Working Paper, version of May 2, 2003

SweetJess: Inferencing in Situated Courteous RuleML via Translation to and from Jess Rules

Benjamin N. Grosof[1], Mahesh D. Gandhe[2], and Timothy W. Finin[3]

[1] MIT Sloan School of Management,
50 Memorial Drive, Cambridge, MA 02142, USA
bgrosof@mit.edu
http://www.mit.edu/~bgrosof
[2] IBM, Business Integration Solutions
577, Airport Blvd., Suite 800, CA 94410 ,USA
maheshg@us.ibm.com
www.cs.umbc.edu/~mgandh1
[3] Department of Computer Science,
University of Maryland Baltimore County
1000 Hilltop Circle, Baltimore MD 21250 ,USA
finin@cs.umbc.edu
www.cs.umbc.edu/~finin

Abstract. We describe the innovative design of our prototyped SweetJess tool for RuleML inferencing. Our first contribution is to give a new, implemented translation from a broad but restricted case of SCLP RuleML into Jess rules, and an inverse translation from a broad but further restricted case of Jess rules into SCLP RuleML. SCLP stands for the Situated Courteous Logic Programs knowledge representation; this is expressively powerful and features prioritized conflict handling and procedural attachments. The translation is intended to preserve semantic equivalence – i.e., for a given rulebase, to entail the same conclusions. The translation often preserves semantic equivalence; in current work, we are developing formal guarantees for the equivalence, including necessary expressive restrictions in each direction. Our second contribution, building upon the translation, is a new, implemented architecture to perform (a broad case of) SCLP RuleML inferencing using the Jess rule engine. Our approach translates from SCLP RuleML rules into Jess rules, runs the Jess rule engine to generate conclusions (and actions), and then translates the concluded Jess facts back into SCLP RuleML. Our third new contribution is to enable bi-directional implemented inter-operability, via RuleML, between several other heterogeneous rule systems (e.g., XSB Prolog and IBM CommonRules) and Jess. For example, to our knowledge, this is the first tool to enable inter-operability between a Prolog and any "production"/"reactive" rule system descended from OPS5 heritage. The prototype implementation of SweetJess is publicly available for Web download.

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 rule-based 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 [17] 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) [22] and the DARPA Agent Markup Language (DAML) Program [4]. Rules indeed are part of the announced mission of the W3C's Semantic Web Activity [18]. Jess [11], acronym for "Java

Expert System Shell", is a popular rule system that is free for academic use and whose source code is relatively easily available.

SweetJess is part of our (first author's) larger toolset system SWEET, acronym for "Semantic WEb Enabling Technology". SWEET also includes SweetRules [10], a system for RuleML inferencing and translation, and SweetDeal [8] [9] [15], an approach to rule-based contracting that builds upon SweetRules and RuleML 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 the Situated Courteous Logic Programs knowledge representation. The SCLP case of RuleML is expressively powerful. The Courteous feature/extension enables prioritized conflict handling and (a limited form of) classical negation. The Situated feature/extension enables 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 [23]; the IBM CommonRules rule engine, a forward SCLP system [5]; Knowledge Interchange Format (KIF) [12], an earlier version of the Common Logic [6] emerging industry standard for knowledge interchange in classical logic; and Smodels, a forward logic-program rule engine. SweetJess aims to complement and extend SweetRules by providing additional capabilities including translation to Jess.

The first new contribution of the SweetJess approach is a fundamental mapping. We give a new, implemented translation from a broad but restricted case of SCLP RuleML into Jess rules, and an inverse translation from a broad but further restricted case of Jess rules into SCLP RuleML. The translation is intended to preserve semantic equivalence. Semantic equivalence means that, for a given rulebase, the same conclusions are entailed, and the same side-effectful actions (triggered by conclusions) are performed when the rulebase is executed (i.e., when the rules are "run"). The translation often preserves semantic equivalence; in current work, we are developing formal guarantees for the equivalence, including necessary expressive restrictions in each direction.

The set of expressive restrictions for the translation from SCLP RuleML into Jess includes several that are imposed by the expressive limitations of Jess; notably, these include:

1. Datalog, i.e., no logical functions;
2. safeness/range-restrictedness of rule heads, i.e., every logical variable in the head must appear in (and thus be bound by) the rule body;
3. "all-bound sensors", i.e., sensor attached procedures require all their arguments to be bound (fully instantiated, i.e., ground) when the sensor procedure is invoked; and
4. safe negation, i.e., variables in negated body literals must appear in (and thus be bound by) positive body literals (NB: negation here means negation-as-failure; and
5. dynamically stratifiable negation, i.e., the LP's model under the Well-Founded Semantics [20] must not need to assign any literals the truth value *undefined*.

In addition, there are some other limitations of Jess with regard to negation-as-failure that we are investigating in current work. Finally, Jess also lacks some naming capabilities (especially, for facts and rulebases), as compared to (SCLP) RuleML; however, these are less fundamental expressively than the above restrictions.

The above restrictions (1.), (2.), (4.), and (5.) are often found in rule-based systems — especially ones that do forward-direction inferencing. Restriction (3.), all-bound sensors, is more unusual and is practically rather limiting — it restricts a sensor attached procedure when queried to answer with simply a boolean, as opposed to a set of bindings. Overall, these restrictions underline some dimensions of the powerful expressive generality of SCLP as a KR. We will discuss the translation's expressive restrictions in more detail later.

The second new contribution of the SweetJess approach builds upon the translation. We give a new, implemented architecture to perform (a broad case of) SCLP RuleML inferencing using the Jess rule engine. Our approach translates from SCLP RuleML rules into Jess rules, runs the Jess rule engine to generate conclusions (and actions), and then translates the concluded Jess facts back into SCLP RuleML.

Translating the Courteous feature of SCLP RuleML, i.e., its prioritized conflict handling and classical-negation aspects, is a particular hurdle, since Jess essentially lacks the ability to directly

express these aspects. (Jess does include a quite limited kind of inferencing-control-agenda prioritization — "salience" — which is a discouraged mechanism.) Our approach is able to surmount this hurdle however, 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), but rather that only employs negation-as-failure (NAF). The IBM CommonRules library provides a Courteous Compiler, for example.

The third new contribution of the SweetJess approach is to enable bi-directional implemented 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 [8]). In particular, as we discussed earlier, SweetRules already enables bi-directional translation from SCLP RuleML to: XSB; Smodels; IBM CommonRules; and KIF. The overall approach to such translation was first given by us in [8]. 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 [3], which in turn are closely related to event-condition-action (ECA) rule systems [19]. 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 [2], together with SQL-type relational database systems (RDB) [19]. The core of SQL RDB's — relational algebra and Datalog — is well-known theoretically to be very closely related to pure Prolog. Systems in this second group (sometimes called "derivational") primarily employ backward chaining (rather than forward), i.e., query-answering.

The fourth new contribution of the SweetJess approach is a bridging across heterogeneous families of CCI rule systems. Our translation and tool are each the first, to our knowledge, to enable inter-operability between a Prolog and any "production"/"reactive" rule system descended from OPS5 heritage. More generally, our translation may be the first to go for a broad expressive case between the two groups (production/reactive vs. Prolog/SQL derivational).

The fifth 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. We discussed the most important of these above.

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.

The prototype implementation of SweetJess is and publicly available free on the Web[4]. It is implemented in Java and makes use of tools for XML, RDF [16] and RuleML.

The remainder of this paper is organized as follows. Section 2 provides background: we review RuleML, Situated LP, and Courteous LP. In section 3, we review Jess Rules and begin the analysis and reformulation that underlies our translation mappings. In section 4, we describe SweetJess' architecture to perform (a broad case of) SCLP RuleML inferencing using the Jess rule engine. In section 5, we come to the heart of the matter: we describe how to translate rules from SCLP RuleML to Jess. In section 6, we describe how to translate back from Jess to SCLP RuleML. In section 7, we wind up with some discussion, including additional directions for future work.

[4] http://daml.umbc.edu/sweetjess

2 Background: RuleML, Situated LP, Courteous LP

2.1 RuleML and LP

RuleML [17] is the leading emerging industry standard for XML-based inter-operable rules, i.e., Semantic Web rules. It is being developed by a coalition which includes participants from several dozen institutions, both academic and industrial. We (first author) co-lead it. The specification of RuleML includes an XML markup syntax, together with a formal semantics based on the knowledge representation (KR) of Logic Programs (LP)[5]. In adddition, there already exist early draft specifications of an abstract syntax, an alternative RDF syntax, and an OWL syntax. RuleML is intended primarily to support inter-operability among currently commercially important kinds (CCI) of rule systems. These CCI rule systems fall primarily into four major families today: relational database management systems (SQL), Prolog, OPS5-heritage production rules, and Event-Condition-Action (ECA) systems. The RuleML language actually provides a range of options for the fundamental KR expressiveness to be used in inter-operating between a group of two or more rule systems. Formally, RuleML organizes these options into a set of "sub-languages". Each sub-language provides a different combination of expressive features / expressive restrictions, and has associated syntax. The set of sub-languages is thus hierarchical, in that a given sub-language may provide a superset of (i.e., subsume) the expressiveness of a particular other sub-language.

In the LP literature, the most studied LP expressive class is what we will call "Ordinary" LP (OLP). OLP extends Horn LP by permitting body literals to be negated; its form of negation is negation-as-failure (NAF). Courteous LP subsumes OLP. OLP is also sometimes called "normal logic programs" (or "general [sic] logic programs" as in [1]). OLP is roughly pure Prolog without built-ins, but not limited to backward direction of inferencing.

Currently defined RuleML sub-languages also include:

1. Datalog Horn LP (no negation, no logical functions)
2. Horn LP (adds logical functions (of non-zero arity); no negation; subsumes Datalog Horn)
3. Situated Courteous LP (adds procedural attachments for sensing/effecting, and prioritized conflict handling; subsumes OLP and Horn)

and several others that provide combinations of various additional syntactic or expressive features/restrictions.

Note that Ordinary LP and Horn LP are "*pure-belief*" KR's — i.e., they lack procedural attachments. Most KR theory, generally, treats pure-belief formalisms.

RuleML's sub-languages differ from the formulation of LP expressive classes in the KR literature primarily in that RuleML provides a standardized set of mechanisms for syntax and for Webizing the KR. Besides providing XML[6] syntactic schemas, an important aspect of these mechanisms is to provide additional capabilities for naming, notably to permit predicates, individuals, and logical functions ("constructors") to be URI's, and to permit rule names and rulebase names to be part of the KR language. These naming capabilities facilitate highly distributed knowledge bases and inferencing for rules and their use of ontologies and databases — in short, they largely equip rule KR for the Semantic Web. In particular, permitting a predicate name to be a URI that refers to an OWL class or property enables RuleML rules to be "on top of" OWL ontologies (e.g., see [9] for the first detailed application scenario that used this capability).

RuleML has been evolving since its first version XML DTD's (V0.7) were released in early 2001. The current version (V0.8) includes additional extensions in expressive and syntactic features, and a few moderate revisions in the syntax. Our SweetJess design and implementation effort was started while V0.8 was still in development, and thus used the same version of the SCLP RuleML XML DTD as the first (still current) version of SweetRules [10]. This DTD (known as "SCLP dtd-v13") differs in a few minor ways with RuleML V0.8. In current work, we are updating the implementation to achieve full compliance with the RuleML V0.8 SCLP DTD[7].

The current SweetJess translation mappings and implementation incorporates all the features of the RuleML V0.8 SCLP DTD with one important exception: the object-oriented argument collections feature ("roli's"), which provides additional syntactic convenience (rather than increasing

[5] see [1] for a helpful review of LP KR

[6] and early drafts of RDF and OWL

[7] http://www.ruleml.org/dtd/0.8/ruleml-sclp-monolith.dtd

fundamental expressive power from a logical KR standpoint). The collection of arguments in a logical atom is thus, for now, simply an ordered tuple (in the current SweetJess translation mappings and implementation). Such an atom when ground is called an "ordered fact" in Jess. In current work, we are designing an extension of the SweetJess translation mappings and implementation to support the object-oriented argument collections feature as well.

Next, we give examples of a Fact and a Rule in RuleML V0.8 syntax.

Example 1. **Premium Customer Fact, in RuleML** A RuleML fact ("fact" element in the XML syntax) expresses a ground atom. It is a kind of statement. E.g.:

```
/* Fact: ‘‘Allan is a premium customer.’’ */
        premiumCustomer(Allan)
```

In RuleML syntax:

```
<fact>
   <_head>
     <atom>
         <_opr>
            <rel>premiumCustomer</rel>
        </_opr>
        <ind>Allan</ind>
     </atom>
  </_head>
</fact>
```

Example 2. **Discounting Rule, in RuleML** A RuleML implication rule ("imp" element in the XML syntax) expresses an if-then rule (a.k.a. a "clause" in the LP KR literature). It is a kind of statement. An implication rule is a pure-belief rule that does not directly specify any procedural attachments for sensing or effecting. E.g.,

```
/* Rule: ‘‘If a customer is a premium customer then give him a 10% discount.’’
        if premiumCustomer(?customer)
        then giveDiscount(percent10, ?customer)
```

For ease of human-readability, in this paper we often give our LP and RuleML examples (e.g., the ones above) in the Prolog-like "SCLPfile" syntax of IBM CommonRules V3.0 [5], which maps straightforwardly to RuleML. ";" ends a rule statement. The prefix "?" indicates a logical variable. "/* ... */" encloses a comment. "//" prefixes a comment line. "< ... >" encloses a rule label (name). Rule labels identify rules for editing and prioritized conflict handling, for example to facilitate the modular modification of contract provisions.

In RuleML syntax:

```
<imp>
   <_head>
      <atom>
         <_opr>
            <rel>giveDiscount</rel>
         </_opr>
         <ind>percent10</ind>
         <var>Customer</var>
      </atom>
   </_head>
   <_body>
      <atom>
         <_opr>
            <rel>premiumCustomer</rel>
         </_opr>
         <var>customer</var>
      </atom>
   </_body>
</imp>
```

For sake of brevity, especially in human authoring and reading, the XML syntax for RuleML uses terse/abbreviated names for the elements (and attributes). "rel" means relation, i.e., predicate. "ind" means individual (object constant). "var" means logical variable. "body" and "head" mean the antecedent (a.k.a. "if" part) and the consequent (a.k.a. "then" part) of a rule, respectively. "opr" stands for relational operator.

Note that a fact is similar syntactically to an (implication) rule that lacks a body. Conceptually, an empty body is viewed as logically True, as is usual in LP and classical logic.

Terminology: From a LP KR viewpoint, a fact is just a special case of a rule. A RuleML rulebase consisting of implication and fact statements is thus often simply called a "rulebase" or "ruleset". More generally, however, the Situated and Courteous features of SCLP extend the LP KR with three additional kinds of statements: sensor, effector, and mutex; we will be discussing those in more detail later. A collection of such SCLP statements (all five kinds) constitutes a RuleML rulebase in the more general sense.

2.2 Situated Logic Programs

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 following example.

Example 3. **Order Cancellation with Notification Action**

```
   /* rule: should 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 statement: associated with the predicate shouldInformCustomer, the ack method
performs a notification action */
       Effector: shouldInformCustomer /* the predicate */
       Class: orderMgmt.Request.mods
       Method: ack
       Path: ‘‘edu.cs.umbc.SLP.examples.orderMgmt.aprocs’’;
   /* sensor statement: associated with the predicate receivedBefore, the earlierReceiptDate
method queries an external order management system */
       Sensor: receivedBefore /* the predicate */
       Class: orderMgmt.Request
       Method: earlierReceiptDate
       Path: ‘‘edu.cs.umbc.SLP.examples.orderMgmt.aprocs’’;
       BindingRequirement: (BOUND,BOUND)
```

The effector statements in a SLP each associate a pure-belief predicate, e.g., shouldInformCustomer, with an external attached procedure, e.g., orderMgmt.request.mods.ack (here, a Java method). During rule inferencing (more precisely, during rule execution), when a conclusion is drawn about the predicate, e.g.,"shouldInformCustomer(cancelRequest4216, accepted)" if the rule above was fired successfully, then the external procedure is invoked as a side-effectful action, e.g., the method "ack" is called with its parameters instantiated to "(request1049, accepted)". "External" here means external to the inferencing engine itself. An (external) attached procedure is also called an "*aproc*" for short.

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

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 above, the sensor aproc 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 provided by a particular individual user/application.

Terminology: a predicate which has one or more associated sensor (effector) statements is called a "sensor predicate" ("effector predicate"). A literal in a sensor (effector) predicate is called a "sensor literal" ("effector literal"). Overall, we call the process of drawing conclusions and performing related effector and sensor invocations in SLP: "*situated inferencing*".

Sensing about a given predicate p occurs in addition to any facts derivable from rules (or facts) whose heads are p atoms. Also, a given predicate p may appear in multiple sensor statements, i.e., p may have multiple sensor aproc's. The results from querying all of these sensor aproc's are combined (i.e., union'd) when a rules body's sensor literal in p is tested.

Likewise, a given predicate p may appear in multiple effector statements, i.e., p may have multiple effector aproc's. When a conclusion is drawn about p, each of these effector aproc's is invoked.

2.3 Courteous Logic Programs: Review

Courteous Logic Programs (CLP) is an expressive super-class of Ordinary Logic Programs (OLP). The Courteous expressive extension enables prioritized conflict handling and also a limited form of classical negation. Next, we give an example of a CLP, having 2 rules, 1 fact, and 1 mutex.

Example 4. **Prioritized Discounting Rules**

```
/* 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 give him a 0% (NO) Discount */
 <slowPayer>
      IF slowToPay(?Cust, last1year)
      THEN giveDiscount(percent0, ?Cust);
  /* prioritization fact: SlowPayer is higher priority than SteadySpender */
      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) ;
```

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., an individual 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 "*mutex*" statements. A mutex specifies a (conditional) pair-wise mutual exclusion between two literals; these are called the "*opposer*" literals of the mutex. The mutex statement also includes a condition, called its "*given*" part. The mutex given part is similar to the body of a rule; it may be empty (i.e., $True$). E.g., the mutex above specifies that it is a contradiction to conclude two different values of the percentage discount for the same customer; i.e., `giveDiscount` is a (partial-)functional predicate. The semantics of Courteous LP enforces consistency of the conclusions wrt each mutex.

Any literal may be classically negated; however, (C)LP as a KR only supports a quite limited form of classical negation (a.k.a. "strong" negation). There is an *implicit* mutex between p and classical-negation-of-p, for each p, where p is a predicate, ground atom, or atom. These implicit mutex's are called "*classical-negation*" mutex's.

The semantics of Courteous LP ensures overall consistency of the conclusion set, including consistency between the two concepts of negation (classical negation of p entails NAF of p, but not vice versa).

Combining Courteous + Situated → SCLP: The Courteous expressive extension can be combined with the Situated expressive extension, to form Situated Courteous LP. Currently in the theory of Situated Courteous LP, however, there is an *expressive restriction* on this combination: the sensor predicates must be conflict-free. More precisely, the restriction on the SCLP rulebase is that: no mutex opposer literal may be a sensor literal. This includes the implicit classical-negation mutex's, thus no sensor literal may be classically-negated. We call this restriction "*conflict-free sensing*" or "*monotonic sensor predicates*", for short. In current work, we are generalizing the theory of SCLP to remove this restriction.

3 Jess Rules: Overview, Analysis, and Reformulation

3.1 Jess Facts

Conceptually, what Jess calls a "fact" is very similar to the concept of a fact in RuleML; it expresses a ground atom. Jess provides some machinery for defining facts and acquiring them from surrounding Java context, with which we will not need to concern ourselves in this paper. We will concern ourselves simply with the basic concept of a Jess fact — specifically, what Jess calls an "ordered" fact, i.e., one where the fact's arguments constitute an ordered tuple. Jess also provides as a syntactic enhancement the concept of an "unordered" fact, which uses slot names (rather than sequence) to define arguments within a fact (or atomic pattern); this is quite similar to the object-oriented argument collections feature of RuleML V0.8, which we mentioned earlier. The current SweetJess translation mappings and implementation just deal with Jess "ordered" facts, however.

Jess uses a Lisp-like syntax, generally. A Jess fact statement has the following kind of form (we can view this roughly as a generic template):

```
(assert (predicateName constant1 constant2 ...constantN) )
```

The statement begins with the "`assert`" keyword. This is followed by an expression syntactically similar to an LP ground atom: a predicate followed by an ordered collection of individual constants. Different predicates have different arities; we indicated this in the form above by writing "N" as the arity. Actually, in Jess, "`assert`" is a system procedure (what Jess calls a "function"); when executed it puts the fact into the currently active stored knowledge base of the Jess inferencing engine. Note that, in the tradition of OPS5-heritage production rule systems, Jess does not conceptually view a Jess fact as a special case of a Jess rule. However, in our translation mapping, we will essentially view a Jess fact as a special case of a *LP* rule, from the viewpoint of KR theory.

Jess lacks the semantic equivalent of logical functions, i.e., constructors, that have non-zero arity. It thus also lacks the semantic equivalent of complex terms formed using constructors. This is known in LP and classical logic KR as the *Datalog* restriction. The closest thing in Jess to a non-zero-arity constructor is a JessMethod procedure (see next sub-section), but that is always evaluated on its arguments; the semantics of a non-zero-arity constructor, however, essentially correspond to not evaluating it.

3.2 Jess Rules

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

```
(defrule ruleName
(predicate1 constant1 ?boundVariable1)
(test (jMethod1 constant2 ?boundVariable1))
=>
(jMethod2 (symbol3 ?boundVariable1)) )
```

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. The LHS of a Jess rule is a "*pattern*" in Jess terminology. A *basic* "pattern" matches facts and corresponds to an atom in

a LP rule body. E.g., the second line in the example form above is a basic pattern. This atom may include logical variables, not just individual constants, as arguments. More complex patterns can be formed in several ways. The LHS may consist of a (top-level) list of basic patterns; these are interpreted (implicitly) as a conjunction.

The basic form of a Jess rule's RHS is an "*action*" in Jess terminology. An action is simply a call to one of Jess' Java procedures. We will call these procedures "JessMethods". (Jess calls them "functions", but we wish to reserve "function" for logical functions cf. LP and classical logic KR terminology.) Syntactically, a JessMethod call is simply a Lisp-like list whose first member is the name of a JessMethod, followed by arguments. Each such argument may be a constant, logical variable, or another JessMethod procedure call expression. E.g., the last line in the template example above is an action. "Jess comes with a large number of built-in [JessMethods] that do everything from math, program control and string manipulations, to giving you access to Java APIs.", says the User Manual[8]. In addition, a user may define their own additional JessMethods, which are essentially general Java procedures (methods); in practice, this capability is often used heavily. In general, an action may essentially arbitrarily modify program state, and is a way to do pretty much anything you can do in Java.

From the standpoint of KR and for designing our translation mappings, however, one JessMethod is particularly germane: "`assert`". A common kind of action in Jess rules is to `assert` a fact. In this case, the Jess rule RHS corresponds to the head atom of a (pure-belief) implication rule in LP KR, and thus in RuleML. When the Jess rule inferencing engine runs, if the rule LHS is satisfied ("matches" in Jess terminology), i.e., if the rule fires (with variable bindings supplied as in the usual manner for rules), then this RHS fact is added to the working fact set portion of Jess' knowledge base. From the standpoint of LP KR, this corresponds to drawing a conclusion. More generally, the RHS may consist of a (top-level) list of actions. If these are `assert`'s, the list is interpreted (implicitly) as a conjunction of its member actions.

The result of the RHS when a rule fires thus may be either (1) to draw a conclusion fact (or several facts), or (2) to perform some (general) procedural action(s) that may be — and typically, are – side-effectful. From the standpoint of Situated LP KR, expressively, case (2) corresponds essentially to generating an effector call to a JessMethod. (A complex RHS can be rewritten as a call to a single JessMethod.)

In addition to basic patterns that correspond to LP atoms, there is one other fundamental kind of (pattern) expression can appear in the LHS of a Jess rule: a "TEST Conditional Element" in Jess terminology, called "TestCE" for short.[9] [10] A TestCE is constructed syntactically using the reserved keyword "`test`" followed by a JessMethod call expression (as in the third line of the template example above). When the rule runs, the TestCE tests whether that JessMethod call expression (when supplied with variable bindings by matching the rest of the LHS) evaluates to True (vs. False). From the standpoint of Situated LP KR, this essentially corresponds to a sensor call. When the JessMethod call is invoked, all the arguments of the JessMethod must be fully bound. This is significantly less (expressively) general than the concept of a sensor call in Situated LP, where some or all of the arguments may be (or contain) free variables. We call this the *all-bound sensors* expressive restriction.

Jess also has explicit logical connectives. The first is "`not`", which is negation-as-failure. "`(not` BP`)`, where BP is a basic pattern, is similar to a negation-as-failure literal in LP. More complex patterns can be built up by explicit use of logical connectives; these include `not`, `and`, `or`, `exists`, and can be nested. This provides enhanced expressiveness in a direction very similar to (but somewhat less general than) "*Lloyd-Topor*" LP. "Lloyd-Topor" LP is the extension of OLP to use the Lloyd-Topor transformation [13]. The Lloyd-Topor transformation does not increase the fundamental expressiveness of OLP, however; it reduces its more expressive version of LP into OLP. In that sense, it can be viewed as syntactic sugar.

[8] Jess V6.1, section 2.2

[9] In addition, Jess permits "predicate constraints" and "return value constraints" to be associated with LHS variables, but these do not provide any extra essential expressiveness; they are reducible to TestCE's plus the other permitted LHS pattern constructs.

[10] Note that a LHS rule pattern may not contain a top-level JessMethod appearance, only `test`.

Jess requires that all logical variables appearing in the RHS be bound by matching in the LHS. I.e., all RHS variables must appear in the LHS. This is known in LP and classical logic KR as *range-restrictedness* and as the *safe-head* expressive restriction. Jess also requires that: all logical variables appearing in a basic pattern (or more complex expression) that is negated (by "`not`"), must be bound by matching (on non-negated patterns) in the rest of the LHS. [11] This is known in LP and classical logic KR as the *safe negation* expressive restriction. Safe-head and safe-negation are frequent restrictions in practical rule-based inferencing systems, and are especially common in forward-direction inferencing systems.

Jess Rule Engine: Above, we have described a Jess rule and a Jess fact, taken one a time. Overall, the KR semantics of a set of Jess rules and facts must be related to what the Jess engine does — what conclusions are drawn and what procedural actions are performed — not just to what is the conceptual intention of a rule or fact. From a KR viewpoint, the Jess engine is similar to the engines of several other OPS5-heritage production rule systems. It uses the Rete (Latin for "net") algorithm [7] for efficiency in "pattern" matching, especially to handle updates to its working set of facts.

In current work, we are developing more formal theory to characterize the production-rule Rete engine algorithm in terms of the LP KR.

4 SweetJess' Overall Architecture for Inferencing and Translation

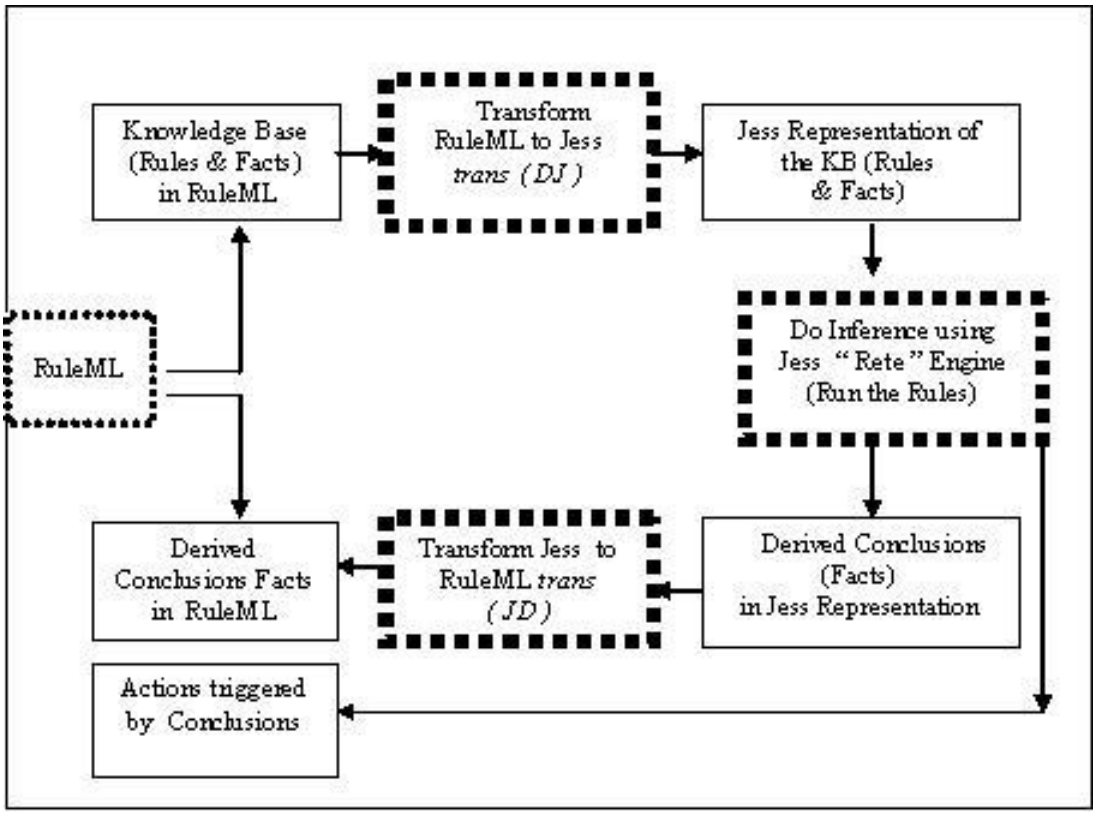

Fig. 1. SweetJess Architecture for RuleML Inferencing via Jess

The bi-directional translation between RuleML and Jess has several potential uses, as discussed in section 1. By way of motivation for the details in the rest of this paper, however, we largely

[11] Jess also allows local existentially-quantified variables within a negated expression, but this is expressively inessential.

focus on one particular use: to perform RuleML inferencing via Jess. Figure 1 shows SweetJess' architecture for this. By "RuleML inferencing", we mean (situated) inferencing from a premise RuleML rulebase ("Knowledge Base of rules and facts" in the Figure) to derive RuleML conclusions and related procedural actions that are triggered from those conclusions via procedural attachments ("effectors" in Situated LP, described in section 2.2). By "via Jess", we mean that Jess is used as an engine to "run" the rules.

To perform such inferencing, SweetJess first translates a premise RuleML rulebase into a set of Jess rules and facts (and JessMethod definitions). This transformation is called trans[RJ]. Inferencing is then performed using Jess' rule inference engine, 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 RuleML facts. This inverse direction transformation is called trans[JR]. The result is a set of RuleML conclusion facts entailed by the original RuleML premise rulebase.

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 RuleML rulebase. The transformations trans[RJ] and trans[JR] impose some expressive restrictions on their input, which we will describe later. For the results of inferencing, only facts need be translated via trans[JR] from Jess to RuleML. Our trans[JR] translation, more generally, actually handles (as input) Jess rules too, not just Jess facts. The current implementation of trans[JR], however, restricts these input Jess rules to be pure-belief rules, i.e., without JessMethods (other than `assert`), and to be in similar syntactic form to what trans[RJ] produces. In current work, we have been extending the design of trans[JR] to relax these restrictions.

5 Transforming RuleML To Jess: trans[RJ]

There are several major challenging aspects of designing the translation mappings between RuleML and Jess. One aspect is to map conceptually between the terminologies and, more deeply, the concepts of the two different rule languages. We described much of that conceptual mapping earlier in sections 3 and 4. A second aspect is to handle procedural attachments for tests/sensing and actions/effecting. A third aspect is to handle prioritized conflict handling. A fourth aspect is to identify the appropriate expressive restrictions in doing all of the above.

Next, we describe in more detail the various aspects of trans[RJ]. The topics of the following subsections are sequenced roughly in the sequence of increasing expressiveness: from facts to Horn LP to OLP to Situated OLP to Situated Courteous LP. As we go, we sometimes need to distinguish the translation mapping (which is at the level of design) from the current implementation of SweetJess.

5.1 Overall Input and Output

The transformation trans[RJ] takes as input a RuleML rulebase — an XML document which we will call a .rml file. 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. The batch file begins with the Jess system command "`(reset)`", which when executed clears out the knowledge base and resets the engine. The batch file ends with the Jess system command "`(run)`", which when executed starts up inferencing. The current version of Jess is V6.1. The current version of RuleML is V0.8. As we discussed earlier in section 1, the current SweetJess implementation still uses a slightly earlier version of the XML DTD for SCLP RuleML.

5.2 Fact

A basic RuleML fact — an atom whose arguments are an ordered tuple of individuals and/or variables — translates straightforwardly into a Jess fact.

Example 5. **Premium Customer Fact, in Jess:** The equivalent Jess fact corresponding to the RuleML fact in Example 1 is as follows:

```
(assert (premiumCustomer Allan) )
```

URI constant names feature: An important Webizing feature of RuleML is "URI constant names": predicate or individual (or other) constant names can be be URI's. A very basic way to translate these in trans[RJ] is simply to treat them as strings; however, then they are not recognized as being URI's by Jess. Translating them round-trip from RuleML to Jess back to RuleML in such a way as to preserve the ability to recognize URI-ness upon the return to RuleML, is relatively straightforward to support in trans[RJ] and trans[JR], e.g., via a prefixing/encoding convention. Our translation mapping design thus includes URI constant names, but the current implementation does not yet support this feature.

Rule naming issues: In RuleML, 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 RuleML fact's rule label, if present, is thus lost by the transformation. Thus formally, there is an expressive restriction on the transformation: "*no fact labels*". (An alternative design to relax this restriction would be to translate a RuleML fact into a Jess rule that has a trivially true body, then the rule label could be preserved. But this would probably have some other disadvantages, e.g., less efficiency.)

Object-oriented argument collections: The current design of the translation mapping also does not yet support the object-oriented argument collections feature of RuleML V0.8. That recently-added feature, as yet experimental in status, provides syntactic and conceptual convenience but does not add fundamental expressiveness from a KR standpoint. We will call this restriction the "*tuple-arguments*" expressive restriction.

Expressive restrictions imposed on the rulebase: Datalog, no fact labels, tuple-arguments.

Expressive features enabled for rulebase: URI constant names.

5.3 Horn Rule

A RuleML Horn LP implication rule (i.e., `imp` statement) is translated into a Jess rule whose LHS corresponds to the RuleML body, and whose RHS corresponds to the RuleML head. Each RuleML body atom is translated into a corresponding basic pattern. The RuleML head atom is translated into an assert of the corresponding basic pattern. The RuleML rule label, if present, is translated into the Jess rule name, else a new rule name is automatically generated.

Example 6. **Discount Rule, in Jess** After transforming the RuleML rule given in Example 2 (sub-section 2.1), the corresponding Jess rule is:

```
(defrule discountRule
   (premiumCustomer ?customer)
   =>
   (assert (giveDiscount percent10 ?customer) ) )
```

A further subtlety arises when translating rulebases that contain sensor statements. The sensor statements modify how a rule mentioning a sensor predicate is translated so as also to generate TestCE patterns in place of basic patterns; see sub-section 5.6 below for details.

Naming Issues: A RuleML rule label (name) is optional and need not be unique. Jess requires a name for every rule, w hich 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 RuleML rules' labels not to coincide with each other. We call this the *unique rule labels* expressive restriction.

Expressive restrictions imposed on the rulebase: unique rule labels, safe head. Note also that the Datalog restriction applies to the LP rule labels, thus each rule label must be simply an individual constant.

Expressive features enabled for the rulebase: Horn LP.

5.4 Lloyd-Topor And-Or (LTAO) Expressiveness

The "*Lloyd-Topor And-Or (LTAO)*" permits: (1) conjunction (of literals) to appear in a rule head; and (2) disjunction to appear in a rule body, even nested with conjunction to form AND-OR expressions formed from literals. This is a portion of the Lloyd-Topor transformation: a rule

$a \wedge b \leftarrow c$ can be rewritten as two rules $a \leftarrow c$ and $b \leftarrow c$; and a rule $a \leftarrow (b \vee c)$ can be rewritten as two rules $a \leftarrow b$ and $a \leftarrow c$.

trans[RJ] supports the LTAO expressive feature. LTAO is convenient but expressively inessential. The SCLP RuleML V0.8 sub-language includes LTAO; many practical rule-based systems support it. LTAO can be added (inessentially) to Horn LP or any of its expressive super-classes (e.g., OLP).

An advantage of this LTAO feature is conciseness and naturalness in authoring of rules. The LTAO feature is straightforward to implement.

Expressive features enabled for the rulebase: Lloyd-Topor And-Or (LTAO).

5.5 Negation-as-Failure (NAF)

Negation-as-failure (NAF) in OLP and SCLP RuleML is translated into Jess negation ("`not`"), which also is NAF.

NAF in OLP (and thus in SCLP), and in rule systems generally, is somewhat tricky in the fully general case, both to define semantically and to implement. As is well-known in the OLP literature, NAF can cause semantic trouble by interacting with cyclic dependencies ("recursion") among rules. A full discussion of the subtleties of NAF is beyond the scope/space of this paper. But we do impose the following restriction: *dynamically stratifiable negation*, i.e., the (O)LP's model under the Well-Founded Semantics [20] must not need to assign any literals the truth value *undefined*. (Note that NAF-free is a special case of dynamically stratifiable.)

In addition, there are some other limitations of Jess with regard to negation-as-failure that we are investigating in current work.

Expressive restrictions imposed on the rulebase: safe negation, dynamically stratifiable negation.

Expressive features enabled for the rulebase: Negation-As-Failure (NAF), thus OLP.

5.6 Situated LP Procedural Attachments for Sensing and Effecting

Next, we describe how sensor and effector statements, in the Situated feature of RuleML, are translated.

An effector statement associating a predicate p with an attached procedure q, is translated into two Jess statements. (1) The first is a Jess rule whose LHS is an open atom in the predicate p and whose RHS is a JessMethod that invokes the aproc q. Here, "Open" atom means that all of the atom's arguments are (free) logical variables. (2) The second is a JessMethod definition statement (using the `deffunction` keyword/command), that defines a a new "user-defined" JessMethod that corresponds to the effector aproc q. There are some other relatively straightforward mechanics of how to pass the path, classname, and methodname of a (sensor or effector) aproc to Jess when defining the corresponding JessMethod; some of these details are in evidence in the example below. There are a few different possible lower-level design choices for exactly how to do this. Below, we illustrate and describe how it is done in the current implementation.

Example 7. **Notification Effector**: The effector statement from Example 3 (sub-section 2.2) is translated into the following Jess rule. (Also — not shown here — there is a JessMethod definition statement for the effector aproc `ack`.)

```
(defrule effect_shouldInformCustomer_1
    (shouldInformCustomer ?orderModificationRequest ?status)
    =>
    (effector ack orderMgmt.Request.mods
                (create$ ?orderModificationRequest ?status) ) )
```

The "`effector`" JessMethod in the Jess rule above is actually a *generic* effector procedure (one for the whole rulebase) that takes the particular effector aproc's methodname, classname, and effector-call argument list as its input parameters. This is a rather elegant low-level design approach that exploits the power of Java in Jess. This generic effector procedure also has a corresponding JessMethod definition statement. Similarly, in this approach, there is a generic "`sensor`" JessMethod (one for the whole rulebase).

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 sensor atom of the form $p(t)$, within the body of r, is then translated into the expression

`(or` $BP(p,t)$ `(test` $MC(q,t)$`) )`

rather than into simply the basic pattern $BP(p,t)$ which corresponds directly to the sensor atom. Here, t stands for a tuple of arguments, and $MC(q,t)$ is a JessMethod call expression to invoke aproc q on arguments t. That is, a TestCE pattern is also generated to do sensing via the aproc q, and that TestCE pattern is disjoined (i.e., OR'd) with the basic pattern in p. The result when the rule LHS is matched/tested is to invoke/query the sensor as well as to match the basic pattern against the working memory's set of facts. Also, like with effectors, the translation generates a JessMethod definition statement for the sensor aproc q.

More generally, there may be multiple (e.g., two) sensor statements for p, each with a corresponding aproc, e.g., $q1$ and $q2$. In this case, multiple TestCE patterns are disjoined — one per aproc — e.g., the sensor atom $p(t)$ is translated into

`(or` $BP(p,t)$ `(test` $MC(q1,t)$`) (test` $MC(q2,t)$`) )`

Example 8. **Receipt Date Sensor**: In Example 3, the rule together with the sensor statement, is translated into the following Jess rule.

```
(defrule rule_526
    (deadlineToCancel order4215 ?Day)
    (or
       (receivedBefore cancelRequest4216 ?Day)
       (test (sensor earlierReceiptDate orderMgmt.Request
                     (create$ cancelRequest4216 ?Day) ) ) )
    =>
    (assert (shouldInformCustomer cancelRequest4216 accepted) ) )
```

(Also — not shown here — there is a JessMethod definition statement for the sensor aproc `earlierReceiptDate`.)

Expressive restrictions imposed on the rulebase: all-bound sensors (recall section 3).

Expressive features enabled for the rulebase: Situated LP (sensors, effectors), thus Situated OLP.

5.7 Courteous Prioritized Conflict Handling

So far, we have described how to translate for the Situated Ordinary LP (SOLP) expressive class of RuleML (along with the LTAO and URI constant names features). We call this the *SOLP case* of the translation. SCLP RuleML also includes the Courteous expressive feature for prioritized conflict handling (and limited classical negation), which we overviewed in sub-section 2.3. Jess, however, does not support anything like the Courteous feature directly; it lacks the ability (directly) to express mutex's or the kind of prioritized conflict handling that Courteous LP enables.

Use Courteous Compiler: We have found a means for overcoming this incapacity of Jess. It is to compose an additional transformation, called the Courteous Compiler, as a first step before the "basic" SOLP-case translation. As we have shown in previous work [8] [5], the Courteous Compiler transforms a (Situated) Courteous LP into a semantically equivalent (Situated) Ordinary LP. IBM CommonRules and SweetRules make use of a Courteous Compiler component, for example; CommonRules provides one as part of its toolset. [12] For an input SCLP RuleML rulebase that contains Courteous expressive features (notably, mutex's explicit or implicit), we thus refine the SweetJess architecture accordingly: as a first step in trans[RJ], the input RuleML rulebase (.rml) is transformed via a Courteous Compiler (CC) component into a different, but semantically equivalent, RuleML rulebase that no longer contains the Courteous features. This post-CC rulebase is then run through the basic SOLP-case translator for trans[RJ]. The SOLP-case translator handles SOLP (plus LTAO and URI constant names).

[12] The Courteous Compiler step is tractable computationally: worst-case $O(n^3)$ but typically more like $O(k * n)$, where $3 \leq k \leq 50$, in practical experience to date.

Expressive features enabled for the rulebase: Courteous (prioritized conflict handling, limited classical negation), thus SCLP.

Expressive restrictions imposed on the rulebase:
(1) the *post-Courteous-Compiler NAF-related* restrictions (dynamic stratifiability)[13]; and
(2) *conflict-free sensing* (recall sub-section 2.3).
Note also that the Datalog restriction applies to the mutex's.

6 Transforming Jess To RuleML: trans[JR]

Next, we describe trans[JR], the translation from Jess to RuleML. In the rest of this section, the "translation" means trans[JR], unless explicitly indicated otherwise.

The input to trans[JR] 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 trans[JR] is a RuleML (.rml) file.

To support RuleML inferencing via translation to/from Jess, via our SweetJess architecture, it suffices simply to translate Jess facts — i.e., the conclusions of inferencing by the Jess rule engine — to RuleML.

The translation of facts is straightforward. Jess facts are defined in calls to the JessMethod "`assert`". To transform a Jess fact, trans[JR] just strips off the `assert` to obtain the inner ground-atom-like expression, and generates a RuleML fact that corresponds to that inner expression. Facts in Jess each have a unique Fact Id which is generated by the system upon loading or inferencing. Jess facts do not have an explicit label for identification. In RuleML facts have an optional rule label. For the time being, we define trans[JR] to simply translate this Jess Fact Id into the RuleML fact label.

Our translation mapping extends to much more than facts, however. It is relatively straightforward to invert the OLP case of trans[RJ], i.e., to "round-trip" the results of trans[RJ]; each Jess rule is translated into a RuleML (implication) rule.

The fundamental expressive class covered by trans[JR] thus includes OLP (with the other OLP-relevant expressive features and restrictions that we discussed in the last section). The current prototype implements this case of trans[JR]. However, in the current implementation of trans[JR], the Situated extension is not supported, i.e., the implementation of trans[JR] does not handle TestCE's or (non-`assert`) actions.

In the larger SweetJess effort, we have been investigating how to extend the fundamental expressive class covered by this translation trans[JR] from OLP to *Situated* OLP. Next, we give a sketch of how.

(1.) For each JessMethod *sproc* appearing in some TestCE:
(a.) introduce a new predicate *spred*; and
(b.) generate a sensor statement associating *spred* with *sproc*.
Note this sensor statement also essentially defines/declares *sproc* from RuleML's viewpoint.

(2.) Likewise, for each JessMethod *eproc* appearing in some action (expression):
(a.) introduce a new predicate *epred*; and
(b.) generate an effector statement associating *epred* with *eproc*.
(Similarly, this effector statement also essentially defines/declares *eproc* from RuleML's viewpoint.)

(3.) Translate a TestCE pattern (*sproc t*) to a sensor atom *spred*(t), where t stands for the arguments.

(4.) Translate an action (*eproc t*) to an effector atom *epred*(t), where t stands for the arguments.

7 Conclusions, Discussion and Future Work

For the main Conclusions, see section 1 "Introduction and Overview", especially the list of novel contributions we gave there. At core, our effort is not particular to RuleML or Jess, but rather

[13] In current work, we are investigating how to characterize syntactic sufficient conditions on the original (S)CLP to guarantee this restriction is met.

between knowledge representations. Its essence is to translate from declarative SCLP to (OPS5-heritage) production rules, and vice versa. This continues the overall approach and vision we first gave in [8].

That the translation between RuleML 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 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 Jess 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.

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; extending to object-oriented argument collections; 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.

References

1. Baral C. and Gelfond M.,"Logic Programming and Knowledge Representation", J. Logic Programming, 19-20: 73-148
2. Clocksin W.F. and Mellish C.S., Programming in Prolog. Springer-Verlag, 1981
3. T. Cooper and N. Wogrin, Rule-Based Programming with OPS5. Morgan-Kaufmann Pub., 1988
4. DARPA Agent Markup Language Program http://www.daml.org/
5. IBM CommonRules. http://www.alphaworks.ibm.com/
6. Common Logic, a proposed ISO standard. http://cl.tamu.edu/
7. Forgy, Charles L., "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem", *Artificial Intelligence* 19(1), pp. 17-37, 1982.
8. 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.
9. Grosof, B.N., Poon, T.C., "Representing Agent Contracts with Exceptions using XML Rules, Ontologies, and Process Descriptions". Proc. 12th Intl. Conf. on the World Wide Web (WWW-2003), 2003. Earlier version in: Proc. Intl. Wksh. on Rule Markup Languages for Business Rules on the Semantic Web, held at 1st Intl. Semantic Web Conf., 2002.
10. 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. Extended report version available at author's website.
11. Jess. http://herzberg.ca.sandia.gov/jess/.
12. Knowledge Interchange Format. http://logic.stanford.edu/kif and http://www.cs.umbc.edu/kif. Closely related is the new Common Logic effort.
13. John W. Lloyd, *Foundations of Logic Programming, Second, Extended edition*, Springer, Berlin, 1987.
14. Niemela, I. and Simons, P., Smodels (version 1). http://saturn.hut.fi/html/staff/ilkka.html.
15. Reeves D.M., Wellman M.P. and Grosof B.N., "Automated Negotiation From Declarative Contract Descriptions". *Computational Intelligence*, special issue on Agent Technology for Electronic Commerce, Nov. 2002. (Revised and extended from 2001 Autonomous Agents conference paper.)
16. Resource Description Format (RDF) from World Wide Web Consortium. http://www.w3.org.
17. Rule Markup Language Initiative. http://www.ruleml.org and http://www.ebusiness.mit.edu/bgrosof/#RuleML.
18. Semantic Web Activity of the World Wide Web Consortium. http://www.w3.org/2001/sw.
19. Ullman J.D. and Widom J., A First Course in Database Systems. Prentice-Hall, 1997.

20. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf, "The Well-Founded Semantics for General Logic Programs," *Journal of the ACM*, 38:3, pp. 620-650, July 1991. http://www.cs.columbia.edu/~kar/pubsk/wfjacm.ps.
21. Web Services Activity of the World Wide Web Consortium. http://www.w3.org/2002/ws.
22. World Wide Web Consortium. http://www.w3.org.
23. XSB logic programming system. http://xsb.sourceforge.net./ and http://www.sunysb.edu/ sbprolog.
24. XSLT (eXtensible Stylesheet Language Transformations), http://www.w3.org/Style/XSL/.