Representing Agent Contracts with Exceptions using XML Rules, Ontologies, and Process Descriptions

Benjamin N. Grosof [1] and Terrence C. Poon [2]

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

[2] MIT Electrical Engineering and Computer Science Dept.
tpoon@alum.mit.edu

Abstract. SweetDeal is a rule-based approach to representation of business contracts that enables software agents to create, evaluate, negotiate, and execute contracts with substantial automation and modularity. It builds upon the situated courteous logic programs knowledge representation in RuleML, the emerging standard for Semantic Web XML rules. Here, we newly extend the SweetDeal approach by also incorporating process knowledge descriptions whose ontologies are represented in DAML+OIL, thereby enabling more complex contracts with behavioral provisions, especially for handling exception conditions (e.g., late delivery or non-payment) that might arise during the execution of the contract. This provides a foundation for representing and automating deals about services – in particular, about Web Services, so as to help search, select, and compose them. Our system is also the first to combine emerging Semantic Web standards for knowledge representation of rules (RuleML) with ontologies (DAML+OIL) for a practical e-business application domain, and further to do so with process knowledge. A prototype is running. We intend to make the prototype publicly available.

1. Introduction

A key challenge in e-commerce is to specify the terms of the deal between buyers and sellers, e.g., pricing and description of goods/services. In previous work [1] [2], we have developed an approach that automates such business contracts by representing and communicating them as modular logic-program rules. That approach, now called SweetDeal, builds upon our situated courteous logic programs (SCLP) knowledge representation in RuleML [3], the emerging standard for Semantic Web XML rules that we (first author) co-lead. SweetDeal also builds upon our SweetRules prototype system for rules inferencing and inter-operability in SCLP RuleML [4].

In this paper, we newly extend the SweetDeal approach by also incorporating process knowledge descriptions whose ontologies are represented in DAML+OIL [5], thereby enabling more complex contracts with behavioral provisions, especially for handling exception conditions that might arise during the execution of the contract.

For example, a contract can first identify possible exceptions like late delivery or non-payment. Next, it can specify handlers to find or fix these exceptions, such as contingency payments, escrow services, prerequisite-violation detectors, and notifications. Our rule-based representation enables software agents in an electronic marketplace to create, evaluate, negotiate, and execute such complex contracts with substantial automation, and to reuse the same (declarative) knowledge for multiple purposes. In particular, our approach provides a foundation for representing and automating *deals about services* – including about electronic services, e.g., Web Services – so as to help search, select, and compose them. It thereby points the way to how and why to combine Semantic Web techniques [6] with Web Services techniques [7], a topic which the DAML-Services effort [8] has also been addressing (although not yet much in terms of describing contractual deal aspects).

Our SweetDeal system is also the first to combine emerging Semantic Web standards for knowledge representation of rules (RuleML) with ontologies (DAML+OIL) knowledge for a practical e-business application domain, and further to do so with process knowledge. The process knowledge ontology (e.g., about exceptions and handlers) is drawn from the MIT Process Handbook [9], a previously-existing repository unique in its large content and frequent use by industry business process designers. This is the first time that the MIT Process Handbook has been automated using XML or powerful logical knowledge representation.

This paper is drawn from a larger effort on SweetDeal whose most recent portion (second author's masters thesis) defines and implements a software agent that creates contract proposals in a semi-automated manner by combining modular contract provisions from a queryable contract repository with process knowledge from a queryable process repository. A prototype of the SweetDeal system is running, and further development of it is in progress. We intend to make the prototype publicly available in the near future.

2. SweetRules, RuleML, SweetDeal: More Background

SweetDeal is part of our larger effort SWEET, acronym for "Semantic WEb Enabling Technology", and is prototyped on top of SweetRules. Our earlier 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. SweetRules enables bi-directional translation from SCLP RuleML to: XSB, a Prolog rule system [10]; Smodels, a forward logic-program rule engine [11]; 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] The latest component of SweetRules is SweetJess [14], currently being prototyped, which aims to enable bi-directional translation to Jess, a popular open-source forward production-rule system in Java [15].

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

The SCLP case of RuleML is expressively powerful. The courteous extension of logic programs enables prioritized conflict handling and thereby facilitates modularity in specification, modification, merging, and updating. The situated extension of logic programs enables procedural attachments for "sensing" (testing rule antecedents) and "effecting" (performing actions triggered by conclusions). Merging and modification is important specifically for automated ("agent") contracts, because contracts are often assembled from reusable provisions, from multiple organizational sources, and then tweaked. Updating is important because a contract is often treated as a template to be filled in. For example, before an on-line auction is held a contract template is provided for the good/service being auctioned. Then when the auction closes, the template is filled in with the winning price and the winner's name, address, and payment method. Indeed, in [2] we show how to use SCLP to represent contracts in this dynamically updated manner, for a real auction server – U. Michigan's AuctionBot – and the semi-realistic domain of a Trading Agent Competition about travel packages. More generally, the design of SCLP as a knowledge representation (KR) grew out of a detailed requirements analysis [1] for rules in automated contracts and business policies. The RuleML standards effort is being pursued in informal cooperation with the World Wide Web Consortium's Semantic Web Activity, which has now included rules in its charter along with ontologies, and with the DARPA Agent Markup Language Program (DAML) [16].

3. Overview of the rest of the paper

In se ction 3, we review the MIT Process Handbook (PH) [9] [17], and Klein *et al*'s extension of it to treat exception conditions in contracts [18]. In section 4, we newly show how to represent the Process Handbook's process ontology (including about exceptions) in DAML+OIL, giving some examples. In section 5, we discuss the inability of DAML+OIL, however, to represent default (i.e., non-monotonic) inheritance -- which the PH ontology employs in the general case, just as does C++ and many other object-oriented (OO) systems. (Elsewhere, we will detail how the courteous extension to logic programs provides an alternative knowledge representation tool for properly treating such default inheritance.) In section 6, we describe our development of an additional ontology specifically about contracts, again giving examples in DAML+OIL. This contract ontology extends and complements the PH process ontology. In section 7, we newly give an approach to using DAML+OIL ontology as the predicates etc. of RuleML rules. In section 8, we newly show how to use the DAML+OIL process ontology, including about contracts and exceptions, as the predicates etc. of RuleML rules, where a ruleset represents part or all of a (draft or final) contract with exceptions and exception handlers. We illustrate by giving a longish example of such a contract ruleset whose rule-based contingency provisions include detecting and penalizing late delivery exceptions, thus providing means to deter or adjudicate a late delivery. In section 9, we give conclusions and briefly discuss the larger SweetDeal effort. In section 10, we wind up with some discussion including of future work.

3. MIT Process Handbook (PH)

In this section, we review the MIT Process Handbook (PH) [9] [17], and Klein *et al*'s extension of it to treat exception conditions in contracts [18].

The MIT Process Handbook (PH) is a previously-existing knowledge repository of business process knowledge. It is primarily textual and oriented to human-readability although with some useful automation for knowledge management using taxonomic structure. Among automated repositories of business process knowledge, it is unique (to our knowledge) in having a large amount of content and having been frequently used practically by industry business process designers from many different companies. Previous to our work in SweetDeal, however, its content had never been automated in XML, nor had that content ever been represented in any kind of powerful logical knowledge representation – the closest was its use of a fairly conventional Object-Oriented (OO) style of taxonomic hierarchy, as a tool to organize its content for retrieval and browsing.

The Handbook describes and classifies major business processes using the organizational concepts of *decomposition*, *dependencies*, and *specialization*. The Handbook models each process as a collection of activities that can be decomposed into subactivities, which may themselves be processes. In turn, coordination is modeled as the management of dependencies that represent flows of control, data, or material between activities. Each dependency is managed by a coordination mechanism, which is the process that controls its resource flow.

Finally, processes are arranged into a generalization-specialization taxonomy, with generic processes at the top and increasingly specialized processes underneath. Each specialization automatically inherits the properties of its parents, except where it explicitly adds or changes a property. This is similar to taxonomic class hierarchies having default inheritance[2], such as in many Object-Oriented (OO) programming languages, knowledge representations (KR's) and information modeling systems. Note that the taxonomy is not a tree, as an entity may have multiple parents. In general, there thus is multiple inheritance. For example, BuyAsALargeBusiness is a subclass of both Buy and ManageEntity. The figure below shows a part of the taxonomy with some of the specializations for the "Sell" process. Note the first generation of children of "Sell" are questions; these are classes used as intermediate categories, analogous to virtual classes (or pure interfaces) in OO programming languages. Since there is multiple inheritance, it is easy to provide several such "cross-cutting" dimensions of categories along which to organize the hierarchy.

[2] a.k.a. "inheritance with exceptions", a.k.a. "non-monotonic inheritance"

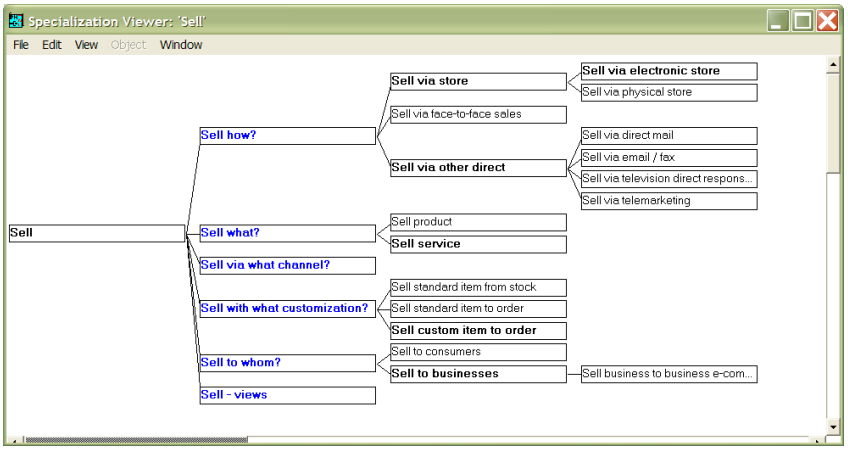

Figure 1: Some specializations of "Sell" in the MIT Process Handbook.

Exception Conditions

The terms of any contract establish a set of commitments between the parties involved for the execution of that contract. When a contract is executed, these commitments are sometimes violated. Often contracts, or the laws or automation upon which they rely, specify how such violation situations should be handled.

Building upon the Process Handbook, Klein *et al* [18] consider these violations to be coordination failures – called "exceptions" – and introduces the concept of exception handlers, which are processes that manage particular exceptions. We in turn build upon Klein *et al*'s approach. When an exception occurs during contract execution, an exception handler associated with that exception may be invoked.

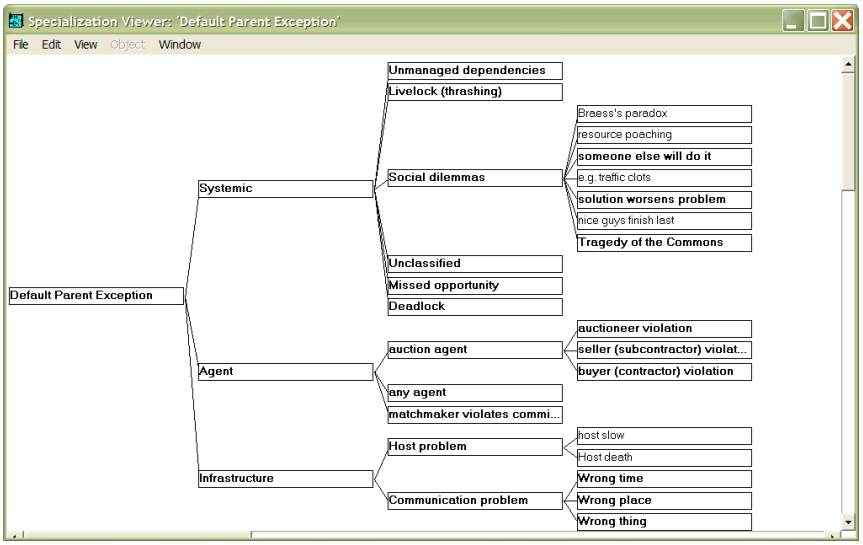

Figure 2: Some exceptions in the MIT Process Handbook.

For example, in a given contract (agreement), company A agrees to pay $50 per unit for 100 units of company B's product, and B agrees to deliver within 15 days (commitments). However, due to unforeseen circumstances, when the contract is actually performed, B only manages to deliver in 20 days (exception). As a result, B pays $1000 to A as compensation for the delay (exception handler).

There are four classes of exception handlers in [18]. For an exception that has not occurred yet, one can use:

- Exception anticipation processes, which identify situations where the exception is likely to occur.
- Exception avoidance processes, which decrease or eliminate the likelihood of the exception.

For an exception that has already occurred, one can use:

- Exception detection processes, which detect when the exception has actually occurred.
- Exception resolution processes, which resolve the exception once it has occurred.

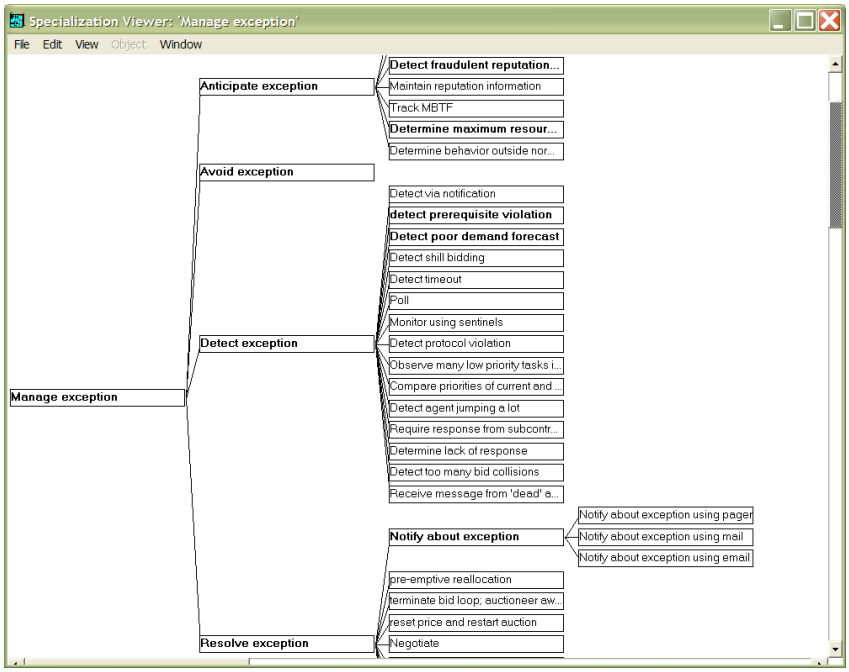

Figure 3: Some exception handlers in the MIT Process Handbook.[3]

[18] extends the MIT Process Handbook with an exception taxonomy. Every process may be associated via *hasException* links to its potential exceptions (zero or more), which are the characteristic ways in which its commitments may be violated. *hasException* should be understood as "has potential exception". Similar to the process taxonomy, exceptions are arranged in a specialization hierarchy, with generic exceptions on top and more specialized exceptions underneath. In turn, each exception is associated (via an *isHandledBy* link) to the processes (*exception handlers*) that can be used to deal with that exception. Since handlers are processes, they may have their own characteristic exceptions.

Following the general style of (default, multiple) inheritance in the MIT Process Handbook, the exceptions associated with a process are inherited by the specializations of that process unless explicitly overridden. Similarly, the handlers for an exception are inherited by the specializations of that exception, unless overridden.

4. Representing the PH Process Ontology in DAML+OIL

In this section, we newly show how to represent the Process Handbook's process ontology (including about exceptions) in DAML+OIL, giving some examples. This on-

[3] *Track MBTF* is a typo in the MIT Process Handbook. It should be *Track MTBF* (mean time between failures) instead.

tology is given a URI of http://xmlcontracting.org/pr.daml, where “pr” stands for “process”. We have registered the xmlcontracting.org domain name and are in the process of setting up the web site. For the current full version of this ontology, and pointer to the xmlcontracting.org site when it is indeed up, please see the first author’s website.

We begin with some DAML+OIL headers:

```
<?xml version="1.0" ?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns     ="http://xmlcontracting.org/pr.daml#" >
<daml:Ontology rdf:about="">
  <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
</daml:Ontology>
```

Next we define some main concepts in the MIT Process Handbook as top-level classes:

```
<daml:Class rdf:ID="Process">
  <rdfs:comment>A process</rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="CoordinationMechanism">
  <rdfs:comment>A process that manages activities between multiple
agents</rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="Exception">
  <rdfs:comment>A violation of an inter-agent commitment</rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="ExceptionHandler">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <rdfs:comment>A process that helps to manage a particular
exception</rdfs:comment>
</daml:Class>
```

Then we define the relations between concepts as object properties:

```
<daml:ObjectProperty rdf:ID="hasException">
  <rdfs:comment>Has a potential exception</rdfs:comment>
  <rdfs:domain rdf:resource="#Process" />
  <rdfs:range rdf:resource="#Exception" />
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="isHandledBy">
  <rdfs:comment>Can potentially be handled, in some way or aspect,
by</rdfs:comment>
  <rdfs:domain rdf:resource="#Exception" />
  <rdfs:range rdf:resource="#ExceptionHandler" />
</daml:ObjectProperty>
```

Specializations are expressed as subclasses[4]:

```
<daml:Class rdf:ID="SystemCommitmentViolation">
  <rdfs:subClassOf rdf:resource="#Exception"/>
  <rdfs:comment>
Violation of a commitment made by the system operator to create an
environment well-suited to the task at hand.
  </rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="AgentCommitmentViolation">
  <rdfs:subClassOf rdf:resource="#Exception"/>
  <rdfs:comment>
Violation of a commitment that an agents makes to other agents.
  </rdfs:comment>
</daml:Class>
```

The Process Handbook expects each specialization to inherit the properties of its parent. The DAML+OIL semantics provide this automatically since it entails monotonic (strict) inheritance of such properties.

Subtlety about Handler Exclusiveness:

Next we discuss a subtlety that arises about the exclusiveness of handler classes. In response to this subtlety, we make use of the `daml:hasClass` restriction. Consider the following fragment, which defines the exception ContractorDoesNotPay as a subclass of ContractorViolation and specifies an `isHandledBy` link to the exception handler ProvideSafeExchangeProtocols.

```
<daml:Class rdf:ID="ContractorDoesNotPay">
  <rdfs:subClassOf rdf:resource="#ContractorViolation"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#isHandledBy"/>
      <daml:hasClass rdf:resource="#ProvideSafeExchangeProtocols"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

Formally, the `daml:Restriction` element here defines an anonymous class of all objects for which at least one value of the `isHandledBy` property is a member of the class ProvideSafeExchangeProtocols. (This does not exclude that there are other values of the `isHandledBy` property that are not members of class ProvideSafeExchangeProtocols.) As a subclass of both ContractorViolation and this restriction, ContractorDoesNotPay is a specialization of ContractorViolation that has at least one instance that is handled by ProvideSafeExchangeProtocols.

Notice we use a `daml:hasClass` restriction here rather than a `daml:toClass` restriction[5] [5]. `daml:toClass` would require that the value of the `isHandledBy`

[4] In Figure 2 (in Section 3), SystemCommitmentViolation and AgentCommitmentViolation are shown as “Systemic” and “Agent”, respectively.

property for ContractorDoesNotPay *must* be of the class ProvideSafeExchangeProtocols. In other words, it would exclude the possibility for any other exception handler class to handle the ContractorDoesNotPay exception. In contrast, `daml:hasClass` leaves open this possibility. This matches the semantics of the *isHandledBy* link in the MIT Process Handbook, which is that some instances of the ContractorDoesNotPay exception are known to be aptly handled by some instances of the ProvideSafeExchangeProtocols handler. The Handbook takes the approach – which we endorse – that it is typically desirable to treat a process repository as potentially extensible, i.e., open. Indeed, it often unrealistic to expect a repository to have an exhaustive listing of all handlers for a given exception.

The Process Handbook is quite large (order of magnitude 10,000 classes). We have (so far) represented in DAML+OIL a relevant fragment amounting to a small percentage of its content. Only some of that fragment is shown in this paper, however. For the full details, see the first author's website.

5. Issue: Default Inheritance

In this section, we discuss the inability of DAML+OIL, however, to represent default (i.e., non-monotonic) inheritance -- which the PH ontology employs in the general case, just as do C++, Java, and many other object-oriented (OO) systems. (Elsewhere, we will detail how the courteous extension to logic programs provides an alternative knowledge representation tool for properly treating such default inheritance.)

The Handbook allows *overrides*[6], where a specialization explicitly omits a property inherited from its generalization. This is impossible to express directly in DAML+OIL, due to its monotonic class system. Consider the following naïve attempt at omitting the isHandledBy ProvideSafeExchangeProtocols property from Special-Nonpayment, a fictitious specialization of ContractorDoesNotPay:

---

[5] The `daml:toClass` restriction is analogous to the universal (for-all) quantifier of predicate logic, while the `daml:hasClass` restriction is analogous to the existential (there-exists) quantifier of predicate logic.

[6] Note this propery "overrides" in the Process Handbook is *not* the same concept as "overrides" in courteous logic programs (CLP). The latter is a syntactically reserved predicate used to specify the prioritization partial ordering among rules/rule-labels. (These two usages of "overrides" are an unfortunate but coincidental case of collision/overloading in the technical terminology.)

```
<daml:Class rdf:ID="SpecialNonPaymentViolation">
  <rdfs:subClassOf rdf:resource="#ContractorDoesNotPay"/>
  <rdfs:subClassOf>
    <daml:Class>
      <daml:complementOf>
        <daml:Restriction>
          <daml:onProperty rdf:resource="#isHandledBy"/>
          <daml:hasClass rdf:resource="#ProvideSafeExchangeProtocols"/>
        </daml:Restriction>
      </daml:complementOf>
    </daml:Class>
  </rdfs:subClassOf>
</daml:Class>
```

Since SpecialNonpayment is a subclass of ContractorDoesNotPay, we would infer that it could have the handler ProvideSafeExchangeProtocols. However, since it is the subclass of the complement of all things that could have the handler ProvideSafeExchangeProtocols, we would infer that it could not have the handler ProvideSafeExchangeProtocols. These two statements contradict each other, implying that the SpecialNonpayment class must be the empty set. If we defined an instance of SpecialNonpayment, a logical inconsistency would result, since it is impossible for such an instance to exist.

Pragmatically, this inability of DAML+OIL to represent *overrides* is not an immediately critical problem in representing the PH ontology, since there are as yet only a few places in the current content of the MIT Process Handbook where *overrides* is used. Coping with this incapability of DAML+OIL about *overrides* is an important area for future work, however, especially since DAML+OIL is becoming a more widely accepted kind of tool.

6. Contract Ontology

In this section, we describe our development of an additional process ontology specifically about contracting concepts and relations, again giving examples in DAML+OIL. This *contract ontology* extends and complements the PH process ontology. We give it the URI http://xmlcontracting.org/sd.daml, where “sd” stands for “SweetDeal”. For the current file version of this ontology, and pointer to the xmlcontracting.org site when it is indeed up, please see the first author’s website.

Again we begin with some DAML+OIL header statements. Notice that we import the PH process ontology:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns     ="http://xmlcontracting.org/sd.daml#" >
<daml:Ontology rdf:about="">
  <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
  <daml:imports rdf:resource="http://xmlcontracting.org/pr.daml"/>
</daml:Ontology>
```

We view a contract as a specification for one or more processes. Accordingly, we define the *Contract* class and a *specFor* relation that links a contract to its process(es):

```
<daml:Class rdf:ID="Contract">
  <rdfs:subClassOf>
    <daml:Restriction daml:minCardinality="1">
      <daml:onProperty rdf:resource="#specFor"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>

<daml:ObjectProperty rdf:ID="specFor">
  <rdfs:domain rdf:resource="#Contract" />
  <rdfs:range rdf:resource="http://xmlcontracting.org/pr.daml#Process"
/>
</daml:ObjectProperty>
```

To represent the common special case of contracts that specify only one process, we define *ContractForOneProcess*, using a `daml:cardinality` restriction to limit the *specFor* relation to exactly one process:

```
<daml:Class rdf:ID="ContractForOneProcess">
  <rdfs:subClassOf rdf:resouce="#Contract"/>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#specFor"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

A contract represents the "terms and conditions" that the parties have agreed upon (typically) *before* performing the contract. E.g., they have come to agreement during a negotiation before their contract commitments actually come due. We define a separate concept, *ContractResult*, to represent the state of how the contract was actually carried out. For example, *ContractResult* could describe the actual shipping date, the quality of the received goods, the amount of payment received, etc.

```
<daml:Class rdf:ID="ContractResult"/>

<daml:ObjectProperty rdf:ID="result">
  <rdfs:domain rdf:resource="#Contract" />
  <rdfs:range rdf:resource="#ContractResult" />
</daml:ObjectProperty>
```

The process ontology provides the *hasException* relation to indicate that a process *could* have a particular exception. How do we indicate that an exception has occurred during contract execution? We define the *exceptionOccurred* relation on *ContractResult* to denote that the exception happened as the contract was being carried out:

```
<daml:ObjectProperty rdf:ID="exceptionOccurred">
  <daml:domain
rdf:resource="http://xmlcontracting.org/pr.daml#ContractResult"/>
  <daml:range
rdf:resource="http://xmlcontracting.org/pr.daml#Exception"/>
```

```
</daml:ObjectProperty>
```

Finally, we introduce some relations to specify the purpose that an exception handler fulfills. A *DetectException* handler is intended to detect certain exception classes, an *AnticipateException* handler is intended to anticipate certain exception classes, etc. We want to identify exception classes, not exception instances. We thus make the range be the class Class.[7]

```
<daml:ObjectProperty rdf:ID="detectsException">
 <daml:domain
rdf:resource="http://xmlcontracting.org/pr.daml#DetectException"/>
  <daml:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="anticipatesException">
  <daml:domain
rdf:resource="http://xmlcontracting.org/pr.daml#AnticipateException"/>
  <daml:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="avoidsException">
  <daml:domain
rdf:resource="http://xmlcontracting.org/pr.daml#AvoidException"/>
  <daml:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="resolvesException">
  <daml:domain
rdf:resource="http://xmlcontracting.org/pr.daml#ResolveException"/>
  <daml:range
rdf:resource="http://www.daml.org/2001/03/daml+oil#Class"/>
</daml:ObjectProperty>
```

There are a number of other interesting concepts and ontological statements about contracts that we are developing in our SweetDeal Contract Ontology, but space prevents us from detailing them further here.

7. Integrating DAML+OIL Ontologies Into RuleML Rules

In this section, we briefly describe the technical representational approach for the integration of the DAML+OIL ontologies into the RuleML rules, in which the RuleML rules are specified "on top of" the DAML+OIL ontology. In the next section, we give examples of RuleML contract rules that make use of DAML+OIL process ontologies.

---

[7] This is based on the interpretation that an instance of class Class is a class, something which we could not find explicitly addressed in the DAML+OIL reference manual. Also, ideally, we would restrict the range to subclasses of Exception, but we did not see a straightforward way to do this in the current version of DAML+OIL.

A DAML+OIL class is treated as a unary predicate. A DAML+OIL property is treated as a binary predicate. Assertions about instances in a class are treated as rule atoms (e.g., facts) in which the class predicate appears. Assertions about property links between class instances are treated as rule atoms in which the property predicate appears. RuleML permits a predicate symbol (or a logical function symbol) to be a URI; we make heavy use of this capability since the names of DAML+OIL classes are URI's. To our knowledge, ours is the first published description and example of such integration of DAML+OIL into RuleML, and one of the first two published descriptions and examples of combination of DAML+OIL with a non-monotonic rule KR -- the other being [19] which was done independently.[8]

A natural question arises: how to define formally the semantics of such integration, i.e., of the *hybrid* KR formed by combining LP rules on top of DL ontologies (or, similarly, by combining Horn FOL rules on top of DL ontologies). One might view this through the lens of the rule KR's semantics and/or through the lens of the ontology KR's semantics. Knowledge specified in a set of premise rules *R1* together with a set of premise DL axioms *O1* may entail knowledge *O2* expressible in DL that goes beyond what was entailed by *O1* alone, and likewise may entail knowledge *R2* expressible in LP that goes beyond what was entailed by *R1* alone. It is also possible, in general, for inconsistency to arise from the combination of *R1* with *O1*, even though each is consistent in itself. We do not have space or focus here to give more details about the formal definition of this semantics, see elsewhere for that. A somewhat similar hybrid KR semantics and proof theory is addressed in [19].

8. RuleML Contracts with Exceptions using the Process and Contract Ontologies

In this section, we newly show how to use the DAML+OIL process ontology, including about contracts and exceptions, as the predicates etc. of RuleML rules, where a ruleset represents part or all of a (draft or final) contract that has exceptions and exception handlers. We illustrate by giving a long-ish example of such a contract ruleset whose rule-based contingency provisions include detecting and penalizing late delivery exceptions, thus providing means to deter or adjudicate a late delivery.

RuleML, like most XML, is fairly verbose. For ease of human-readability, as well as to save paper space, we give our RuleML examples in a Prolog-like syntax that maps straightforwardly to RuleML. More precisely, this syntax is IBM CommonRules V3.0 "SCLPfile" format, extended to support URI's as logical predicate (and function) symbols and to support names for rule subsets (i.e., "modules") . "<-" stands for implication, i.e., "if". ";" ends a rule statement. The prefix "?" indicates a logical variable. "//" prefixes a comment line. "<…>" encloses a rule label (name) or rule module label. "{…}" encloses a rules module. Rule labels identify rules for editing and prioritized conflict handling, for example to facilitate the modular modifica-

[8] We have described this approach verbally in communal design discussions about DAML and about RuleML since when those discussions began in summer 2000. The overall goal of rules on top of ontologies has, indeed, been a communal goal in those discussions since then.

tion of contract provisions. Module labels are used to manage the merging of multiple rule modules to form a contract.

In the examples below, DAML+OIL classes and properties, taken from the PH process ontology and contract (process) ontology, are used as predicate symbols.

Notice that we do not show namespace qualification of the predicate names. This is because the current QName ("*q*ualified" name) syntactic mechanism only works for XML element names but not for attribute or text values. (In RuleML a predicate name is a text value.) For instance, with `daml` defined as `http://www.daml.org/2001/03/daml+oil#`, `daml:Class` serves as shorthand for `http://www.daml.org/2001/03/daml+oil#Class` in element names, but the same syntax cannot be used for values. One workaround approach is to use XML entities. If we define an XML entity `sd` to be `http://xmlcontracting.org/sd.daml#`, then `&sd;Contract` expands to `http://xmlcontracting.org/sd.daml#Contract` in attribute or text values. This issue is a subject of active discussion currently in the RDF and Semantic Web standards efforts.

Let's begin with an example draft contract `co123` where Acme is purchasing 100 units of plastic product #425 from Plastics Etc. at $50 per unit. Acme requires Plastics Etc. to ship the product no later than three days after the order is placed [9]. We specify this draft contract as the following rulebase (i.e., set of rules):

```
http://xmlcontracting.org/sd.daml#Contract(co123);
http://xmlcontracting.org/sd.daml#specFor(co123,co123_process);
http://xmlcontracting.org/sd.daml#BuyWithBilateralNegotiation(
  co123_process);
http://xmlcontracting.org/sd.daml#result(co123,co123_res);

buyer(co123,acme);
seller(co123,plastics_etc);
product(co123,plastic425);
shippingDate(co123,3); // i.e. 3 days after the order is placed

price(co123,50);
quantity(co123,100);
// base payment = price * quantity
payment(?R,base,?Payment) <-
  http://xmlcontracting.org/sd.daml#result(co123,?R) AND
  price(co123,?P) AND quantity(co123,?Q) AND
  Multiply(?P,?Q,?Payment) ;
```

Continuing our example, suppose the seller wants to include a contract provision to penalize late delivery – so as to reassure the buyer. First we add some rules to declare that this contract has an exception instance `e1` that is an instance of the LateDelivery class from the process ontology:

```
http://xmlcontracting.org/pr.daml#hasException(co123_process,e1);
http://xmlcontracting.org/pr.daml#LateDelivery(e1);
```

[9] Here we use a relative date (e.g. 3) rather than an absolute date (e.g. 2002-04-02), for sake of simplicity and because the rule engine that we are using in our prototype (IBM CommonRules) does not (yet) provide convenient date arithmetic functions.

Note that the actual occurrence of an exception is associated with a contract result, as opposed to its potential occurrence which is associated with the contract (agreement)'s process. hasException specifies the potential occurrence. We will see below more about the actual occurrence.

Next, we give a rules module (i.e., a set of additional rules to include in the overall draft contract ruleset) that specifies a basic kind of exception handler process – to detect the late delivery.

In our approach, exception handler processes themselves may be rule-based (in part or totally), although in general they need not be rule-based at all. The exception handler detectLateDelivery is rule-based in this example. Below, the variable ?CO stands for a contract, ?R for a contract result, ?EI for an exception instance, ?PI for a process instance, ?COD for a promised contract shipping date, and ?RD for a contract result's actual shipping date.

```
<detectLateDelivery_module> {

// detectLateDelivery is an instance of DetectPrerequisiteViolation
//   (and thus of DetectException, ExceptionHandler, and Process)

http://xmlcontracting.org/pr.daml#DetectPrerequisiteViolation(
  detectLateDelivery) ;

// detectLateDelivery is intended to detect exceptions of class
// LateDelivery

http://xmlcontracting.org/sd.daml#detectsException(detectLateDelivery,
  http://xmlcontracting.org/pr.daml#LateDelivery);

// a rule defines the actual occurrence of a late delivery in a contract
// result

<detectLateDelivery_def>
http://xmlcontracting.org/sd.daml#exceptionOccurred(?R, ?EI) <-
  http://xmlcontracting.org/sd.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#LateDelivery(?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,
                                                detectLateDelivery) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  shippingDate(?CO,?COD) AND shippingDate(?R,?RD) AND
  greaterThan(?RD,?COD) ;
}
```

Then we add the following rule to the contract to specify detectLateDelivery as a handler for e1:

```
<detectLateDeliveryHandlesIt(e1)>
http://xmlcontracting.org/pr.daml#isHandledBy(e1,detectLateDelivery);
```

Merely detecting late delivery is not enough; the buyer also wants to get a penalty (partial refund) if late delivery occurs. Continuing our example, we next give a rules module to specify a penalty of $200 per day late, via an exception handler process lateDeliveryPenalty. Again, this handler is itself rule-based.

```
lateDeliveryPenalty_module {

// lateDeliveryPenalty is an instance of PenalizeForContingency
//   (and thus of AvoidException, ExceptionHandler, and Process)

http://xmlcontracting.org/pr.daml#PenalizeForContingency(
  lateDeliveryPenalty) ;

// lateDeliveryPenalty is intended to avoid exceptions of class
// LateDelivery.

http://xmlcontracting.org/sd.daml#avoidsException(lateDeliveryPenalty,
  http://xmlcontracting.org/pr.daml#LateDelivery);

// penalty = - overdueDays * 200 ; (negative payment by buyer)
<lateDeliveryPenalty_def> payment(?R, contingentPenalty, ?Penalty) <-
  http://xmlcontracting.org/sd.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/pr.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,lateDeliveryPenalty)
AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  http://xmlcontracting.org/sd.daml#exceptionOccurred(?R,?EI) AND
  shippingDate(?CO,?CODate) AND shippingDate(?R,?RDate) AND
  subtract(?RDate,?CODate,?OverdueDays) AND
  multiply(?OverdueDays, 200, ?Res1) AND multiply(?Res1, -1, ?Penalty) ;
}
```

We add a rule to specify lateDeliveryPenalty as a handler for e1:

```
<lateDeliveryPenaltyHandlesIt(e1)>
http://xmlcontracting.org/pr.daml#isHandledBy(e1,lateDeliveryPenalty);
```

During contract execution, if Plastics Etc. ships its product 8 days after the order is placed (i.e. 5 days later than the agreed-upon date), then the rules `detectLateDelivery` will declare that late delivery exception has occurred, which will trigger `lateDeliveryPenalty` to impose a penalty of $200 per day late, totaling $1000.

More precisely, suppose we represent the contract result as the ruleset formed by adding (to the above contract) the following “result” fact:

```
shippingdate(co123_res, 8) ;
```

Then the contract result ruleset entails various conclusions, in particular

```
http://xmlcontracting.org/sd.daml#exceptionOccurred(co123_res,e1) ;
payment(co123_res, contingentPenalty, -1000) ;
```

Our SweetRules prototype system, which implements SCLP RuleML inferencing, can generate these conclusions automatically.

Next, we (relatively briefly, due to space constraints) illustrate how to use prioritized conflict handling, enabled by the courteous feature of SCLP RuleML, to modularly modify the contract provisions, e.g., during bilateral negotiation. The seller

might like to specify that the late delivery exception should be handled by the handler lateDeliveryRiskPayment, which imposes an up-front insurance-like discount to compensate for the risk of late delivery, basing risk upon a historical average probability distribution (defined separately) of delivery lateness. First, we define a rules module for the risk payment handler:

```
lateDeliveryRiskPayment_module {

// lateDeliveryRiskPayment is an instance of AvoidException
//   (and thus of ExceptionHandler, and Process)

http://xmlcontracting.org/pr.daml#AvoidException(
  lateDeliveryRiskPayment) ;

// lateDeliveryRiskPayment is intended to avoid exceptions of class
// LateDelivery.

http://xmlcontracting.org/sd.daml#avoidsException(
 lateDeliveryRiskPayment,
 http://xmlcontracting.org/sd.daml#LateDelivery) ;

// penalty = - expected_lateness * 200 (negative payment by buyer)
<lateDeliveryRiskPayment_def>
payment(?R, contingentRiskPayment, ?Penalty) <-
  http://xmlcontracting.org/sd.daml#specFor(?CO,?PI) AND
  http://xmlcontracting.org/sd.daml#hasException(?PI,?EI) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI,
lateDeliveryRiskPayment) AND
  http://xmlcontracting.org/sd.daml#result(?CO,?R) AND
  historical_probabilistically_expected_lateness(?CO, ?EOverdueDays) AND
  Multiply(?EOverdueDays, 200, ?Res1) AND Multiply(?Res1, -1, ?Penalty);
}
```

Then we add a rule to specify lateDeliveryRiskPayment as a handler for e1:

```
<lateDeliveryRiskPaymentHandlesIt(e1)>
http://xmlcontracting.org/pr.daml#isHandledBy(e1,
                                              lateDeliveryRiskPayment);
```

Next, we give some rules that use prioritized conflict handling to specify that late deliveries should be avoided by lateDeliveryRiskPayment rather than any other candidate avoid-type exception handlers for the late delivery exception (here, simply, lateDeliveryPenalty). We specify this using a combination of a MUTEX statement and an overrides statement that gives the lateDeliveryRiskPaymentHandlesIt(e1) rule higher priority than the lateDeliveryPenaltyHandlesIt(e1) rule.

```
// There is at most one avoid handler for a given exception instance.
// This is expressed as a MUTual EXclusion between two potential
// conclusions, given certain other preconditions.
// The mutex is a consistency-type integrity constraint, which is
// enforced by the courteous aspect of the semantics of the rule KR.

MUTEX
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI, ?EHandler1) AND
  http://xmlcontracting.org/pr.daml#isHandledBy(?EI, ?Ehandler2)
GIVEN
  http://xmlcontracting.org/sd.daml#AvoidException(?Ehandler1) AND
```

```
  http://xmlcontracting.org/sd.daml#AvoidException(?Ehandler2) ;

// The rule lateDeliveryRiskPaymentHandlesIt(e1) has higher priority
// than the rule lateDeliveryPenaltyHandlesIt(e1).

overrides(lateDeliveryRiskPaymentHandlesIt(e1),
          lateDeliveryPenaltyHandlesIt(e1) ) ;
```

Now suppose the probabilistically expected lateness of the delivery (before actual contract execution) is 3 days. I.e., suppose the contract also includes the following fact.

```
historical_probabilistically_expected_lateness(co123, 3) ;
```

If upon execution the modified-contract's result facts are as before – i.e., delivery is 5 days late – then the modified-contract's result entails as conclusions that the late delivery will be handled by the up-front risk payment of $600 = (3 days * $200).

```
payment(co123_res, contingentRiskPayment, -600) ;
```

The modified-contract's result does *not* entail that late delivery is handled by the penalty of $1000 – as it should not. The courteous aspect of the rules knowledge representation has properly taken care of the prioritized conflict handling to enforce that the new higher-priority contract provision about risk payment dominates the provision about penalty.

9. Conclusions and Overall SweetDeal Effort

To recap, this work makes novel contributions in several areas:

- Represents process knowledge from the MIT Process Handbook (PH) using an emerging Semantic Web ontology KR (DAML+OIL). This is the first time PH process knowledge has been represented using XML or powerful KR.
- Extends our previously existing SweetDeal approach to rule-based representation of contracts in SCLP/RuleML with the ability to reference such process knowledge and to include exception handling mechanisms. (The SweetDeal approach enables software agents to create, evaluate, negotiate, and execute contracts with substantial automation and modularity.)
- Enables thereby more complex contracts with behavioral provisions.
- Provides a foundation for representing and automating contractual *deals about Web Services* (and e-services more generally), so as to help search, select, and compose them.
- Gives a new point of convergence between Semantic Web and Web Services.
- Gives a conceptual approach to specifying LP/RuleML rules "on top of" DL/DAML+OIL/DL ontologies (for the first time to our knowledge). Moreover, this is for the highly expressive SCLP case of RuleML. And this is one of the first two published descriptions and examples of combination of DAML+OIL with a *non-monotonic* rule KR — the other being [19] which was

done independently. Our approach to rules on top of ontologies is described here conceptually and by examples, but only informally, however, in that we do not have space or focus to give a formal semantics (or proof theory) for it.
- Combines (SC)LP/RuleML with DL/DAML+OIL (i.e., emerging Semantic Web rules with emerging Semantic Web ontologies) *for a substantial business application domain scenario/purpose* (for the first time, to our knowledge).

A prototype is running. We intend to make it publicly available in the near future.

This paper also:
- Demonstrates an inherent limitation of DAML+OIL in representing inheritance overrides (a.k.a. "default inheritance", a.k.a. "inheritance with exceptions"). The limitation is due to the logical monotonicity of its underlying Description Logic KR. The limitation was well known in principle, but we gave here one of the first practical example application contexts in which it arises.

For more discussion of conclusions, see the Introduction.

The larger SweetDeal effort further:
- Defines and implements a queryable *process repository*.
- Defines and implements a mechanism for rule-based contracts in RuleML to be built from reusable modular provisions, called *contract fragments*, that are retrieved from queryable *contract repositories*.
- Designs and implements the mechanisms of a *market agent* that largely automates the creation of such contracts as part of a negotiation process, in support of a human user.
- Provides an overall interaction architecture for an agent marketplace with such rule-based contracts.

10. Future Work

One interesting research direction is to develop more and longer example scenarios and test them out by running them using SweetRules together with tools for DAML+OIL and, later, tools for reasoning specifically about process knowledge. In particular, we are investigating aspects specific to Web Services. We are focusing on relating our SweetDeal approach and its elements (rules, ontologies, process knowledge) to the Web Services area's standards (e.g., WSDL [24]), techniques (e.g., SOAP invocations [23], UDDI [25]), and exploratory application areas. A second interesting direction is how to incorporate legal aspects of contracting into our approach, including to connect to the Legal XML emerging standards effort [26].

Other interesting directions involve ontologies. One is to further develop the DAML+OIL ontology for business processes, e.g., by drawing on the Process Handbook. A second is to further develop the contract ontology. Currently, we are inves-

tigating how to formalize more deeply the relationship between a contract rulebase and a rule-based handler process. Third, we are prototyping an approach to using DAML+OIL as an ontology about RuleML itself. We have released an early version of this specification. Called DamlRuleML [14], it provides a relatively straightforward encoding of SCLP RuleML's syntax in DAML+OIL. A natural next step is then to combine DamlRuleML with a DAML+OIL domain ontology (e.g. about processes) whose classes serve as predicates for RuleML rules, as presented in this paper.

Another important aspect of ontologies is to develop the theory of combining rules on top of ontologies, including expressive union and intersection, semantics, proof theory, algorithms, and computational complexity. Our development of this theory is in progress.

Yet other directions for future work include tying in to agent negotiation strategies, to emerging standards for general-purpose e-business/agent communication, e.g., ebXML [20] and FIPA's Agent Communication Language [21], and to more general efforts on combining Semantic Web and Web Services, e.g., DAML-S.

Finally, there is the challenge we discussed in section 5: how to cope with the issue of default inheritance in regard to DAML+OIL and also to the Process Handbook. In current work, we are taking an approach to default inheritance using the prioritized conflict handling capability provided by the courteous feature of SCLP.

Acknowledgements

Thanks to our MIT Sloan colleagues Mark Klein, Chrysanthos Dellarocas, Thomas Malone, Peyman Faratin, and John Quimby for useful discussions about the Process Handbook and representing contract exceptions there. Also thanks to the workshop's anonymous reviewers for numerous helpful comments.

References

1. 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.
2. 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.)
3. Rule Markup Language Initiative, http://www.dfki.de/ruleml and http://www.mit.edu/~bgrosof/#RuleML.
4. 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.
5. DAML+OIL Reference (Mar. 2001). http://www.w3.org/TR/daml+oil-reference/
6. Semantic Web Activity of the World Wide Web Consortium. http://www.w3.org/
7. Web Services activity of the World Wide Web Consortium. http://www.w3.org/

8. DAML Services Coalition (alphabetically A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, H. Zeng), ''DAML-S: Semantic Markup for Web Services", Proc. International Semantic Web Working Symposium (SWWS), 2001.
9. MIT Process Handbook. http://ccs.mit.edu/ph/
10. XSB logic programming system. http://xsb.sourceforge.net/
11. Niemela, I. and Simons, P., Smodels (version 1). http://saturn.hut.fi/html/staff/ilkka.html.
12. IBM CommonRules. http://www.alphaworks.ibm.com/ and http://www.research.ibm.com/rules/
13. Knowledge Interchange Format http://logic.stanford.edu/kif and http://www.cs.umbc.edu/kif/. A new effort is named CommonLogic.
14. Grosof, B.N., Gandhe, M.D., and Finin, T.W., "SweetJess: Translating DamlRuleML to Jess". Proc. Intl. Wksh. on Rule Markup Languages for Business Rules on the Semantic Web, held at 1st Intl. Semantic Web Conf., 2002.
15. Jess (Java Expert System Shell). http://herzberg.ca.sandia.gov/jess/
16. DARPA Agent Markup Language Program http://www.daml.org/
17. Malone, T.W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., Quimby, J., Osborn, C.S.., Bernstein, A., Herman, G., Klein, M., and O'Donnell, E., "Tools for Inventing Organizations: Toward a Handbook of Organizational Processes." *Management Science*, 45(3): p. 425-443, 1999.
18. Klein, M., Dellarocas, C., and Rodríguez-Aguilar, J.A., "A Knowledge-Based Methodology for Designing Robust Multi-Agent Systems." Proc. Autonomous Agents and Multi-Agent Systems, 2002.
19. Antoniou, G., "Nonmonotonic Rule Systems using Ontologies". Proc. Intl. Wksh. on Rule Markup Languages for Business Rules on the Semantic Web, held at 1st Intl. Semantic Web Conf., 2002.
20. ebXML (ebusiness XML) standards effort, http://www.oasis-open.org
21. FIPA (Foundation for Intelligent Physical Agents) Agent Communication Language standards effort, http://www.fipa.org
22. XSLT (eXtensible Stylesheet Language Transformations), http://www.w3.org/Style/XSL/
23. SOAP, http://www.w3.org/2000/xp/Group/ and http://www.w3.org/2002/ws/
24. WSDL (Web Service Definition Language), http://www.w3.org/2002/ws and www.w3.org/TR/wsdl
25. UDDI (Universal Description, Discovery, and Integration), http://www.uddi.org
26. Legal XML, http://www.oasis-open.org