SweetJess: Translating DamlRuleML to Jess

Benjamin N. Grosof [1], Mahesh D. Gandhe [2], Timothy W. Finin[2]

[1] MIT Sloan School of Management,
50 Memorial Drive, Cambridge, MA 02142, USA
bgrosof@mit.edu
http://www.mit.edu/~bgrosof/

[2] Institute for Global Electronic Commerce,
Department of Computer Science,
University of Maryland Baltimore County,
1000 Hilltop Circle, Baltimore MD 21250 ,USA
{mgandh1, finin}@cs.umbc.edu
http://www.csee.umbc.edu/~mgandh1/
http://www.csee.umbc.edu/~finin/

Abstract. We describe the design of SweetJess, our new system for Semantic Web rules in Jess. The SweetJess approach makes four main new contributions. First, we show how to translate from rules in the Situated Courteous Logic Programs (SCLP) knowledge representation, syntactically encoded in RuleML, into Jess rules, and likewise to translate from a broad but restricted case of Jess rules into SCLP RuleML. SCLP is expressively powerful and features prioritized conflict handling and procedural attachments. The translation applies to a broad but restricted case in each direction, and preserves semantic equivalence – i.e., for a given rulebase, the same conclusions are entailed. Second, we give an architecture to perform (a broad case of) SCLP RuleML inferencing using the Jess rule engine. Third, rather straightforwardly, we have developed a DAML+OIL ontology for (SCLP) RuleML itself. The resulting syntax for RuleML is called "DamlRuleML"; the DAML+OIL is simply used as "syntactic sugar" for encoding of RuleML. Fourth, our translation newly enables bi-directional inter-operability, via RuleML, between Jess — a "reactive" rule system — and multiple other heterogeneous rule systems, including Prologs and relational database systems ("derivational" rule systems), for which translation to RuleML has already been shown and among which there are several existing translation tools (e.g., our SweetRules system). It thereby moves a discernible step closer to the Semantic Web's vision of wide knowledge sharing and integration among intelligent applications, e.g., where rules are already often deployed for e-business policies and workflow. Prototyping of SweetJess is in progress. We intend to make the implementation publicly available.

1 Introduction and Overview

The overall problem we address is how to enable inter-operability between heterogeneous rule systems (including relational database systems as an important special case), and between heterogeneous intelligent applications that make use of such rule systems. Rules are widely deployed today to represent and automate e-business policies and workflows, for example. Practical advances in such inter-operability would offer the promise to greatly facilitate program-to-program knowledge sharing and integration, and thereby to stimulate a global virtuous circle of growing value creation in e-business. In short, we seek to realize business intelligence on the Semantic Web.

In this paper, we describe the design of SweetJess, our new system for inter-operability of rules between RuleML and Jess. RuleML [1] [2] is an emerging industry standard for XML rules that we (first author) co-lead, being pursued in informal cooperation with the World Wide Web Consortium (W3C) [3] and the DARPA Agent Markup Language (DAML) Program [4]. Rules indeed are part of the announced mission of the W3C's Semantic Web Activity [5]. Jess, acronym for "Java Expert System Shell", is a popular open-source rule system [6].

SweetJess is part of our (first author's) larger system SWEET, acronym for "Semantic WEb Enabling Technology". SWEET also includes SweetRules [2], a system for RuleML inferencing and translation, and SweetDeal [7][8][9], an approach to rule-based contracting that builds upon SweetRules, RuleML, and process ontologies in DAML+OIL, e.g., to represent deals about Web services. Our previous SweetRules prototype was the first to implement SCLP RuleML inferencing and also was the first to implement translation of (SCLP) RuleML to and from multiple heterogeneous rule systems. "SCLP" stands for "Situated Courteous Logic Programs". The SCLP case of RuleML is expressively powerful – SCLP features prioritized conflict handling and classical negation as well as procedural attachments for sensing (testing rule antecedents) and effecting (performing actions triggered by conclusions). SweetRules enables bi-directional translation from SCLP RuleML to: XSB, a Prolog rule system [11]; Smodels, a forward logic-program rule engine; the IBM CommonRules rule engine, a forward SCLP system [12]; and Knowledge Interchange Format (KIF, a.k.a. "CommonLogic"), an emerging industry standard for knowledge interchange in classical logic [13].[1] SweetJess aims to complement and extend SweetRules by providing additional capabilities including translation to Jess.

The first new contribution we describe here is that we have developed, rather straightforwardly, a DAML+OIL [14] ontology for RuleML itself – in particular, for the SCLP case of RuleML. The resulting syntax for RuleML is called "DamlRuleML". Actually, DamlRuleML (with its DAML+OIL ontology) is not limited to SweetJess; it is useful even when Jess is not involved. However, this DAML+OIL ontology essentially constitutes "syntactic sugar"; it merely enables a DAML+OIL syntactic encoding of RuleML.

---

[1] SweetRules is built in Java. It uses XSLT [22] and components of the IBM CommonRules library.

The second new contribution of the SweetJess approach is more fundamental, interesting, and meaty. We show how to translate from a broad but restricted case of SCLP DamlRuleML into Jess rules, and likewise to translate from a broad but restricted case of Jess rules into SCLP DamlRuleML. Actually, this translation does not at all depend on the DAML ontology/encoding for RuleML; it applies equally well to plain RuleML. The translation preserves semantic equivalence – i.e., for a given rulebase, the same conclusions are entailed. Likewise, for a given rulebase, the same side-effectful actions (triggered by conclusions) are performed when the rulebase is executed (i.e., when the rules are "run").

The third new contribution of the SweetJess approach builds upon the translation. We give an architecture to perform (a broad case of) SCLP DamlRuleML inferencing using the Jess rule engine. Again, this does not depend at all on the DAML ontology/encoding for RuleML; it applies equally well to plain RuleML.

Translating the prioritized conflict handling (Courteous aspect) of RuleML is a particular hurdle, due to Jess' limitations in that regard; we surmount it by utilizing a Courteous Compiler component. The Courteous Compiler "compiles away" the courteous aspect of an input rulebase, transforming it into a semantically equivalent rulebase that does not contain the Courteous expressive features (priorities and mutual exclusion integrity constraints. The IBM CommonRules library provides a Courteous Compiler, for example.[2]

The fourth new contribution of the SweetJess approach is to enable bi-directional inter-operability, via RuleML as an interlingua, between Jess and multiple other heterogeneous rule systems, including Prologs and relational database systems for which translation to RuleML has already been shown, and for which there are existing translation tools (e.g., in SweetRules and our other earlier work [7]). In particular, as we discussed earlier, SweetRules already enables bi-directional translation from SCLP RuleML to: XSB; Smodels; IBM CommonRules; and KIF/CommonLogic. The overall approach to such translation was first given by us in [7]. The RuleML website lists additional translation tools as well. For a given rule system such as Jess, the software engineering effort of specification, design and implementation of translation to multiple other rule systems is greatly eased by use of a single intermediate interlingua, i.e., the emerging RuleML standard.

Jess is a representative member of one group of currently commercially important (CCI) rule systems: namely, production rule systems descended from OPS5 [16], which in turn are closely related to event-condition-action (ECA) rule systems [17]. These systems primarily employ forward chaining (rather than backward), and their applications heavily rely on their capabilities for procedural attachments. This group is sometimes called "reactive" for short; often, rules are run in response to the arrival of knowledge-base updates consisting of facts (or "events"). Another quite distinct group of CCI rule systems is comprised of Prolog systems [18], together with SQL-type relational database systems (RDB) [17]. The core of SQL RDB's – relational algebra and Datalog – is well-known theoretically to be very closely related to pure

[2] SweetJess' design and SweetRules' current implementation make use of IBM CommonRules' Courteous Compiler component.

Prolog. Systems in this second group (sometimes called "derivational") primarily employ backward chaining (rather than forward), i.e., query-answering.

The fifth new contribution is that our translation is particularly interesting in that (to our knowledge) it is the first to go for a broad expressive case between the two groups (reactive vs. Prolog/SQL derivational).

The sixth new contribution of our translation effort is to compare the expressive capabilities of each rule system and its underlying fundamental knowledge representation, and in particular to bring out several limitations of Jess relative to SCLP RuleML.

In continuing the overall SweetRules approach by laying these new foundations for inter-operability, the SweetJess approach thereby moves a discernible step closer to the Semantic Web's vision of wide knowledge sharing and integration among intelligent applications, e.g., where rules are already often deployed for e-business policies and workflow, and SQL RDB's are ubiquitous.

Prototyping of SweetJess, and testing of the DamlRuleML ontology, are in progress. The prototype is implemented in Java and makes use of tools for XML, RDF [19], DAML+OIL, and RuleML. We intend to make the implementation publicly available.

The remainder of this paper is organized as follows. In section 2, we describe SweetJess' architecture to perform (a broad case of) SCLP DamlRuleML inferencing using the Jess rule engine. In section 3, we describe DamlRuleML. In section 4, we review the Courteous extension of declarative Logic Programs (LP). This enables prioritized conflict handling and also a limited form of classical negation. In section 5, we review the Situated extension of LP. This enables procedural attachments for sensing (testing antecedents) and effecting (performing actions triggered by conclusions). In section 6, we review Jess rules. In section 7, we come to the heart of the matter: we describe how to translate rules from SCLP DamlRuleML to Jess. In section 8, we describe how to translate back from Jess to SCLP DamlRuleML. We have space only to describe this for the special case of facts, e.g., conclusions from the Jess engine. (See, rather, the extended version of this paper for full details.) In section 9, we wind up with some discussion, including directions for future work.

Note that for reasons of space limitations, we assume that the reader is familiar with the basics of RuleML and DAML+OIL.

2 SweetJess' Architecture for DamlRuleML Inferencing Via Jess

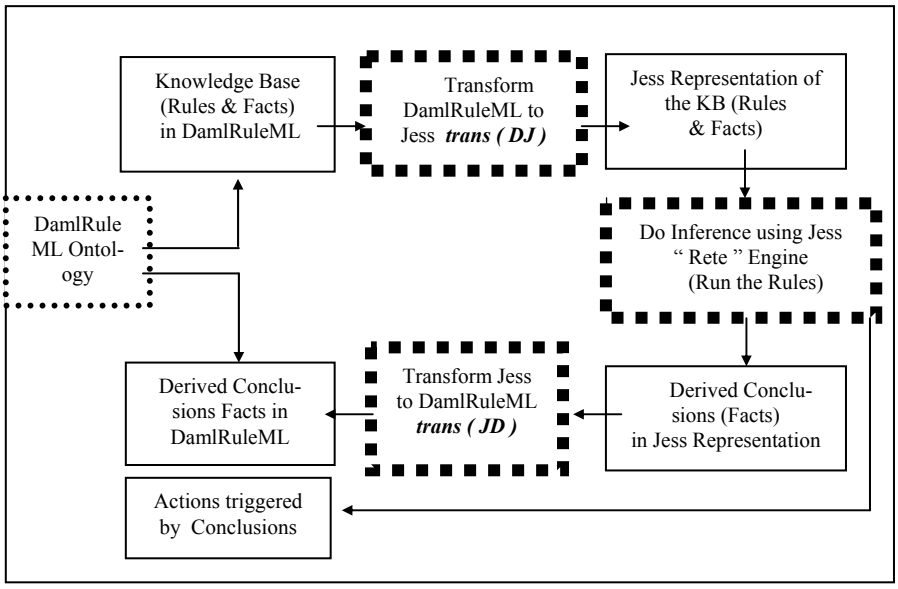

Figure 1: SweetJess' Architecture for DamlRuleML Inferencing via Jess

The bi-directional translation between (Daml)RuleML and Jess has several potential uses, as discussed in the Introduction and Conclusions sections. By way of motivation for the details in the rest of this paper, however, we largely focus on one particular use: to perform DamlRuleML inferencing via Jess. Figure 1 shows SweetJess' architecture for this. By "DamlRuleML inferencing", we mean inferencing from premise DamlRuleML rules to derive DamlRuleML conclusions and related procedural actions that are triggered from those conclusions via procedural attachments ("effectors" in Situated LP, described in section 5). By "via Jess", we mean that Jess is used as an engine to "run" the rules. SweetJess starts with a knowledge base (KB) of rules and facts in DamlRuleML. (Note that conceptually, LP facts are viewed as a special case of rules.) Then these are transformed by a SweetJess translator component into a set of Jess rules and facts. This transformation is called *trans(DJ)*. Inferencing is then performed using Jess' rule inference engine (which is based on the Rete algorithm), i.e., the rules are "run". This generates a set of derived conclusions (facts) in the Jess representation. Then these facts are transformed by a SweetJess translator component into a set of DamlRuleML facts. This inverse direction transformation is called *trans(JD)*. The result is a set of DamlRuleML conclusion facts entailed by the original DamlRuleML premise rules and facts. When the rules are run in the Jess engine, a set of actions, triggered by conclusions, is also performed. These actions are invocations of attached procedures, i.e., side-effectful calls to Java methods. These actions are those sanctioned by the Situated ("effecting") aspect of the

semantics of the premise DamlRuleML rules and facts. The DAML+OIL ontology for RuleML itself is used, in a background fashion, when encoding rules and facts in DamlRuleML. The transformations trans(DJ) and trans(JD) impose some expressive restrictions on their input, which we will describe later. Algorithmically, the transformer component (for each transformation) recognizes whether its input ruleset meets those restrictions. For the results of inferencing, only facts need be translated via trans(JD) from Jess to DamlRuleML. Our trans(JD) translation handles general rules not just facts, but we will not have space in this paper to describe it in detail beyond the case of facts; see, rather, the extended version of this paper.

3. DamlRuleML : DAML + OIL Ontology for RuleML

In this section, we describe DamlRuleML. DAMLRuleML is fairly straightforward. Next, we give, as examples of our DAML+OIL ontology for RuleML, its specification of two classes "Ind" (Individual, i.e., a 0-ary logical function symbol) and "Imp" (Implication rule, i.e., an if-then).

```
********** DAML+OIL Class for Individual **********
<daml:Class  rdf:ID="Ind">
   <rdfs:label>Ind</rdfs:label>
   <rdfs:comment>Ind stands for Individual, i.e., a 0-ary logical
                 function symbol.
    </rdfs:comment>
</daml:Class>
```

```
********** DAML+OIL Class for Implication **********
<daml:Class  rdf:ID="Imp">
      <rdfs:label>Imp </rdfs:label>
      <rdfs:comment>Imp stands for Implication rule, i.e., an if-then
                    rule.
       </rdfs:comment>
      <rdfs:subClassOf>
             <daml:Restriction daml:minCardinality="0">
                 <daml:onProperty  rdf:resource="#_rlab"/>
             </daml:Restriction >
       </rdfs:subClassOf>
       <rdfs:subClassOf>
             <daml:Restriction daml:Cardinality="1">
                 <daml:onProperty  rdf:resource="#_head"/>
             </daml:Restriction >
       </rdfs:subClassOf>
     <rdfs:subClassOf>
             <daml:Restriction daml:Cardinality="1">
                 <daml:onProperty  rdf:resource="#_body"/>
             </daml:Restriction >
       </rdfs:subClassOf>
</daml:Class>
```

For the complete DAML + OIL specification of DamlRuleML see http://gentoo.cs.umbc.edu/~mgandh1/#Ontology

Next, we give two examples of knowledge represented in DamlRuleML: a fact and a rule. Note that in RuleML, as is usual in declarative logic programs, a fact is simply a special case of a rule: one with an empty body. A rule has an optional label (name). As we will see later, a rulebase also has an optional label (name).

Example 1: DamlRuleML Fact
"Allan is a shopper"

********** DamlRuleML Representation **********

```
<damlRuleML:fact>
    <damlRuleML:_rlab>fact107</damlRuleML:_rlab>
        <damlRuleML:_head>
          <damlRuleML:atom>
             <damlRuleML:_opr>
                <damlRuleML:rel>shopper<damlRuleML:rel>
             </damlRuleML:_opr>
             <damlRuleML:ind>Allan</damlRuleML:ind>
           </damlRuleML:atom>
        </damlRuleML:_head>
</damlRuleML:fact>
```

Example 2: DamlRuleML Rule :
"The discount for a shopper is 5 percent if his/her spending history is loyal."

********** DamlRuleML Representation **********

```
<damlRuleML:imp>
   <damlRuleML:_rlab>
      <damlRuleML:ind>discountRule</damlRuleML:ind>
   </damlRuleML:_rlab>
   <damlRuleML:_body>
     <damlRuleML:andb>
        <damlRuleML:atom>
           <damlRuleML:_opr>
              <damlRuleML:rel>shopper<damlRuleML:rel>
           </damlRuleML:_opr>
           <damlRuleML:var>Cust</damlRuleML:var>
        </damlRuleML:atom>
        <damlRuleML:atom>
          <damlRuleML:_opr>
             <damlRuleML:rel>spendingHistory<damlRuleML:rel>
          </damlRuleML:_opr>
          <damlRuleML:var>Cust</damlRuleML:var>
          <damlRuleML:ind>loyal</damlRuleML:ind>
        </damlRuleML:atom>
      </damlRuleML:andb>
    </damlRuleML:_body>
   <damlRuleML:_head>
    <damlRuleML:atom>
      <damlRuleML:_opr>
        <damlRuleML:rel>giveDiscount<damlRuleML:rel>
      </damlRuleML:_opr>
       <damlRuleML:ind>percent5</damlRuleML:ind>
       <damlRuleML:var>Cust</damlRuleML:var>
    </damlRuleML:atom>
   </damlRuleML:_head>
  </damlRuleML:imp>
```

For a generic representation of a DamlRuleML Fact and Rule, see the extended version of this paper.

4. Courteous Logic Programs (CLP): Review

Courteous Logic Programs (CLP) is an expressive super-class of Ordinary Logic Programs (OLP)[3]. Next, we briefly review CLP. Each rule has an optional rule label. This is used as a handle for specifying prioritization information. Each label is a logical term, e.g., a logical 0-ary function constant. The "overrides" predicate is used to specify prioritization. "overrides(lab1,lab2)" means that any rule having label "lab1" is higher priority than any other rule having label "lab2". "overrides" is syntactically reserved, but otherwise is treated as an ordinary predicate. In particular, "overrides" can itself be the subject of inferencing, The scope of what is conflict is specified by pair-wise mutual exclusion statements called "mutex's". E.g., a mutex (or set of mutex's) might specify that there is at most one amount of discount granted to any particular customer. Any literal may be classically negated. There is an implicit mutex between p and classical-negation-of-p, for each p, where p is a ground atom, atom, or predicate.

Next, we give an example of a CLP, having 3 rules and 1 mutex. The syntax of XML generally, and both RuleML and DamlRuleML in particular, is fairly verbose for humans to read. For ease of human-readability, as well as to save paper space, we give our RuleML examples in the Prolog-like "SCLPfile" syntax of IBM Common-Rules V3.0, which maps straightforwardly to DamlRuleML. ";" ends a rule statement. The prefix "?" indicates a logical variable. "/* … */" encloses a comment. "//" prefixes a comment line. "<…>" encloses a rule label (name). "{…}" encloses a rules module. Rule labels identify rules for editing and prioritized conflict handling, for example to facilitate the modular modification of contract provisions. For an extended version of this syntax, see [21]. The first rule below is the same rule as example 2 in section 3, for clarity.

```
/*** If the Customer has a Loyal Spending History then give him a 5% Discount
****/
<steadySpender>
    if       shopper(?Cust) and  spendingHistory(?Cust, loyal)
    then     giveDiscount(percent5, ?Cust);

/***  If the Customer was Slow to Pay last year then grant him NO Discount  *****/
<slowPayer>
    if       slowToPay(?Cust, last1year)
    then     giveDiscount(percent0, ?Cust);
```

[3] OLP is also sometimes called "normal logic programs", or "general logic programs" as in [20] which provides a helpful review of declarative LP. OLP is roughly pure Prolog without built-ins, but not limited to backward direction of inferencing.

```
/******  prioritization fact:  SlowPayer  rule Overrides SteadySpender Rule
*******/
  overrides(slowPayer, steadySpender);

/********  The amount of the Discount given to a customer is Unique  *********/
  MUTEX         giveDiscount(?X, ?Cust) and giveDiscount(?Y, ?Cust)
                      GIVEN          notEquals(?X, ?Y) ;
```

For the DAMLRuleML representation of this ruleset, see the extended version of this paper.

5: Situated Logic Programs: Review

The Situated extension of (Courteous or Ordinary) Logic Programs allows actions and queries to be performed by procedural attachments. SLP uses effector and sensor statements to specify these, as in the example below.

The effector statements in a SLP (e.g., example below) each associate a pure-belief predicate, e.g., shouldInformCustomer, with an external procedure (here, a Java method), e.g., orderMgmt.request.mods.ack. During rule inferencing (more precisely, during rule execution), when a conclusion is drawn about the predicate, e.g., "should-InformCustomer(cancelRequest4216, accepted)" if the rule below was fired successfully, then the external procedure is invoked as a side-effectful action, e.g., the method orderMgmt.request.mods.ack is called with its parameters instantiated to "(request1049, accepted)".

The sensor statements in a SLP (e.g., example below) each associate a pure-belief predicate, e.g., receivedBefore, with an external procedure (here a Java method), e.g., orderMgmt.request.earlierReceiptDate. During rule inferencing/execution, when a rule antecedent condition (i.e., a literal in the rule's ``if'' part) is tested, e.g., received-Before(cancelRequest4216,?Day) in the rule below, then the external procedure is queried to provide information about that condition's truth. More precisely, the external procedure is queried for its answer bindings since the condition may contain logical variables. Some external sensor procedures require some or all of the arguments to be bound (i.e., fully instantiated) at the time that external procedure is invoked. A sensor statement thus includes a *binding pattern* that specifies such requirements.

For example, consider an external procedure myCompany.BluePages.getPhoneNumber, provided by a company phone directory application, that has two arguments, where the first is a person name and the second is a phone number. It has an associated binding pattern that requires the first argument to be bound but permits the second argument to be unbound. When invoked with the first argument bound to "Fred.Green" and the second argument a free variable

("?X"), it returns the binding 617-555-9876 for that variable. In the example below, the sensor procedure orderMgmt.Request.earlierReceiptDate requires both of its arguments to be bound when it is invoked.

Some sensor statements, e.g., for the predicate lessThanOrEqual, correspond to what in Prolog (or many other commercial rule systems) are "built-ins", utility procedures provided as a standard package with the rule system rather than specified by a particular individual user/application.

```
/* Notify customer if order cancellation request was received in time to be accepted */
   <rule_526>
      if    deadlineToCancel(order4215, ?Day) and
            receivedBefore(cancelRequest4216, ?Day)
      then  shouldInformCustomer(cancelRequest4216, accepted);

/*** effector for informing a customer about status of order modification request **/
      Effector: shouldInformCustomer
      Class: orderMgmt.Request.mods
      Method: ack
      path: "edu.cs.umbc.SLP.examples.orderMgmt.aprocs";

   /*** sensor statement for the receivedBefore predicate ******/
      Sensor: receivedBefore
      Class:orderMgmt.Request
      Method: earlierReceiptDate
      BindingRequirement: (BOUND,BOUND)
      path: "edu.cs.umbc.SLP.examples.orderMgmt.aprocs";
```

For the DAMLRuleML representation of this ruleset, see the extended version of this paper.

6. Jess Rules: Review

6.1 Overview of a Jess Fact

A Jess fact has the following kind of form (we can view this roughly as a generic template):

(assert (predicateName const1 const2 …
(jMethodName param1 param2 paramM) … constN …))

A Jess fact begins with the "assert" keyword, followed by an expression roughly syntactically similar to an LP ground atom. Note that Jess does *not* conceptually view a fact as a special case of a rule (unlike LP). A Jess fact corresponds essentially to a ground atom in LP. A fact may contain (an appearance of) a "*JessMethod*" (in our terminology; Jess terminology dubs it a "Function", but that's fairly confusing to

one accustomed to LP terminology and concepts). A JessMethod is somewhat similar to a logical function (a <ctor> element in RuleML), and syntactically constructs an expression somewhat similar to a logical term (a <cterm> element in RuleML). For example, the template example above contains the sub-expression

(jMethodName param1 param2 paramN)

in which jMethodName is a JessMethod. However, a JessMethod appearing in a fact (or in a "then" part of a rule – see next sub-section) is evaluated on its arguments immediately at the time of loading (or inferring) a fact in which it appears. Essentially, a JessMethod is thus a Java method that gets called as a procedure, rather than a true constructor. Generally, in the context of a fact, a JessMethod call returns a single value that can be viewed as an ind (rather than, say, a general-form cterm) in RuleML/LP.

In a different context, however, a JessMethod call may return a boolean (truth value), as in Jess rule body "Test C.E." expressions described in the next sub-section.

6.2 Overview of a Jess Rule

A Jess rule has the following kind of form (we can view this roughly as a generic template):

Syntax	**…**	**and what it specifies**
(defrule ruleName	⟶	Name of the Rule
(predicate1 constant1 ?boundVariable1)	⟶	Pattern in the Body
(test (jMethod1 constant2 ?boundVariable1))	⟶	JessMethod (Sensor) in Body
=>		
(jMethod2 (constant3 ?boundVariable1))	⟶	JessMethod (Effector) in Head
)		

A Jess rule definition begins with the "defrule" keyword followed by a rule name, and has two further parts, an "if" part on its left hand side (LHS) and a "then" part on its right hand side (RHS), separated by the "=>" symbol which roughly means implication. In the LHS of a Jess rule, a "pattern" (in Jess terminology) that matches facts, corresponds to an atom in an LP rule body. Several such "patterns" may appear in a given Jess rule; they are AND'ed together, just as atoms may be in an LP rule body. A LHS rule pattern may *not* contain a JessMethod appearance.

In addition, another kind of expression can appear in the LHS of a Jess rule: a "TEST Conditional Element" (in Jess terminology, or "Test C.E." for short) which is constructed syntactically using the reserved keyword "test" (as in the template example above). A "Test C.E." corresponds in Situated LP essentially to an atom whose predicate has an associated attached-procedure for sensing (a "sensor atom"). More precisely, the "Test C.E." expression specifies a sensing procedure call to a JessMethod (e.g., jMethod1 in the template example above). "Test C.E." expressions can be AND'ed together with each other and with "patterns", just as "patterns" can. The JessMethod within a "Test C.E." expression must return a boolean value. A "Test

C.E." expression's arguments must all be fully bound at the time the JessMethod is called.

The RHS of a Jess rule is an expression formed by a JessMethod. This JessMethod (jMethod2 in the generic template above) may be "assert", or it may be something else. If it is "assert", then the rule is essentially a pure-belief rule – when fired the rule generates a conclusion fact rather than a side-effectful action. In this case, the RHS of the Jess rule also is similar to a Jess fact but with variable bindings supplied as in the usual manner for rules. If the JessMethod in the RHS of the Jess rule is something other than "assert", however, then when fired the rule essentially generates an effector call to that JessMethod – moreover, without generating any conclusion fact. (Situated LP behavior, by contrast, always generates a conclusion fact even when an effector call is made.) This effector call is typically side-effectful.

For efficiency in "pattern" matching -- especially to handle updates to its working set of facts -- Jess uses an algorithm known as the Rete (Latin for "net") algorithm. Computational complexity per iteration of this algorithm is at most linear in the size of the fact base.

7. Transforming DamlRuleML to Jess: *trans (DJ)*

7.1 Input and Output

The transformation trans(DJ) takes as input a DamlRuleML rulebase, which has file extension .daml . This is marked up according to the DamlRuleML ontology, of which the current version is V0.2. The output of this transformation is a Jess knowledge base – i.e., a .jess batch file. This batch file contains facts and rules which can be directly fed to a Jess Rete engine, of which the current version is V6.1.

7.2 Fact

In DamlRuleML, a fact has an optional rule label (name). A fact in Jess has a unique Fact Id which is generated by the system upon loading, and is only accessible to the system rather than being explicit in its Jess representation. The DamlRuleML fact's rule label, if present, is thus lost by the transformation.

Example: The equivalent Jess fact corresponding to the DamlRuleML fact in Example 1 of Section 3 ("Allan is a shopper") is as follows :

********** Equivalent Jess Representation **********

```
(assert (shopper Allan) )
```

For the generic transformation of a DamlRuleML Fact to a Jess fact, see the extended version of the Paper.

7.3 Rule

The basic approach is translating a DamlRuleML rule is fairly straightforward: a corresponding Jess rule is generated whose body corresponds to the DamlRuleML body, and whose head corresponds to the DamlRuleML head. Each DamlRuleML rule body atom is translated into a corresponding pattern. The DamlRuleML head atom is translated into an assert of the corresponding pattern.

In general, (Daml)RuleML may permit a conjunction of atoms in a rule head. For now, for simplicity's sake, we expressively restrict the input ruleset to permit only a single atom in the head of a rule. See the extended version of this paper for more discussion on this point.

A subtlety is that Jess lacks the semantic equivalent of a non-zero-arity DamlRuleML constructor (logical function, i.e., a <ctor> element in RuleML) and, therefore, of a DamlRuleML logical term (i.e., a <cterm> element in RuleML). Of course Jess does have the semantic equivalent of an ind. The closest thing to a non-zero-arity constructor is a JessMethod, but that is always evaluated on its arguments; the semantics of a non-zero-arity constructor, however, essentially correspond to not evaluating it. For the time being, we thus expressively restrict the input DamlRuleML to prohibit non-zero-arity constructors. In LP terminology, this is also known as the "Datalog" restriction.

Another subtlety is that in a DamlRuleML rule the rule label (name) is optional and need not be unique. Jess requires a name for every rule, which moreover must be unique (if not, the last-loaded with that name blows away any previously-loaded rule with the same name). For the time being, we thus expressively restrict the input DamlRuleML rules' labels not to coincide with each other. Hence if the DamlRuleML rule has a label then in translation it becomes the name of the corresponding Jess rule. If the DamlRuleML Rule does not have a label, then the Sweet-Jess translator generates a new rule name for the translated rule in Jess.

A third subtlety arises in translating sensor statements; these may modify how a rule mentioning a "sensor" predicate is translated so that a "Test C.E." element replaces a body pattern; see sub-section 7.7 below for details.

Example: After transforming the DamlRuleML rule in Example 2 of Section 3, the corresponding Jess rule is:

********** Equivalent Jess Representation **********

```
(defrule discountRule
 (shopper ?Cust)
 (spendingHistory ?Cust loyal)
 =>
 (assert (giveDiscount percent5 ?Cust) ) )
```

For the generic transformation of a DamlRuleML Rule to a Jess Rule, see the extended version of this paper.

7.4 Negation-as-Failure (NAF)

Negation in Jess – the "~" operator – is implemented as negation-as failure (NAF). Thus negation-as-failure in DamlRuleML translates into it straightforwardly. As is well-known in the LP literature, NAF can cause semantic trouble by interacting with cyclic dependencies ("recursion") among rules. For the time being, we thus expressively restrict the input DamlRuleML rulebase to be "stratified" (terminology of the LP literature – see [20] for a helpful review). This is a fairly broad case which can be recognized in time O(n·log(n)) where n= |input rulebase|. Note that NAF-free is a special case of stratified.

7.5 Courteous Prioritized Conflict Handling

DamlRuleML supports representation of Courteous LP's expressive features for prioritized conflict handling which include mutex integrity constraints. Jess, however, does not support CLP directly.

7.5.1 Use Courteous Compiler

As we have shown in previous work [7] [12], there is a way around this incapacity of Jess: transform the CLP into a semantically equivalent Ordinary LP, via a Courteous Compiler. IBM CommonRules and SweetRules make use of a Courteous Compiler, for example. For an input DamlRuleML rulebase that contains courteous features (notably, mutex's), we thus refine the SweetJess architecture accordingly: as a first step in trans(DJ), the input DamlRuleML rulebase (.daml) is transformed via a Courteous Compiler (CC) component into a different, but semantically equivalent, DamlRuleML rulebase that no longer contains the courteous features. This post-CC rulebase is then run through the basic-case translator for trans(DJ).

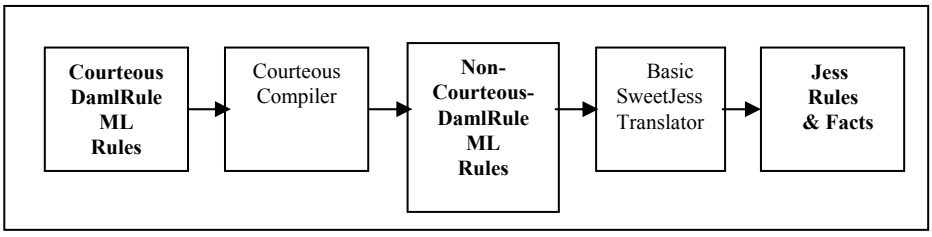

Figure 2: Transformation of Rules with Courteous features

Note that the basic-case DJ-translator's expressive restrictions, in particular the ctor-related restriction (Datalog) and the NAF-related restriction (stratified) discussed above in 7.3 and 7.4, must be met in the post-CC DamlRuleML rulebase.

Note also that the Courteous Compiler has (as usual when employing it) an associated simpler CC-inverse transformation required for tightest-possible semantic equivalence.[4]

7.5.2 Salience

In Jess, each rule has a property called salience that is a kind of control-dependent rule priority. Activated rules of the highest salience will fire first, followed by rules of lower salience. To force certain rules to always fire first or last, rules can include a salience declaration. Jess manual discourages the use of the Salience feature for two reasons: first it is considered bad style in rule-based programming to try to force rules to fire in a particular order. Secondly, use of salience will have a negative impact on performance, at least with the built-in conflict resolution strategies. For the time being, since salience is a fairly control-dependent expressive feature, we do not attempt to exploit it to represent Courteous-style prioritization.

7.6 Classical Negation

The Courteous features of SCLP DamlRuleML also permit classical negation of atoms, which is not permitted in Ordinary LP (nor in non-Courteous Situated LP). The Courteous Compiler step, discussed above in 7.5, thus also enables appropriate translation of classical negation that may be present in an input DamlRuleML rulebase.[5]

7.7 Situated LP Procedural Attachments for Sensing and Effecting

The "Test C.E." and JessMethod features of Jess provide a target for translation of sensor and effector statements in DamlRuleML. The "Test C.E." is useful only for sensors, but the JessMethod feature is useful for effectors as well. Jess allows JessMethods to be "user-defined functions" (in the general Java programming sense) which can be invoked either in a rule head (for effecting) or in a rule body (for sensing). trans(DJ) thus defines a new Jess function for each input sensor or effector statement. However, Jess requires all logical variables passed as arguments to these JessMethod's to be (fully) bound (to a constant value). If the JessMethod is being used for effecting, this is similar to the semantics of Situated (C)LP. But if the Jess-

4 This is because the CC introduces some extra predicates to represent classically negated versions of predicates and "adorning" predicates to represent intermediate phases of prioritized argumentation during prioritized conflict handling.

5 Note that Courteous LP's semantics have no trouble combining classical negation with negation-as-failure, in a manner similar to the literature on "extended" LP (see [20] for a helpful review of LP literature). The semantics of Courteous LP relies on the well-founded semantics for LP, and ensures consistency between the two concepts of negation (classical negation of p entails NAF of p, but not vice versa).

Method is being used for sensing, this is indeed a strict restriction. In short, trans(DJ) must thus expressively restrict the input SCLP DamlRuleML accordingly: in all sensor statements, the Binding-mode must be "Bound" (rather than "Free") for all of the parameters of the predicate/attached-procedure. We call this the *"all-bound-sensors"* expressive restriction. There are some other mechanics and subtleties of how to pass the path, classname, and methodname to the JessMethod, but space prevents us from describing them here; see the extended version of this paper for details. Some of these details are in evidence in the example below, however.

An effector statement associating a predicate *p* with an attached procedure *q* is translated into a rule whose "if" part is an open atom in the predicate *p* and whose "then" part is a JessMethod that invokes the attached procedure *q*. A sensor statement, associating a predicate *p* with an attached procedure *q*, is translated more indirectly. Its presence in the input results in modifying the translation of every rule *r* whose body mentions the predicate *p*: a translated version of rule *r* is generated whose body has a "Test C.E." that invokes *q*, rather than a (body) pattern in *p*. If there are two sensor statements for the same predicate *p*, one associating it attached with procedure *q1* and another with attached procedure *q2*, then Jess rules are generated from *r* that invokes *q1* and that invoke *q2*. If there are more than two sensor statements per predicate, the translation is handled likewise. See the extended version of this paper for further details; note that there are some semantically equivalent alternative ways to define the translation. Next, we give an example of one way to define it.

Example: After transforming the example SLP ruleset in section 5, the corresponding Jess KB is:

```
(defrule  rule_526
    (deadlineToCancel order4215 ?Day)
    (test (generic_sensor_with_arity_2 cancelRequest4216 ?Day
                           orderMgmt.Request earlierReceiptDate) )
  =>
    (assert (shouldInformCustomer cancelRequest4216 accepted
                             orderMgmt.Request.mods ack) ) )

(defrule  effect_shouldInformCustomer_via_orderMgmt_Request_mods_ack
    (shouldInformCustomer ?request ?status)
  =>
    (generic_effector_with_arity_2 ?request ?status
                             orderMgmt.Request.mods  ack) )


(deffunction generic_sensor_with_arity_2
   (?param1 ?param2  ?className ?methodName)
     (bind ?vt (new ?className))
     (call ?vt ?methodName  ?param1 ?param2 )
   return (new Value (true))  )


(deffunction generic_effector_with_arity_2
  (?param1 ?param2 ?className ?methodName)
    (bind ?vt (new ?className))
    (call ?vt ?methodName  ?param1 ?param2)   )
```

7.8 Naming conflicts with Jess system commands

Certain syntactically reserved ("built-in") JessMethods (besides "assert") are actually Jess system commands that manipulate the knowledge base or affect the system level process of the engine. These JessMethods must be handled with caution. However, it is possible for (predicate or attached procedure) names in the input SCLP DamlRuleML to collide (presumably, inadvertently) with the names of these Jess system commands. For the time being, the translation trans(DJ) thus expressively restricts the input SCLP DamlRuleML to prohibit the appearance of these Jess system command names. The list of these command names includes: "clear", "exit", "halt", "reset", and several more (see the extended version of this paper for details).

7.9 Lose top-level rulebase label

The Jess Rule engine has only one Global knowledge base which is created automatically by the engine at the beginning. This global knowledge base cannot be named. In the transformation, the top-level DamlRuleML rulebase label, if present, thus is lost.

7.10 Limitations of the transformation

The trans[DJ] has various limitations . Courteous LP mutex's[6] and classical negations present in the input DamlRuleML cannot be transformed directly. SweetJess instead uses a Courteous Compiler and to transform away the Courteous features before then applying the basic-case (i.e., for non-Courteous) translation trans(DJ). In addition, we underlined above to flag several other expressive restrictions, including Datalog, stratified, all-bound-sensors, single head atom, and various naming mechanics. Furthermore, certain naming information is lost by the transformation. Finally, some inessential[7] new information is introduced by the transformation – e.g., to create new rule labels or new predicates (as part of the Courteous Compiler's transformation) or new JessMethods (for Situated sensor or effector procedures). We observe that detecting violations of these restrictions appears fairly easy computationally. [8]

---

[6] and thus prioritized conflict handling behavior

[7] formally: conservatively extending, in the logical sense, of the semantic equivalence

[8] We conjecture that it is $O(n \cdot \log(n))$ or better.

7.11 Sketch of Algorithm for Transformation trans(DJ)

1. Read the entire input RuleML .daml file and determine a list of all the facts .
2. Determine the facts which do not have naming conflicts with Jess system commands as defined earlier .
3. Create the .jess batch file, and make its first statement be "(reset)" which clears out the knowledge base of the Jess engine before the new facts and Rules are loaded into it. This asserts the fact called "Initial Fact", and is essential for Negation-as-Failure to work properly.
4. Determine whether the .daml file's rulebase employs Courteous expressive features.
5. If it does, then transform it using the Courteous Compiler.
6. Detect any violations of the expressive restrictions imposed by the transformation.
7. One by one, transform all facts. Each transformed fact is loaded into the Jess engine using "assert".
8. Read the .daml file and find the list of the predicates which have sensor statements about them, and likewise find the list of the predicates which have effector statements about them.
9. One by one, transform the sensor and effector statements. As part of this, define JessMethod's using "deffunction" statements, and Jess rules using "defrule" statements, in the .jess output file.
10. Read the .daml file and determine list of all its rules.
11. One by one, transform the rules. Each transformed rule is loaded into the Jess engine using "defrule". Hence for every rule a new statement with "defrule" is made in the .jess output file.
12. The last command of the .jess batch file is "(run)". When executed, this statement starts up inferencing by the Jess engine.

8 Transforming Jess to DamlRuleML: *trans(JD)*

8.1 Overview of the trans(JD)

The input to trans(JD) is a Jess batch (.jess) file containing facts, rules, and JessMethod definitions, that can be directly fed to the Jess Rete engine in its current version (V6.1). Output of the transformation trans(JD) is a (Daml)RuleML (.daml) file which is marked up cf. the DamlRuleML ontology (current version V0.2).

Due to space limitations, in this paper we describe trans(JD) only for the case of facts. For full details of trans(JD), including its translation of rules, see the extended version of this paper.

8.2 Transformation of a Fact (Transformation of the Jess facts)

Jess facts are defined in calls to the JessMethod "assert". To transform a Jess fact, trans(JD) obtains the inner ground atomic-looking expression by stripping off the outside "assert", and generates a DamlRuleML fact that corresponds to that inner expression. Facts in Jess have unique Fact Id which is generated by the system upon loading. Jess facts do not have an explicit label for identification. In DamlRuleML facts have an optional rule label. For the time being, we define trans(JD) to simply translate this fact id into the DamlRuleML rule label.

Next, we give an example of translating a single fact. This fact might have been derived as a conclusion by the Jess inferencing engine.

Example: "Grant Allan a 5% Discount"

********** Jess Representation **********

```
(assert (giveDiscount percent5 Allan))
```

********** Equivalent DamlRuleML Representation **********

```
<damlRuleMLML:fact>
     <damlRuleML:_rlab>fact1</damlRuleML:_rlab>
     <damlRuleML:_head>
       <damlRuleML:atom>
          <damlRuleML:_opr>
            <damlRuleML:rel>giveDiscount<damlRuleML:rel>
          </damlRuleML:_opr>
            <damlRuleML:ind>percent5</damlRuleML:ind>
            <damlRuleML:ind>Allan</damlRuleML:ind>
       </damlRuleML:atom>
     </damlRuleML:_head>
</damlRuleML:fact>
```

For the generic transformation of a Jess fact to a DamlRuleML fact, see the extended version of this paper.

9 Conclusions, Discussion and Future Work

For the main Conclusions, see the "Introduction and Overview" section, especially the list of novel contributions we gave there.

At core, our effort is not particular to RuleML or Jess, but rather between knowledge representations. Its essence is to translate from declarative SCLP to production rules, and vice versa. This continues the overall approach and vision to rules interoperability, based on SCLP in XML, that we first gave in [7].

That the translation between DamlRuleML and Jess imposes some expressive restrictions in each direction is entirely typical when engaged in defining translations between two heterogeneous rule systems (or any other kind of heterogeneous systems) – the translation handles their expressive overlap.

Another contribution of our translation effort is to discover and compare the expressive/inferencing capabilities of each rule system and its underlying fundamental KR. In particular, we discovered and highlighted some limitations of Jess as compared to SCLP, including about its ability to represent attached procedures. Jess is less expressively powerful than Situated (Courteous) LP, in that sensor arguments must be fully bound, and sensors may only return true or false; whereas in SCLP, sensor arguments may contain variables that are unbound at the time the sensor is called, and sensors may return sets of bindings (or sets of facts, viewed alternatively). Jess also can make use of Courteous prioritized conflict handling since it does not provide a comparably powerful or clean way to express prioritized conflict handling. The comparative insights emerging from the translation effort thus show the potential value of SCLP as an expressive enhancement relative to production rule systems.

The translation effort also helps to clarify what features in Jess are control-oriented and thus "impure" or "non-declarative". The translation effort furthermore suggests some ways in which Jess might (and, arguably, should) be extended so as to overcome some of its expressive limitations — in order to better support translation from and to RuleML, and/or for the sake of making Jess more powerful in itself. We suggest that the Jess development team might consider remedying some of these various current incapabilities we identified, e.g., all-bound-sensors, naming of global KB, naming of facts, etc. (Of course, it may not be so easy to do so…)

We did not have much space to discuss details about translating *rules* (not just facts) *from* Jess *to* (Daml)RuleML. One potential value of such translation, however, is to *merge* knowledge originating from knowledge bases built in Jess with knowledge originating from other rule systems, e.g., Prolog or SQL systems. Another potential value of such translation is to overcome the expressive limitations of Jess, e.g., in regard to prioritized conflict handling or procedural attachments (e.g., sensing).

Our current work includes implementation and testing of the translation mapping and the overall architecture; development of more formal theory/theorems about the semantic equivalencies including about correctness of the translation and about semantics of negation-as-failure; and integration with SweetRules. In this regard, there may be some additional, relatively minor, expressive restrictions to be added, or other relatively minor modifications needed, to ensure the correctness of the translation. The version of the translation design in this paper is penultimate, rather than finalized, in that sense.

Additional directions for future work include a closer treatment of Jess' features for rule salience (a kind of control priority) and backward chaining ("queries").

On a larger scope, our current work also includes further development of SWEET, including SCLP RuleML rules "on top of" DAML+OIL ontologies, where classes and properties from DAML+OIL are treated as unary and binary predicates that may appear in the rules, and applications in e-contracting ("SweetDeal") and finance [21].

Other interesting directions for future work include to develop more translators between RuleML and various further rule systems — database and Event-Condition-Action rule systems would be especially interesting to tackle — and to explore applications that use such translators and motivate their value.

Acknowledgements

Thanks to Said Tabet, Steve Ross-Talbot, Harold Boley, and Hoi Chan for useful discussions about Jess and reactive rules versus declarative logic programs. Thanks to anonymous reviewers for helpful comments.

References

1. Rule Markup Language Initiative. http://www.dfki.de/ruleml and http://www.mit.edu/~bgrosof/#RuleML.
2. Grosof B.N., "Representing E-Business Rules for Rules for the Semantic Web: Situated Courteous Logic Programs in RuleML". Proc. Wksh. on Information Technology and Systems (WITS '01), 2001.
3. World Wide Web Consortium. http://www.w3.org
4. DARPA Agent Markup Language Program http://www.daml.org/
5. Semantic Web Activity of the World Wide Web Consortium. http://www.w3.org
6. Jess. http://herzberg.ca.sandia.gov/jess/
7. Grosof B.N., Labrou Y., and Chan H.Y., "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML". Proc. 1st ACM Conf. on Electronic Commerce (EC-99), 1999.
8. Reeves D.M., Wellman M.P. and Grosof B.N., "Automated Negotiation From Declarative Contract Descriptions". To appear 2002 in Computational Intelligence, special issue on Agent Technology for Electronic Commerce. (Revised and extended from 2001 Autonomous Agents conference paper.)
9. Grosof B.N. and Poon T.C., "Representing Agent Contracts with Exceptions Using XML Rules, Ontologies, and Process Descriptions".
10. Web Services activity of the World Wide Web Consortium. http://www.w3.org/
11. XSB logic programming system http://xsb.sourceforge.net/
12. IBM CommonRules http://www.alphaworks.ibm.com/
13. Knowledge Interchange Format. http://logic.stanford.edu/kif and http://www.cs.umbc.edu/kif. Closely related is the new CommonLogic effort.
14. DAML+OIL (March 2001) Reference Description. http://www.w3.org/TR/daml+oil-reference
15. DamlRuleML: DAML+OIL Ontology for RuleML. http://gentoo.cs.umbc.edu/~mgandh1/
16. T. Cooper and N. Wogrin, Rule-Based Programming with OPS5. Morgan-Kaufmann Pub., 1988.
17. Ullman J.D. and Widom J., A First Course in Database Systems. Prentice-Hall, 1997.
18. Clocksin W.F. and Mellish C.S., Programming in Prolog. Springer-Verlag, 1981.
19. Resource Description Format (RDF) from W3C http://www.w3.org/
20. Baral C. and Gelfond M., "Logic Programming and Knowledge Representation", J. Logic Programming, 19-20: 73-148.
21. Grosof, B.N., and Poon, T.C., "Representing Agent Contracts with Exceptions using XML Rules, Ontologies, and Process Descriptions". Proc. Intl. Wksh. on Rule Markup Languages for Business Rules on the Semantic Web, held at 1st Intl. Semantic Web Conf., 2002.
22. XSLT (eXtensible Stylesheet Language Transformations), http://www.w3.org/Style/XSL/
23. Niemela, I. and Simons, P., Smodels (version 1). http://saturn.hut.fi/html/staff/ilkka.html.