no well defined guidelines of how to implement the transformations and it also does not specify how to test the correction of the translation.

In this context, EU network of Excelence REWERSE[11] developed R2ML as an interchange language for deploying and sharing rules between different rule systems and tools (e.g. Object Oriented rule languages, Semantic Web rule languages, Artificial Intelligence rule languages). Actually, R2ML (now at version 0.5) is a mature and experienced enough rule interchange language to provide a concrete interchange format for different rule systems and languages (i.e.`http://oxygen.informatik.tu-cottbus.de/rewerse-i1/?q=node/15`).

R2ML has a rich syntax, so it can represent business rules from both Drools and Jess languages, providing this way the interchange possibility. As an interchange language, R2ML addresses the PIM level. The main idea is to use a model transformation language (MTL), or an application transformation language (ATL) to transform a PIM model into a PSM as in the Figure 1.

Business rules are built following a business model representation. In many cases, a business model is first represented in a natural language description based on core ontologic concepts like classes and variables (OMG's MDA - CIM level).

At this stage, we can identify all objects referenced in the rules, and for each object we identify all referenced properties. For each property, we identify all its constraints.

2 Drools to R2ML mapping

In this section we describe the general JBoss business rules transformation into R2ML interchange language. Drools engine project, (now at version $4.0.x$) is an open source and standards-based business rule engine and it uses an enhanced implementation named ReteOO[12].

Drools is classified as an Object-Oriented Production Rules engine written entirely in Java language, and more specifically it is a Forward-Chaining rule engine.

A Production Rules System (i.e. PRS) relies on an Inference Engine that is able to scale to a large number of rules and facts. The Inference Engine matches facts and data, against PRs, also called Productions or just Rules, to infer conclusions which result in actions. The Rules are stored in the Production Memory. The facts that the Inference Engine matches against the rules are stored in the Working Memory.

R2ML is a visual rule markup, XML-based language, whose purpose is to capture rules formalised in different languages and to interchange them between rule systems and tools. It provides support for all kind of rules:

- Integrity Rules
- Derivation Rules
- Production Rules
- Reaction Rules

A R2ML production rule has *conditions* and *post-conditions*. The conditions and post-conditions of a R2ML production rule are usually interpreted as logical formulae which correspond to a general first order formula: quantified formula, existentially quantified or universally quantified (i.e. R2ML uses the concept of `r2ml:QuantifiedFormula` and by default, all R2ML formulae are universally quantified). Usually, PRS does not explicitly refer to events, but events can be simulated in a production rule system by externally asserting corresponding facts into the Working Memory. The R2ML production rules metamodel is depicted in the Figure 2:

The mapping from Drools to R2ML is possible as R2ML supports the representation of the PRs by relying on the OMG's PRR (8) Specification. Following sections describe general principles of mapping from JBoss rules into R2ML PRs.

2.1 Mapping rules vocabularies

Object oriented rules systems as Drools and ILOG JRules are build on top of Java vocabularies. Drools is designed to use Java beans as *facts*. These facts represent the domain of the rules, meaning the rules vocabulary. Java beans objects are defined by users in their applications.

These objects inserted into Working Memory represent the valid facts that rules can access. Facts are the application data, meanwhile the rules represent the logic layer of the application. This vocabulary is used by rules through the *import* declarations, which are specified inside of the rules file (*drl* files or *xml* files). For example, a rule from Drools may use one or many Java beans classes in order to describe its own vocabulary. The Java bean classes represent a description of the facts used by the Drools rule engine.

A R2ML rule always refers to a vocabulary which can be R2ML own vocabulary or an imported one (i.e. UML[13], RDF(S)[14] and OWL[15] - see lines 3.-4. from Section 2.2 i.e. an example of the importing an OWL external vocabulary for an entire R2ML production rule set). R2ML vocabulary is a serialisation of an UML fragment of class diagrams. Below, we describe the corresponding translation class from a usual Java bean into R2ML elements of the vocabulary namespace with the help of the optionally element `r2mlv:Vocabulary` i.e. Since almost all names from

[11]REWERSE - http://rewerse.net/

[12]RETE adaptation for an object-oriented language, a descendant of the well-known RETE algorithm

[13]UML - http://www.uml.org

[14]RDF(S) - http://www.w3.org/TR/rdf-schema

[15]OWL - http://www.w3.org/2004/OWL

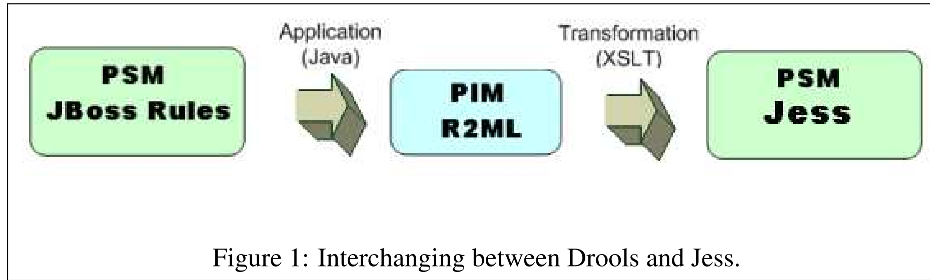

Figure 1: Interchanging between Drools and Jess.

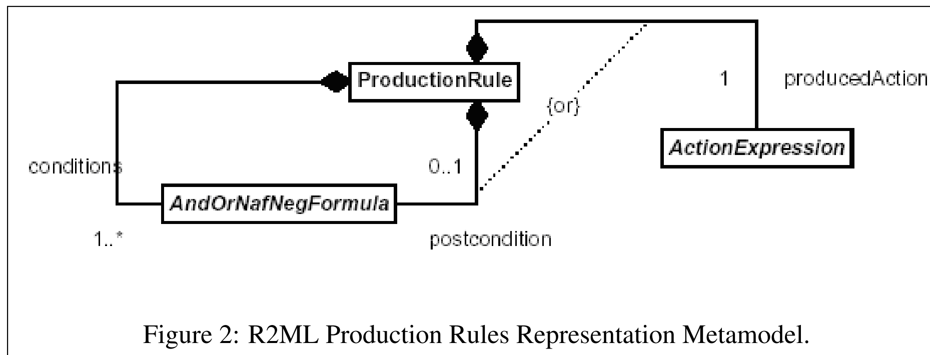

Figure 2: R2ML Production Rules Representation Metamodel.

```
 1.<r2ml:RuleBase
 2.     xmlns:r2ml=
        "http://www.rewerse.net/I1/2006/R2ML"
 3.     xmlns:dc=
        "http://purl.org/dc/elements/1.1/"
 4.     xmlns:ex=
        "http://www.businessrulesforum.com/2007/"
 5.     xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance"
 6.     xsi:schemaLocation=
        "http://www.rewerse.net/I1/2006/R2ML
 7.     http://oxygen.informatik.tu-cottbus.de/
                                   R2ML/0.5/R2ML.xsd"
 8.<r2mlv:Vocabulary>
 9. <r2mlv:Class r2mlv:ID="Cheese">
10.   <r2mlv:Attribute r2mlv:ID="type">
11.    <r2mlv:range><r2mlv:Datatype
        r2mlv:ID="xs:string"/>
12.   </r2mlv:range>
13.   </r2mlv:Attribute>
14.   <r2mlv:Attribute r2mlv:ID="price">
15.    <r2mlv:range><r2mlv:Datatype
        r2mlv:ID="xs:integer"/>
16.   </r2mlv:range>
17.   </r2mlv:Attribute>
18.   <r2mlv:Attribute r2mlv:ID="bestBefore">
19.    <r2mlv:range><r2mlv:Datatype
        r2mlv:ID="xs:dateTime"/>
20.    </r2mlv:range>
21.   </r2mlv:Attribute>
22. </r2mlv:Class>
23.</r2mlv:Vocabulary>
24.</r2mlv:RuleBase>
```

R2ML rule bases are qualified names (`xs:QName`), they must have declared the corresponding namespaces (i.e. see above lines 2.-5.). In the same manner, any Java qualified class name will be translated into a qualified name (`xs:QName`) together with the corresponding names declarations.

For example, if we assume the Drools import declaration (i.e. a Java qualified name): `org.drools.usecase.Cheese`, this will translate into the following namespace declaration `xmlns:ex="http://www.drools.org/usecase"` used in the qualified name (i.e. `ex:Cheese`), in order to reference the class name.

2.2 Rule Sets Mapping

All imported Java beans in Drools rule packages form the rules vocabulary. The set of Drools rules is individualized by its *package namespace*, declared at the beginning of the rules file, namespace that can be equal or can differ from Drools *import* declarations i.e. The Drools package of rules

```
package org.drools.rules;

import org.drools.usecase.Cheese;
/* set of Drools rules */
```

finds its correspondent into the `r2ml:ProductionRuleSet` element. It contains three optional attributes:

- `r2ml:ruleSetID` - is the name of the rule set. The name of the Java package of classes identifies in an unique way the name of a R2ML ProductionRuleSet (i.e. see line 2. from below R2ML code example).
- `r2ml:externalVocabulary` - represents an URI of an external vocabulary. We used OWL to represent the vocabulary of the rule.
- `r2ml:externalVocabularyLanguage` - refers the language of the external vocabulary.

```
1. <r2ml:ProductionRuleSet
2.  r2ml:ruleSetID="org.drools.rules"
3.  r2ml:externalVocabulary="http://..."
4.  r2ml:externalVocabularyLanguage="OWL">
```

Excepting the *rules* and their *import* declarations, a Drools package may contain other specific constructs like: *globals*, user-defined *functions* and *queries*, but they do not represent the subject of our translation.

```
1. rule "<name>"
2.  when
3.   <LHS>
4.  then
5.   <RHS>
6. end
```

```
// java-like, single line comment
#  single line comment
/* ...
   java-like, multi lines comment
   ... */
```

2.3 Rule Mapping

In Drools, a rule consists of the rule *identifier*, the *conditions* part called LHS (i.e. Left Hand Side) and the *actions* part called RHS (i.e. Right Hand Side). The general principles of mapping a Drools rule into a R2ML production rule is:

- Every JBoss production rule is translated into a `r2ml:ProductionRule` element. An optional element `r2ml:Documentation` can contain elements which comprise the rule text and also the representation of the rule in a specific rules language.
- A R2ML `r2ml:ruleID` production rule attribute is generated using the JBoss `<name>` value. The `r2ml:ruleID` unique identifies a rule inside a rule set.
- A JBoss rule has a conditions part (i.e. *when* part) and an action part (i.e. *then* part). The condition part of a JBoss rule is mapped into the content of `r2ml:conditions` role element. The RHS part of a JBoss rule which contains multiple actions maps into the content of `r2ml:producedActionExpr` role element.
- The Drools language syntax also contains the comments expressed in Java-like syntax, such as:

  When translated into R2ML syntax, they map into the XML `<[!CDATA[...]]>` construct. For example:

```
1.<r2ml:Documentation>
2. <r2ml:RuleText r2ml:textFormat="plain">
3.   <[!CDATA[
4.    JBoss rule expressed in natural language...
5.     ]]>
6. </r2ml:RuleText>
7.</r2ml:Documentation>
```

In the following lines we describe the mapping of Drools conditions into R2ML appropriate ones.

The LHS (i.e. *when* part) of a JBoss rule consists of *patterns* (i.e. columns) and `eval` as *Conditional Elements* (i.e. CE) in order to facilitate the encoding of propositional logic and First Order Logic i.e. FOL. The entire LHS of a Drools rule is in fact a tuple of facts (i.e. a tuple of patterns). Each pattern may have zero or more field constraints i.e. the pattern terms (see Figure 4). The `and` (i.e. `&&`) CE is implicit when the JBoss rule condition contains multiple patterns. Field constraints compare and assess the field values from the fact object instances. Drools facts from Working Memory are Java beans objects instances, therefore these field constraints can be accessed from the "no arguments" methods, also called the accessors (i.e. getters).

2.3.1 Mapping Drools patterns without Field Constraints

A Drools pattern without field constraints, will map into the `r2ml:ObjectClassificationAtom`. For example, the following Drools pattern, which corresponds to universally quantified formula from classical logic: $\forall ?c\, Cheese(?c)$ is expressed in Drools as following:

```
$c: Cheese()
```

This Drools pattern finds its R2ML translation into the below code. As an explanation, we mention that all the R2ML formulae are implicitly universal quantified:

```
1.<r2ml:ObjectClassificationAtom
2. r2ml:class="Cheese">
3. <r2ml:ObjectVariable r2ml:name="c"
4.  r2ml:class="Cheese"/>
5.</r2ml:ObjectClassificationAtom>
```

Following RuleML, R2ML framework defines the generic concepts of variable. However, R2ML makes a clear distinction between *object terms* and *data terms*.

Typed terms are either *object terms* standing for *objects*, or *data terms* standing for *data values*. The concrete syntax of first-order non-Boolean OCL (7) expressions can be directly mapped to R2ML abstract concepts of *ObjectTerm* and *DataTerm*, which can be viewed as a predicate-logic-based reconstruction of the standard OCL abstract syntax.

The bounded variable `c` represents the value of the `r2ml:name` attribute of the corresponding term (`r2ml:ObjectName` and/or `r2ml:ObjectVariable`) and the name of the Java bean class (i.e. `Cheese`) is the value of `r2ml:class` attribute. The above Drools pattern can be declared inside rules conditions also without the `c` variable, such as: `Cheese()`, but to be able to refer to the matched facts, usually, the rules conditions use a pattern binding variable such as `c` (i.e. in Drools terminology we refer to it as a *fact variable* or *declaration*).

Any JBoss variables translate into R2ML variables. Notice that the translation of the Drools variables into R2ML eliminates the `$` symbol (used in Drools only as a notation convention) from the names of the variables. The JBoss *fact variable* used in the previous pattern example (i.e. `c:Cheese()`) is mapped into `r2ml:ObjectVariable` using the value of `r2ml:name="c"` property to describe the variable name, which represents an instance of the `Cheese` class (see lines 3.-4.). The usage of this instance gives us the possibility to call properties and functions of `Cheese` class in the actions part of a JBoss rule. The optional `r2ml:class` property (see line 4. from the above example) specifies the type of the object variable (i.e. `Cheese`). An `r2ml:ObjectVariable` is a variable that can be only instantiated by objects.

2.4 Mapping Drools patterns with Field Constraints

In many cases, a JBoss *pattern* (see Figure 3) may contain many field constraints, all of them referring to the same context variable. The Drools field constraints may be of the following possible types (i.e. string, numeric, boolean and date). When separated by the following operators (i.e. enumerated here in their priority order see also Figure 5): `&&`, `||` and `,` (i.e. comma), they form a Drools pattern formula.

A Drools pattern formula translates into R2ML formula, using the R2ML simple/imbricated concepts of `r2ml:qf.Disjunction` and `r2ml:qf.Conjunction` (`qf` stands for "quantifier free") applied on R2ML atoms, in order to serialize the Drools CE `||` and `&&`, respectively.

In the example below, we have two Drools patterns that in classical logic have the following representation, taking into account the operators order from Drools i.e.

$$\forall ?c\, \forall ?p\, \exists ?youngCheese\ (Person(?p) \wedge like(?p, ?youngCheese)\ \wedge\ (Cheese(?c) \wedge (type(?c, ?youngCheese)\ \wedge\ price(?c) < 10) \vee bestBefore(?c)\ <\ "27-Oct-2010"))$$

```
1.$p:Person($youngCheese:like)
2.$c:Cheese(type == $youngCheese &&
3.          price < 10 ||
4.          bestBefore < "27-Oct-2010")
```

The `Cheese` pattern (see lines 2.-4.) has three field constraints combined with a conjunctive connector (i.e. `&&`) and a disjunctive connector (i.e. `||`). We mention that the comma represents by default the conjunctive logic operator. The pattern refers literal constraints used to match the facts (i.e. instances of `Cheese` class): `type` (i.e. string constraint), `price` (i.e. numeric constraint) and `bestBefore` (i.e. date type constraint). The valid operators that apply for the numeric and date operands are: ==, !=, <,>, <=, >=.

The above Drools pattern translates into the following R2ML formula (i.e. the example below describes only the imbrication of the operators (`&&`, `||`) inside the Drools pattern):

```
 1.<r2ml:qf.Disjunction>
 2. <r2ml:qf.Conjunction>
 3.  <r2ml:DatatypePredicateAtom>
 6.    ...
 7.  </r2ml:DatatypePredicateAtom>
 8.   <r2ml:DatatypePredicateAtom>
 9.    ...
10.  </r2ml:DatatypePredicateAtom>
11. </r2ml:qf.Conjunction>
12. <r2ml:DatatypePredicateAtom>
14.    ...
15. </r2ml:DatatypePredicateAtom>
16.</r2ml:qf.Disjunction>
```

In the absence of the `qf.Disjunction` or `qf.Conjunction`, all atoms from the R2ML rule body are implicitly connected by conjunction.

First pattern from the Drools example above (see line 1.) contains as field constraint a bound variable, called declaration. The JBoss variable `$youngCheese` is bound to the `like` property, so it can later constrain the `type` property of the `Cheese` class. Since the `like` property is a data type property (i.e. String), the R2ML mapping is an `r2ml:AttributionAtom`, while for an object type property would find its mapping into the `r2ml:ReferencePropertyAtom`.

```
<!--$p:Person($youngCheese:like)-->
 1.<r2ml:AttributionAtom
 2. r2ml:attribute="ex:Person.like">
 3. <r2ml:subject>
 4.  <r2ml:ObjectVariable
 5.   r2ml:name="p"
 6.   r2ml:class="Person"/>
 7. </r2ml:subject>
 8. <r2ml:dataValue>
 9.  <r2ml:DataVariable
10.   r2ml:name="youngCheese"
11.   r2ml:datatype="xs:string"/>
12. </r2ml:dataValue>
13.</r2ml:AttributionAtom>
```

The `r2ml:AttributionAtom` contains the `r2ml:subject` element which encloses the object term we refer. This can be expressed, for example, by an `r2ml:ObjectVariable` (i.e. lines 4.-6.). The value of the JBoss property is referred by the `r2ml:dataValue`. In this example, the value is encoded by the `r2ml:DataVariable` element (i.e. lines 9.-12.). The `youngCheese` variable borrows the type of the `youngCheese` property. R2ML uses XML Schema Datatypes[16] as its default namespace for encoding basic datatypes. The usage of this set of pre-declared datatypes is not mandatory, the user can specify any other appropriate URI and namespace for referring later in the rules its datatypes declaration. The Java/XML type correspondence it is done according to the JAXB[17] binding style correspondence principle, therefore a Java String value will be translated into `xs:string` qualified name (see line 11.).

The relational operations from Drools are serialized into R2ML language using the `r2ml:DatatypePredicateAtom` construct. Until this version, R2ML had not declared its own built-in constructs, but it allows the use of external ones, such as SWRL[18] built-ins for representing the predicate type of the relational operations (i.e. `swrlb:lessThen`). We also use the construct of `r2ml:DatatypePredicateAtom` to serialize the Drools literal field constraints that test equality / inequality of data types properties. When serializing object types literal field constraints we use the `r2ml:EqualityAtom` to express the concept of equality and the `r2ml:InequalityAtom` to express the concept of inequality (!=).

The following Drools pattern describes a Drools literal String constraint. Drools pattern translates into `r2ml:DatatypePredicateAtom` using the `r2ml:datatypePredicate="swrlb:equal"` SWRL build-in to represent the equality operator and serialises the `type` property into the `r2ml:AttributeFunctionTerm` (see

[16]XML Schema Part 2: Datatypes Second Edition - http://www.w3.org/TR/xmlschema-2/

[17]Java Architecture for XML Binding - java.sun.com/developer/technicalArticles/WebServices/jaxb/

[18]SWRL - http://www.w3.org/Submission/SWRL

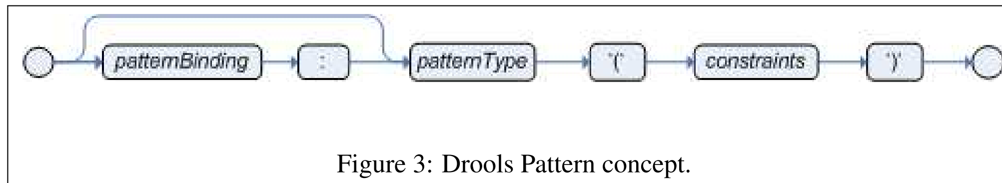

Figure 3: Drools Pattern concept.

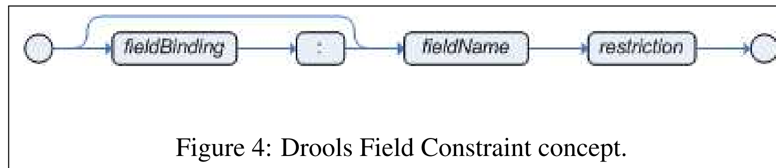

Figure 4: Drools Field Constraint concept.

lines 4.-10.). The `r2ml:dataArguments` attribute comprises

```
$c:Cheese(type == $youngCheese)
```

the Drools operands translation into R2ML terms objects or data types, depending on the types of the involved properties. The Drools `$youngCheese` variable is expressed using the concept of `r2ml:DataVariable` i.e.

```
<!--$c:Cheese(type == $youngCheese)-->
1.<r2ml:DatatypePredicateAtom
2. r2ml:datatypePredicate="swrlb:equal">
3. <r2ml:dataArguments>
4.  <r2ml:AttributeFunctionTerm
5.   r2ml:attribute="ex:Cheese.type">
6.   <r2ml:contextArgument>
7.    <r2ml:ObjectVariable r2ml:name="c"
8.     r2ml:class="ex:Cheese"/>
9.    </r2ml:contextArgument>
10.  </r2ml:AttributeFunctionTerm>
11.  <r2ml:DataVariable r2ml:name="youngCheese"
12.   r2ml:datatype="xs:string"/>
13. </r2ml:dataArguments>
14.</r2ml:DatatypePredicateAtom>
```

The relational operation `$c:Cheese(price < 10)` is expressed by the `r2ml:DatatypePredicateAtom` and the build-in `swrlb:lessThan`. The case also involves the `r2ml:AttributeFunctionTerm` for representing the property `price` of the Cheese class (i.e. `ex:Cheese.price` (see lines 4.-10.) and the `r2ml:TypedLiteral` term (see lines 11.-12.) for encoding the Java integer value into XML `xs:integer`.

```
<!--$c:Cheese(price < 10)-->
1.<r2ml:DatatypePredicateAtom
2. r2ml:datatypePredicate="swrlb:lessThan">
3. <r2ml:dataArguments>
4.  <r2ml:AttributeFunctionTerm
5.   r2ml:attribute="ex:Cheese.price">
6.   <r2ml:contextArgument>
7.    <r2ml:ObjectVariable r2ml:name="c"
8.     r2ml:class="ex:Cheese"/>
9.    </r2ml:contextArgument>
10.  </r2ml:AttributeFunctionTerm>
11.  <r2ml:TypedLiteral r2ml:lexicalValue="10"
12.   r2ml:datatype="xs:integer"/>
13. </r2ml:dataArguments>
14.</r2ml:DatatypePredicateAtom>
```

The Drools literal date type field constraints are represented into R2ML using XML qualified name `xs:dateTime`.

The Drools numeric operators work analogous for this type of field constraint, so the serialization into R2ML code is also the `r2ml:DatatypePredicateAtom` i.e.

```
<!--$c:Cheese(bestBefore<"27-Oct-2007")-->
 1.<r2ml:DatatypePredicateAtom
 2. r2ml:datatypePredicate="swrlb:lessThan">
 3.  <r2ml:dataArguments>
 4.  <r2ml:AttributeFunctionTerm
 5.   r2ml:attribute="bestBefore">
 6.   <r2ml:contextArgument>
 7.    <r2ml:ObjectVariable r2ml:name="c"
 8.     r2ml:class="Cheese"/>
 9.   </r2ml:contextArgument>
10.  </r2ml:AttributeFunctionTerm>
11.  <r2ml:TypedLiteral
12.   r2ml:datatype="xs:dateTime"
13.   r2ml:lexicalValue="2007-10-27Z"/>
14. </r2ml:dataArguments>
15.</r2ml:DatatypePredicateAtom>
```

Another meaningful example of Drools field constraints is testing the equality or inequality of a property against the Java `null` value i.e.

```
$c:Cheese(buyer == null)
$c:Cheese(buyer != null)
```

The corresponding formula from the classical logic would be:

$$\forall ?c\, \forall ?t\; Cheese(?c) \wedge \neg buyer(?c, ?t)$$

$$\forall ?c\, \exists ?t\; Cheese(?c)\; \wedge\; buyer(?c, ?t)$$

Taking into account the above classical logical formula, the R2ML serialization results in the `r2ml:EqualityAtom`, having its meaning negated using `r2ml:isNegated=true` attribute. The child elements for the `r2ml:EqualityAtom` are object terms. Our example involves an `r2ml:ReferencePropertyTerm` with the attribute `r2ml:referenceProperty="ex:Cheese.buyer"` and a generated object term expressed using `r2ml:ObjectVariable` with the generated attribute value `r2ml:name="t_24535899"` and the `r2ml:class="ex:Person"` as the type of the `buyer` property.

The second logic formula involves the existence of a `Cheese` fact into Working Memory, whose `buyer` property is not null. The R2ML first step in the R2ML serialization is the generation of an object term (i.e. the `r2ml:ObjectVariable` `t_57685642`) of `ex:Person` type which is bounded to `buyer` property. We use the `r2ml:ReferencePropertyAtom` element i.e.

We have mentioned before that implicitly, the `and` operator binds the Drools patterns inside the rule condition.

```
<!--$c:Cheese(buyer==null)-->
 1.<r2ml:EqualityAtom
 2. r2ml:isNegated="true">
 3. <r2ml:ReferencePropertyFunctionTerm
 4.  r2ml:referenceProperty="ex:Cheese.buyer">
 5.  <r2ml:contextArgument>
 6.   <r2ml:ObjectVariable
 7.    r2ml:name="c"
 8.    r2ml:class="ex:Cheese"/>
 9.  </r2ml:contextArgument>
10. </r2ml:ReferencePropertyFunctionTerm>
11. <r2ml:ObjectVariable
12.  r2ml:name="t_24535899"
13.  r2ml:class="ex:Person"/>
14.</r2ml:EqualityAtom>
```

```
<!--$c:Cheese(buyer!=null)-->
 1.<r2ml:ReferencePropertyAtom
 2. r2ml:referenceProperty="ex:Cheese.buyer">
 3. <r2ml:subject>
 4.  <r2ml:ObjectVariable
 5.  r2ml:name="c" r2ml:class="ex:Cheese"/>
 6. </r2ml:subject>
 7. <r2ml:object>
 8.  <r2ml:ObjectVariable r2ml:name="t_57685642"
 9.   r2ml:class="ex:Person"/>
10. </r2ml:object>
11.</r2ml:ReferencePropertyAtom>
```

```
//$p:Person()
1.$c:Cheese(buyer == $p, inStock == true)
```

A Drools pattern containing a formula that implies only field constraints conjunctions, can be split into as many Drools patterns as field constraints it contains i.e.

```
//$p:Person()
1.$c:Cheese(buyer == $p)
2.$c:Cheese(inStock == true)
```

There are two possibilities to markup the above `Cheese` patterns (see lines 1.-2.): to use a conjunction of R2ML appropriate atoms or to use the `r2ml:ObjectDescriptionAtom` construct.

First option implies the use of the `r2ml:ReferencePropertyAtom` in order to markup the first pattern (see line 1. from Drools example) and a `r2ml:AttributionAtom` for the data type boolean field constraint (see line 2. from Drools example) i.e.

```
<!--$c:Cheese(buyer == $p)-->
 1.<r2ml:ReferencePropertyAtom
 2. r2ml:referenceProperty="ex:Cheese.buyer">
 3. <r2ml:subject>
 4.  <r2ml:ObjectVariable
 5.   r2ml:name="c"
 6.   r2ml:class="ex:Cheese"/>
 7. </r2ml:subject>
 8. <r2ml:object>
 9.  <r2ml:ObjectVariable
10.   r2ml:name="p"
11.   r2ml:class="ex:Person"/>
12. </r2ml:object>
13.</r2ml:ReferencePropertyAtom>
```

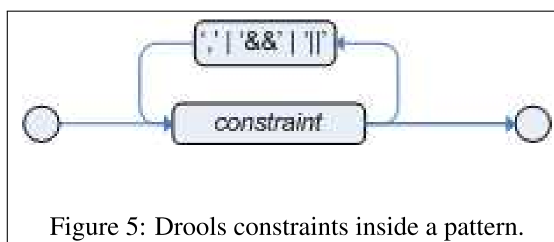

Figure 5: Drools constraints inside a pattern.

A `r2ml:ReferencePropertyAtom` associates two object terms, having different meanings: `subject` (see lines 3.-7.) and `object` (see lines 8.-12.). For example, to express the concept of: "buyer of cheese" we use the code above, where the subject is a `Cheese` instance and the object is a `Person` instance.

The second solution translates directly the two `Cheese` patterns of the rule using the `r2ml:ObjectDescriptionAtom`. The referred R2ML serialization is a conjunction of equality constraints i.e. an enumeration of `r2ml:DataSlot`(s) or `r2ml:ObjectSlot`, depending on the type of the involved properties objects or data, respectively. The attribute `r2ml:class="Cheese"` corresponds to the patterns name from Drools implementation i.e.

```
 1.<r2ml:ObjectDescriptionAtom
 2. r2ml:class="Cheese">
 3. <r2ml:subject>
 4.  <r2ml:ObjectVariable r2ml:name="c"/>
 5. </r2ml:subject>
 6. <r2ml:ObjectSlot
 7.  r2ml:referenceProperty="Cheese.buyer">
 8.  <r2ml:object>
 9.   <r2ml:ObjectVariable r2ml:name="p"
10.    r2ml:class="Person"/>
11.   </r2ml:object>
12.  </r2ml:ObjectSlot>
13.  <r2ml:DataSlot
14.   r2ml:attribute="ex:Cheese.inStoc">
15.   <r2ml:value>
16.    <r2ml:DataVariable
17.     r2ml:name="true"
18.     r2ml:datatype="xs:boolean"/>
19.   </r2ml:value>
20.  </r2ml:DataSlot>
21.</r2ml:ObjectDescriptionAtom>
```

Another CE from the LHS of a Drools rule is the pattern disjunction (`||` / `or`). The Drools disjunction of multiple patterns results in multiple rule generation, called subrules, for each possible outcome i.e.
$\forall ?c\, Cheese(?c) \ \wedge \ (type(?c, stilton) \vee type(?c, cheddar))$

```
1.Cheese(type=="stilton") or Cheese(type=="cheddar")
2.Cheese(type=="stilton") || Cheese(type=="cheddar")
```

In the above examples the Drools `or` CE is a shortcut for generating two additional rules. There can be multiple activations for a rule, if both sides of the CE are true. The R2ML serialization uses the `r2ml:qf.Disjunction` that contains each of the `Cheese` properties mapped with the `r2ml:AttributionAtom`.

The Drools negation `not` represents the existential quantifier that checks for the non existence of some facts in Working Memory. Currently, this existential quantifier it is applied only for patterns i.e. $\forall ?c\, \neg Cheese(?c)$.

```
not Cheese()
```

The above pattern tests if there are not `Cheese` facts in the Working Memory. The "not" pattern (see Figure 7) can

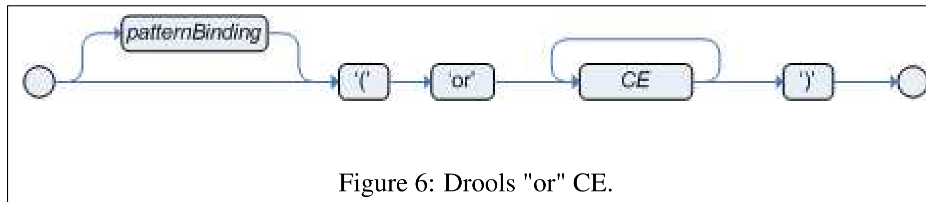

Figure 6: Drools "or" CE.

not have a pattern binding. We still can not serialize the existentially quantifier concept into R2ML language.

But, by applying the negation to the entire formula, we obtain the following expression from the classical logic: $\forall ?c \neg Cheese(?c)$. The R2ml serialization of the above formula uses the concept of `r2ml:qf.Negation` which embeds a `r2ml:ObjectClassificationAtom`.

```
1. <r2ml:ObjectClassificationAtom r2ml:isNegated="true"
2.  r2ml:classID="Car">
3.  <r2ml:ObjectVariable r2ml:name="c_974376"/>
4. </r2ml:ObjectClassificationAtom>
```

```
not Cheese(type == "stilton")
```

The above pattern tests if there are not `Cheese` facts of type `stilton` in the Working Memory. In the classical logic would be i.e.

$$\forall ?c\, Cheese(?c) \;\wedge\; \neg\, type(?c, stilton)$$

. We serialize it using the `r2ml:AttributionAtom` i.e.

```
 1.<r2ml:AttributionAtom
 2. r2ml:attribute="ex:Cheese.type"
 3. r2ml:isNegated="true">
 4. <r2ml:subject>
 5.  <r2ml:ObjectVariable r2ml:name="c_6587483"
 6.   r2ml:class="ex:Cheese"/>
 7. </r2ml:subject>
 8. <r2ml:dataValue>
 9.  <r2ml:TypedLiteral
10.   r2ml:lexicalValue="stilton"
11.   r2ml:datatype="xs:string"/>
12. </r2ml:dataValue>
13.</r2ml:AttributionAtom>
```

2.4.1 Mapping Drools Actions

Java beans objects/instances are defined by users in their applications. These objects inserted into Working Memory (i.e. WM) represent the valid facts which the rules can access. Facts are the application data, meanwhile the rules represent the logic layer of the application. The term Working Memory Actions is used to describe assertions, retractions and modifications of facts within Working Memory. When discussing about the Drools - R2ML mapping of actions, we are only referring to the JBoss rule actions that find their mapping into R2ML.

R2ML actions are built according with the OMG PRR Specification (8), which stipulates that an action is either an `r2ml:InvokeActionExpression` or an `r2ml:UpdateStateActionExpr`. The R2ML actions are encoded by the content of `r2ml:producedActionExpr` role element i.e.

`r2ml:InvokeActionExpression` - invokes an operation (by means of the `r2ml:operation` attribute) with an ordered, possible empty list of parameter arguments represented as R2ML terms. In the following example, we map a Java output operation, which has as `r2ml:arguments` the Cheese instance (i.e. `c`), previously, supposed to be declared in the JBoss rule condition (see lines 4.-5.) and a String argument, that is translated into `r2ml:TypedLiteral` (see lines 6.-8.).

```
then
 System.out.println($c+" out of stock.");
end
```

The R2ML translation:

```
 1.<r2ml:InvokeActionExpression
 2. r2ml:operation="System.out.println">
 3. <r2ml:arguments>
 4.  <r2ml:ObjectVariable r2ml:name="c"
 5.   r2ml:Class="ex:Cheese"/>
 6.  <r2ml:TypedLiteral
 7.   r2ml:lexicalValue=" out of stock"
 8.   r2ml:datatype="xs:string"/>
 9. </r2ml:arguments>
10.</r2ml:InvokeActionExpression>
```

`AssertActionExpr` - contains a collection of slots i.e. property-value pairs (e.g. `r2ml:DataSlot` / `r2ml:ObjectSlot`) in order to represent the data/object type properties of a fact. We use this R2ML action call in order to map the JBoss `insert(object)` / `insertLogical(object)` Working Memory Actions, which has the purpose to insert new memory data.

```
then
  // Offer(cheese, price)
  Offer offer = new Offer($cheese,100)
  insert(offer);
end
```

The R2ML translation needs an instance of the `Offer` class for its `r2ml:contextArgument`, which encode the context of the action call, so we generate an `r2ml:ObjectVariable` with the `r2ml:name="offer"`. We translate the instance of the Cheese class to a `r2ml:ObjectSlot` and the direct value `100` as a `r2ml:TypedLiteral` having the type of the `price` property of the `Offer` class (i.e. `xs:integer`).

```
 1.<r2ml:AssertActionExpr
 2. r2ml:class="ex:Offer">
 3. <r2ml:contextArgument>
 4.  <r2ml:ObjectVariable
 5.   r2ml:name="offer"
 6.   r2ml:class="ex:Offer"/>
 7. </r2ml:contextArgument>
 8. <r2ml:ObjectSlot
 9.  r2ml:referenceProperty="ex:Offer.cheese">
10.  <r2ml:object>
11.   <r2ml:ObjectVariable
12.    r2ml:name="cheese"
```

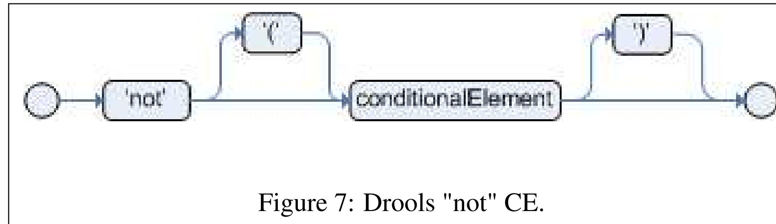

Figure 7: Drools "not" CE.

```
13.    r2ml:class="ex:Cheese"/>
14.  </r2ml:object>
15. </r2ml:ObjectSlot>
16. <r2ml:DataSlot
17.  r2ml:attribute="ex:Offer.price">
18.  <r2ml:value>
19.   <r2ml:TypedLiteral
20.    r2ml:datatype="xs:integer"
21.    r2ml:lexicalValue="100"/>
22.  </r2ml:value>
23. </r2ml:DataSlot>
24.</r2ml:AssertActionExpr>
```

RetractActionExpr - deletes an object. Its r2ml:contextArgument always evaluates to an object term. We use this R2ML construct to map the JBoss WM Action for removing a previously declared fact (i.e. c:Cheese()) from the memory i.e.

```
then
  retract($c);
end
```

The R2ML translation is:

```
1.<r2ml:RetractActionExpr r2ml:class="ex:Cheese">
2. <r2ml:contextArgument>
3.  <r2ml:ObjectVariable
4.   r2ml:name="c"
5.   r2ml:class="ex:Cheese"/>
6. </r2ml:contextArgument>
7.</r2ml:RetractActionExpr>
```

UpdateActionExpr - updates a property of a specific object term specified by the r2ml:contextArgument. The below Drools example modifies the type property of a particular Cheese instance (i.e. c). The update(c) is necessary in order to notify the Drools engine about the changes from WM.

```
then
 $c.setType("cheddar");
 update($c);
end
```

The R2ML translation is:

```
 1.<r2ml:UpdateActionExpr
 2. r2ml:property="ex:Cheese.type">
 3. <r2ml:contextArgument>
 4.  <r2ml:ObjectVariable r2ml:name="c"
 5.   r2ml:class="ex:Cheese"/>
 6. </r2ml:contextArgument>
 7. <r2ml:TypedLiteral
 8.  r2ml:lexicalValue="cheddar"
 9.  r2ml:datatype="xs:string"/>
10.</r2ml:UpdateActionExpr>
```

3 R2ML to Jess mapping

In this section we describe how R2ML rules are translated into Jess rule language. In Jess, rules are defined using the defrule construct. They are, in fact written as lists where the head is the special symbol defrule.

Jess provides two main categories of rules: forward-chaining rules and backward-chaining rules. Forward-chaining rules are the most common and used rules in Jess, and our translation will obtain Jess forward-rules. Jess is a forward-chaining reasoning engine, backward-chaining rules being simulated in terms of forward chaining (3).

To translate a R2ML rule into a Jess rule we will use Jess *unordered facts*. Unordered facts from Jess are alternatives for Java bean instances: objects that have named fields (i.e. properties) in which data appears (although the properties are traditionally called slots (see code example from 3.1)).

Any R2ML rule will translate into a Jess rule which uses unordered facts, because they are nice structured and are better emulating the internal structure of a R2ML rule (which includes the rule vocabulary).

We also use the *new-style*, simplified syntax of the Jess language, introduced in version 7.0[19]. We mention that the Drools language syntax had received and still receives a strong influence from Clips/Jess made and ongoing researches.

3.1 Mapping rules vocabulary

Jess business rules vocabulary consists of deftemplate(s) structures.

A deftemplate describes a fact, in the same way as a Java class describes an object. In particular, a deftemplate is a Jess concept which includes a name, an optional documentation string, an optional "extends" clause, an optional list of declarations, and a list of zero or more member variables (called slot descriptions) with a type qualifier. Each slot description can optionally include a type qualifier or a default value qualifier.

Using vocabulary classes from R2ML, corresponding Jess deftemplates are generated. The deftemplate[20] structure corresponds to the class description from the R2ML vocabulary 2.1 i.e. They can be placed in the same

```
(deftemplate Cheese
  (slot type (type STRING))
  (slot price (type INTEGER))
  (slot bestBefore (type OBJECT)) )
```

[19] Jess 7.0 - http://herzberg.ca.sandia.gov/docs/70/release_notes.html

[20] http://herzberg.ca.sandia.gov/docs/71/api/jess/Deftemplate.html

file as the rules, or in a separate file, which need to be imported into the rules file, using the `import` keyword.

3.2 Rule Sets Mapping

The output of the translation from R2ML to Jess is a `.jess` batch file (i.e. a Jess knowledge base). This batch file contains the facts and the rules which represent the input data and the logic for the Jess Rete engine, of which the current version is 7.0. A Jess rule set is a Jess batch file. The name of this file is obtained from the `r2ml:ruleSetID` attribute (see Section 2.2).

3.3 Rule Mapping

- The `r2ml:ruleID` attribute value is used to obtain a Jess rule ID, which is not allowed to contain spaces.
- In R2ML framework the content of `r2ml:conditions` role element which corresponds to an universally quantified formula is translated into the conditions part of a Jess rule i.e. LHS pattern. The LHS of a Jess rule consists of patterns that match facts.
- The content of a `r2ml:producedActionExpr` is markup as the actions part of a Jess rule i.e. RHS pattern introduced by the symbol => , which roughly denotes implication. The actions of a Jess rule are composed only of function calls.
- Jess language supports two kinds of comments: Lisp-style line comments (;) and C-style block comments (/*...*/), in order to translate the R2ML comments.

```
(defrule ruleName
  (pattern1)
  (pattern2)
   ;; ...
  =>
  (function calls))
```

3.3.1 Mapping r2ml:ObjectClassificationAtom

An R2ML atom corresponds to a Jess pattern. The `r2ml:ObjectClassificatioAtom` is used to capture the *instanceOf* relationship between objects and classes. Any R2ML object classification atom consists from a mandatory attribute `r2ml:class` with a (`xs:QName`) value and an object term as an argument.

A positive `r2ml:ObjectClassificationAtom` (see Section 2.3.1) is mapped into a Jess pattern without field constraints i.e. `?fact_variable <- (PatternName{})` where `?fact_variable` is the corresponding term (i.e. `r2ml:ObjectVariable`) and `PatternName` is the value of `r2ml:class` attribute i.e.

```
?c <- (Cheese{})
```

3.3.2 Mapping R2ML Formulae

Any `r2ml:qf.Conjunction` (*qf* stands from *quantifier free*) of atoms corresponds to a conjunction inside of Jess patterns. It represents an enumeration of Jess field constraints. If conjunctions contain conditions referring to the same context, then we translate all the R2ML atoms (i.e. the R2ML formula) into a Jess single pattern with a number of field constraints. Every R2ML atom finds its mapping into the Jess field constraint concept.

Any R2ML variable name is mapped into Jess variable identifier by adding the ? symbol as first character: `?fact_variables` and `?field_variables`. R2ML variables are provided in the form of `r2ml:ObjectVariable` and `r2ml:DataVariable`. `r2ml:ObjectVariable` are variables that can be only instantiated by objects, meanwhile `r2ml:DataVariable` are variables that can be only instantiated by data literals.

The `r2ml:ObjectVariable` is mapped into the Jess concept of `?fact_variable` using the value of the `r2ml:name` attribute as the variable name with type `xs:NCName`. The optional `r2ml:class` attribute specifies the membership of the object variable.

The `r2ml:DataVariable` with attribute `r2ml:typeCategory='individual'` is mapped into Jess as `?field_variable`, being instantiated only by data literal types properties.

The `r2ml:AttributionAtom` captures data valued properties of objects. Any `r2ml:AttributionAtom` maps into a Jess pattern i.e. `?fact_variable<-(PatternName {property ?value})` where `?fact_variable` is the content of the child role element `r2ml:subject` of the involved atom (see Section 2.4 lines 3.-7.), `PaternName` represents the content of attribute `r2ml:class` (see Section 2.4 line 6.), `property` is the value of the attribute `r2ml:attribute` and `?value` is the content of the child role element `r2ml:dataValue` of the atom (see Section 2.4 lines 8.-12.). The appropriate translation into Jess language is:

```
?p <- (Person{} (like ?youngCheese))
```

The `r2ml:DatatypePredicateAtom` is designed to capture built-in predicates. It refers to a user-defined datatype predicate by its mandatory `r2ml:datatypePredicate` attribute and consists of a number of data terms as data arguments (the children of `r2ml:dataArguments` attribute).

Current translation in Jess supports only operations which have two arguments. The `r2ml:datatypePredicate`, represents qualified names which express the appropriate build-ins: (e.g. `swrlb:equal`, `swrlb:lessThan`, `swrlb:lessThanOrEqual`, `swrlb:greaterThan`, `swrlb:greaterThanOrEqual`), and corresponds to appropriate, Jess operators used inside the patterns (==, <, <=, >, >=, ! =, <>).

The `r2ml:DatatypePredicateAtom` code examples from Section 2.4 find their mapping into the following Jess pattern with field constraints i.e.

```
?c <- (Cheese{type == ?youngCheese &&
            price < 10 ||
            bestBefore <= "27/10/2010"})
```

The r2ml:AttributeFunctionTerm is used by R2ML in order to express data valued properties of objects. The name of the attribute is revealed through the mandatory r2ml:attribute value. Usually, a R2ML r2ml:AttributeFunctionTerm is contained, as a child of r2ml:dataArguments construct in R2ML atoms (i.e. r2ml:DatatypePredicateAtom), in order to assign values to object properties (i.e. type, price, bestBefore).

The Jess translation will always take into consideration, the relation between r2ml:AttributeFunctionTerm and other R2ML atoms (see Section 2.4 lines 3.-10.).

R2ML data literals i.e. r2ml:TypedLiteral are mapped into Jess using the values of the r2ml:lexicalValue and r2ml:datatype attributes, into corresponding, Jess valid types. Internally, all Jess values (symbols, numbers, strings, lists etc) are represented by instances of the class jess.Value. Its possible values are enumerated by a set of constants defined into the jess.RU (i.e. Rete Utilities) class (i.e. ANY, INTEGER, FLOAT, NUMBER, SYMBOL, STRING, LEXEME, and OBJECT).

As Jess language does not provide a particular datatype in order to express the XML xs:dateTime value, a possible translation could be the use of the Jess class: jess.RU (i.e. OBJECT constant). As a consequence, the bestBefore slot from the cheese deftemplate construct has the type OBJECT and encapsulates the java.util.Date class.

An R2ML r2ml:ReferencePropertyAtom associates an object term as r2ml:subject with other object term as r2ml:object.

Therefore, the R2ML example of r2ml:ReferencePropertyAtom (see Section 2.4) will translate into the following Jess patterns:

```
?p <- (Person{})
?c <- (Cheese{buyer == ?p &&
            inStock == true})
```

The r2ml:EqualityAtom is intended to express equality of two object terms (i.e. r2ml:ObjectVariable). The corresponding translation into Jess implementation is the symbol of == operator. The r2ml:isNegated attribute set to true value, involves the use of != operator.

The r2ml:InequalityAtom is just a convenience construct to express the negation of the object terms equality. The corresponding translation into Jess implementation is the symbol of != operator. The r2ml:isNegated attribute set to true value, involves the use of == operator.

The Jess nil value has an equivalent meaning with the null value from Java language. Therefore, the corresponding mappings from Section 2.4 are the following i.e.

```
?c <- (Cheese{type == nil})
?c <- (Cheese{type != nil})
```

The r2ml:ObjectDescriptionAtom is a convenience construct to describe a set of property-object value pairs and/or a set of attribute-value pairs which refer to the same object as r2ml:subject.

We translate the r2ml:ObjectDescriptionAtom into a Jess pattern with field constraints. These constraints refer to the same child role element r2ml:subject and can be object terms (i.e. r2ml:ObjectVariable) captured in r2ml:ObjectSlots, or data terms (i.e. r2ml:DataVariable) captured in r2ml:DataSlots.

The R2ML code example from Section 2.4 will translate into:

```
?p <- (Person{})
?c <- (Cheese{buyer == ?p &&
            inStock == true})
```

Any r2ml:qf.Disjunction (*qf* stands from *quantifier free*) is translated into a disjunctive list of Jess patterns, using the Jess conditional element or, all bound to the same fact-variable. Any number of patterns can be enclosed in a list with or conditional element as the head.

This structure is a shortcut for generating two or more additional rules. For a rule which contains such disjunction of patterns, there could be multiple activations if multiple sides of the or are true i.e.

```
(or (Cheese{type=="stilton"})
    (Cheese{type=="cheddar"}) )
```

All R2ML negations i.e. r2ml:qf.Negation, and here we refer to both r2ml:qf.StrongNegation, and r2ml:qf.NegationAsFailure collapse in the Jess negation denoted by the keyword not, which currently applies only for patterns (see R2ML example from Section 2.4).

```
not(Cheese{})
```

Any R2ML atom has an optional, boolean property r2ml:isNegated which tells if the atom is, or is not negated. The corresponding Jess translation describes the negated R2ML atom. If the attribute r2ml:isNegated is missing, this is interpreted in R2ML by the default r2ml:isNegated="false". The example from Section 2.4 describes an r2ml:AttributionAtom which supports a negation through its property r2ml:isNegated="true". The Jess corresponding translation pattern is:

```
not (Cheese{type == "stilton"})
```

We mention that a not pattern cannot define any variables that are used in subsequent patterns (since a not pattern does not match any facts, it can not be used to define the values of any variables). Also, in our both previous examples a not pattern can not have a pattern binding.

3.4 Mapping R2ML actions into Jess Function Calls

The possibles actions of a R2ML production rule are defined using OMG's PRR Proposal (the content of r2ml:producedActionExpr role element). They are mapped into the corresponding Jess function calls i.e.

3.4.1 Mapping r2ml:InvokeActionExpression

An `r2ml:InvokeActionExpression` will map into Jess into a function call with/without arguments. Notice that `?c` is a bound variable to the `Cheese` pattern in the LHS of the rule. The R2ML code example from Section 2.4.1 finds its translation into the following Jess code:

```
(printout t ?c " out of stock!")
```

3.4.2 Mapping r2ml:AssertActionExpr

The `r2ml:AssertActionExpr` assert a new data into Working Memory. The analogous construct in Jess is the `assert` function containing the name of the `deftemplate` name and (slot value) pairs. The R2ML code example from Section 2.4.1 translates into:

```
(assert(offer (cheese ?c) (price 100)) )
```

3.4.3 Mapping r2ml:RetractActionExpr

The `r2ml:RetractActionExpr` deletes an object term from WM. According with the R2ML code from Section 2.4.1 the corresponding Jess translation is:

```
(retract (?c))
```

3.4.4 Mapping r2ml:UpdateActionExpr

The `r2ml:UpdateActionExpr` from our R2ML rule example (see Section 2.4.1) corresponds to the `modify` function invocation from Jess language, which updates an unordered fact from WM. Notice that `?c` is a bound variable to the `score` field value i.e.

```
(modify ?c (type cheddar))
```

4 Limitations of the proposed interchange

The translation process from PSM to PSM using intermediate R2ML (as PIM level) demands also the mapping of rules vocabulary. The Drools to R2ML Translator is a Java application that requests access to the Drools vocabulary (Java compiled classes) in order to establish the types of objects and primitives. As R2ML supports Production Rules format(8), the Drools to R2ML translation of rule conditions part relies naturally.

Interchange limits appear in the translation of the actions part of a JBoss rule, which may contain any Java valid code: variable declaration, non-declarative structures like `if...then` structures or cycling structures(i.e. `while`, `for`).

R2ML intends to solve this problem by providing in its future version an `<r2ml:OpaqueExpression>` with the role to encapsulate the code which do not find its semantic equivalent into R2ML `<r2ml:producedActionExpr>` role element. Also Drools tries to emerge its syntax from the OMG proposal for PRR, therefore the future design will take much more into consideration the standard actions.

The purpose of R2ML, as PIM level markup language, is to provide PSM to PSM rules translation by sharing a vocabulary model (actually R2ML vocabulary), which can easily be mapped into Jess vocabulary (deftemplate(s) structures) i.e.

$$\mathcal{V}_{Drools} \rightarrow \mathcal{V}_{R2ML} \rightarrow \mathcal{V}_{Jess}$$

Therefore, it makes possible the translation from R2ML to Jess language, as the business rules expressed in R2ML format do not require any conceptual changes in order to be implemented in PSM target platforms (e.g. Jess, F-Logic, RuleML, OCL, SWRL, Drools, Jena[21]). However, R2ML is not concerned with the vocabulary interchange issues, therefore this interchange of rules depends of the capabilities of a vocabulary interchange format. In this case, we use the R2ML vocabulary but rules may come with different vocabularies (for example UML, RDF(S) or OWL).

5 Interchange soundness

The soundness of the discussed interchange brought under attention the lack of a well established semantics for both languages (i.e. Drools and Jess). Our solution to establish the interchange soundness is by *testing rules*.

Let $\mathcal{W}_0 = \{f_1, \ldots, f_m\}$ the initial facts from Drools Working Memory and $\mathcal{R} = \{R_1, \ldots, R_n\}$ the current rules set of Drools encapsulating the logic of a specific application. The inference engine (Drools) will execute the rule set $\mathcal{R}$ against $\mathcal{W}_0$ obtaining $\mathcal{W} = \{g_1, \ldots, g_p\}$.

We encode the logic of the same application using the following set of Jess rules: $\mathcal{R}' = \{R'_1, \ldots, R'_n\}$ and as data the initial set of facts $\mathcal{W}'_0 = \{f'_1, \ldots, f'_m\}$. Running the Jess Inference Engine, we obtain a set of final Jess facts i.e. $\mathcal{W}' = \{g'_1, \ldots, g'_p\}$.

We translate the JBoss production rules into Jess implementation via R2ML and we execute those rules based on analogous facts from the Working Memory. The correctness of the translation implies the same obtained results regarding the facts from Working Memory.

A translation from Drools to Jess involves:

- a transformation function

  $$Tr_{(Drools,R2ML)} : Drools \rightarrow R2ML$$

  describing the serialisation from Drools to R2ML, where: Drools = ($\mathcal{V}_{Drools}$ x $\mathcal{W}$ x $\mathcal{R}$)

- a transformation function

  $$Tr_{(R2ML,Jess)} : R2ML \rightarrow Jess$$

  describing the mapping from R2ML to Jess, where: Jess = ($\mathcal{V}_{Jess}$ x $\mathcal{W}'$ x $\mathcal{R}'$)

[21] Jena Framework - `http://jena.sourceforge.net/`

Table 1: Drools to Jess Mapping Rules

Excerpt from Mapping from Drools to Jess		
Drools	R2ML	Jess
Drools rule	R2ML PR	Jess rule
import JavaBeans files	declare R2ML vocabulary	import deftemplates
fact variable	ObjectVariable	fact variable
field variable	ObjectVariable	field variable
Drools pattern	ObjectClassificationAtom	LHS's pattern
conjunction of field constraints	ObjectDescriptionAtom	conjunction of field constraints
field variable binding	AttributionAtom	field variable binding
relational operators on data	DatatypePredicateAtom	relational operators
==/ ! = operators on objects	ReferencePropertyAtom	==/ ! = on object terms
not	isNegated="true"	Jess negation
Drools pattern conjunction	qf.Conjunction	Jess conjunction
Drools pattern disjunction	qf.Disjunction	Jess disjunction
function call	InvokeActionExpression	Jess function call
create/assert an object	AssertActionExpr	assert function call
delete an object	RetractActionExpr	retract function call
setter call	UpdateActionExpr	modify function call

– a translation function $T_{(Drools,Jess)} = Tr_{(Drools,R2ML)}(Tr_{(R2ML,Jess)})$

and results in:

– $T_{(Drools,Jess)}(\mathcal{V}_{Drools}$ x $\mathcal{W}$ x $\mathcal{R}$) = ($\mathcal{V}_{Jess}$ x $\mathcal{W}'$ x $\mathcal{R}'$) (i.e. by applying the translation function $T_{(Drools,Jess)}$ on the $\mathcal{W} = \{g_1, \ldots, g_p\}$ set of facts, we obtain the following set of facts $\mathcal{W}_h = \{h_1, \ldots, h_p\}$ which is semantically and syntactically equivalent with the set $\mathcal{W}' = \{g'_1, \ldots, g'_p\}$, obtained from Jess inference process.

An informal rules translation from Drools to Jess, using R2ML as interchange language is presented in Table 1. The reverse translation, from Jess to Drools is also possible, as Jess supports a new richer syntax[22] which offers the capability to represent object types using *deftemplate* structures. One limitation of this syntax is that it can only be used with unordered facts.

6 Conclusion and future works

The paper provides a description of rule translation from Drools, an Object Oriented rule language, into Jess, an Artificial Intelligence rule language using R2ML as interchange format. The results presented in this paper are based on our previous work (6).

Our future works intend to establish a mathematical model of semantic soundness of rule interchange. Also compatibility of the interchange with the work in progress of W3C (i.e. RIF) will be analysed. Further results will be reported in a future work.

[22]http://herzberg.ca.sandia.gov/docs/71/memory.html

Acknowledgements

We would like to thanks to Edson Tirelli and Kris Verlaenen from the Drools Team for their helpful comments on our questions as well to Ernest Friedman-Hill, a precious mentor in all issues concerning the Jess language.

References

[1] H. Boley, M. Kifer (2007). *RIF Core Design*, W3C Working Draft, March 30, 2007 `http://www.w3.org/TR/rif-core/`

[2] H. Boley, S. Tabet and G.Wagner (2001) Design Rationale of RuleML: A Markup Language for Semantic Web Rules,*In Proc. of Int. Semantic Web Working Symposium (SWWS)*, Stanford University, California, USA.

[3] E.Friedman-Hill (2003), *Jess in Action - Rule-Based Systems in Java*, Manning Publications Co.

[4] N. E. Fuchs, U. Schwertel, R. Schwitter (1997), Attempto Ů Englisch als (formale) Spezifikationssprache, In: F. Bry, B. Freitag, D. Seipel (eds.), *Proceedings of the Twelfth Workshop on Logic Programming WLP*, Munich.

[5] M. Kifer, G. Lausen and J. Wu, Logical Foundations of Object-Oriented and Frame-Based Languages, *Journal of ACM*, May 1995.

[6] Oana Nicolae, Adrian Giurca and Gerd Wagner.*On Interchange between JBossRules and Jess* Proceedings of 1st International Symposium on Intelligent

and Distributed Computing (IDC 2007), October, 2007.

[7] OMG (2005). *OCL 2.0 Specification*, June 06, 2005, `www.omg.org/docs/ptc/05-06-06`

[8] OMG (2007). *Production Rule Representation Ver. 1.0* , March 5, 2007, `http://www.omg.org/docs/bmi/07-03-05.pdf`

[9] OMG (2006). *Semantics of Business Vocabulary and Business Rules (SBVR)*, `http://www.omg.org/docs/dtc/06-03-02.pdf`

[10] G. Wagner, A. Giurca and S. Lukichev (2005), R2ML: A General Approach for Marking up Rules, *Dagstuhl Seminar Proceedings 05371, In F. Bry, F. Fages, M.Marchiori, H. Ohlbach (Eds.) Principles and Practises of Semantic Web Reasoning*, ISSN:1862-4405.