LTML — A Language for Representing Semantic Web Service Workflow Procedures [*]

Mark Burstein[1], Robert P. Goldman[2], Drew V. McDermott[3], David McDonald[1], Jacob Beal[1], and John Maraist[2]

[1] {burstein,dmcdonald,jbeal}@bbn.com, BBN, 10 Moulton St., Cambridge, MA
[2] {rpgoldman,jmaraist}@sift.info, SIFT, LLC, Minneapolis, MN
[3] dvm@cs.yale.edu, Computer Science Department, Yale University, New Haven, CT

Abstract. The Learnable Task Modeling Language (LTML) was developed by combining features of OWL, OWL-S, and PDDL, using a more compact and readable syntax than OWL/RDF to create human readable representations of web service procedures and hierarchical task models. Our goal was in part to develop a more robust and developer-friendly language based on the principles and design that led to OWL-S and demonstrate that such a language also provided the basis for developing tools that could learn web service procedures by demonstration. LTML's initial and driving use is as an interlingua for the learning and procedure execution components of POIROT, a system that learns web service workflow procedures from 'observations' of one or a small number of semantic web service traces. The LTML language uses an s-expression based syntax for improved readability but has parsers and generators that translate the surface forms into RDF for storage in a SESAME triple store implementing POIROT's internal blackboard. All language elements are grounded in a set of OWL ontologies. The language encompasses and extends coverage of the OWL-S process and grounding models, and introduces elements to support sets of hierarchical task methods indexed by goals, semantic execution traces, and internal tasks and learning goals. This short paper gives an overview of LTML and describes the areas where LTML diverges from or extends OWL-S and PDDL.

1 Introduction

The Learnable Task Modeling Language (LTML) was developed to provide human and machine readable representations of semantic web service procedures and hierarchical task models suitable for use by a workflow learning and execution system. POIROT (Plan Order Induction by Reasoning from One Trial) [1] is the system for which LTML was initially designed and by which it's utility has been demonstrated. POIROT utilizes an RDF-based blackboard architecture and meta-controller to orchestrate a multi-strategy learning process that analyzes semantic web service execution traces and learns web service procedures. Specifically, POIROT learns and can then execute hierarchical task models given an execution trace of a single demonstration of a repetitive web

service task. Here, traces are OWL representations of sequences of instances of semantic web service calls. The services (atomic processes) are represented using OWL-S. The hierarchical task models that POIROT learns can then be executed by SHOPPER [2], a component of POIROT that interprets LTML procedures and calls web services. SHOPPER in turn uses a version of the OWL Virtual Machine (OVM) [3] to invoke the individual semantic web services. Figure 1 shows the overall structure of the POIROT system.
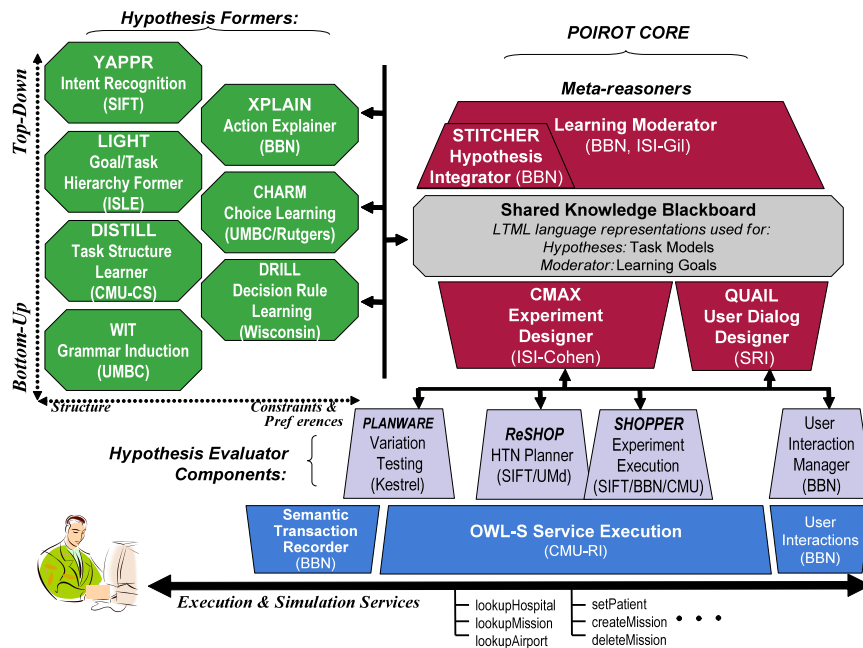

Fig. 1. Overview of POIROT Architecture

The workflow models that POIROT as a whole learns are the result of combining a number of incomplete models learned by POIROT components that reason about

– segmentation of the trace into subtasks (YAPPR [4]),
– causal explanations of related trace elements (XPLAIN [5]),
– dataflow (service output to other service input) dependencies (IODA),
– overall temporal ordering structure (WIT [6]),
– loops and conditional branches (DISTILL [7]).

The different learned products are merged together in a two step process by the Stitcher, which reasons about parallel structures, and ReSHOP (based on SHOP2), which reasons about goal achievement.

POIROT processing starts when the system receives one or more traces of human task performance. A trace is an example of a human operator carrying out some particular workflow, captured by intercepting the inputs and outputs of a set of web services. The trace will be analyzed by Yappr and XPLAIN, each of which will write its analysis, in LTML, into the POIROT blackboard. The blackboard contents will be analyzed by IODA, WIT, and DISTILL, which will write partial method definitions learned from the trace and its analysis, into the blackboard. The different method definitions are then merged by the Stitcher. Finally, in order to experiment with the learned method definitions, ReSHOP will be activated to assemble the methods into a goal-achieving workflow, which will be fed to SHOPPER for execution.

POIROT components share open learning goals, hypotheses, internal tasks and learned workflow models via a central blackboard that is implemented as an RDF store (SESAME). All communication with that store is in one of several 'dialects' of LTML. Traces, methods (hierarchical procedures), class definitions and service definitions use sytactic forms hide complex or repetitive idioms required to express the content in OWL and make them more readable/editable by people. Most other aspects of the communication use what we call the 'striped' form of LTML which is an s-expression variant of N3[8], a compact notation for RDF, relying directly on OWL concepts and properties.

In this paper we will briefly present some of the syntactic forms that were used to make what is at heart an extension of the OWL-S procedure language more accessible, and then describe some of the extensions to the language and ontology that we found necessary to represent the procedures that we were learning.

2 Background: OWL-S to LTML

Briefly, OWL-S (Owl for Services) [9, 10] is a set of OWL [11] ontologies for describing web services and web service procedures. It has three main subparts: Profile, Process and Grounding. The OWL-S Profile ontology is used to represent what the service is used for and when its use is appropriate, including issues such as quality of service. The Process ontology describes what the service does (its inputs, outputs and effects) and the Grounding ontology describes how it is executed, specifically how parts of the process ontology relate to a specific model of how it is called by relating the semantic model to a more syntactic calling specification like WSDL[12] or SOAP[13]. This has typically included a mapping described using XSLT[14].

LTML is mainly concerned with Process representations[4] It currently uses OWL-S groundings directly. The OWL-S process ontology describes *atomic* and *composite* processes, which represent, respectively, the elements of individual service calls, and combinations of service calls connected by data and control flow constructs. Though somewhat different in some of its details, the LTML process ontology borrows heavily from that model. An example of the difference is that it represents sequences of steps using lists modeled as sets of instances with numbered elements rather than a first/rest structure. We will mention several other differences shortly. Atomic processes on the

---

[4] One could argue that its means of associating goals with methods (the *to-achieve* form) is similar to one function of Profiles.

other hand are modeled nearly identically, allowing us to use the OWL Virtual Machine to execute atomic services.

The main purpose of the LTML language is to be able to capture semantic web service execution traces and workflow descriptions that are learned from those traces by POIROT's learning processes. The traces and workflows reference semantic descriptions of web services, which in OWL-S were called atomic processes. We needed a more compact surface notation than OWL/XML to be able to work effectively with the complex OWL-S procedure models being generated by POIROT, and we also needed to extend the language in several ways to make it possible to represent all of the aspects of the procedures and traces that POIROT was reasoning over. LTML's surface syntax provides notations for workflows (composite processes) and traces to simplify their textual form and make them more readable to developers such as those reviewing what the POIROT components have learned about the traces that were demonstrated. It borrows some ideas from an initial design for a surface syntax for OWL-S developed by McDermott [15].

We required a language that was both human and machine readable, and could be interpreted in both LISP and Java™ so that it could be used as an interlingua between components written in either language to learn about, planning with or executing web service procedures. What developed was an considerably easier-to-use s-expression oriented surface notation for something very close to OWL and OWL-S descriptions of web services, web service procedures and the ontologies supporting their description of specific domains processes. Additional ontologies and some language features were then added to capture other parts of the the AI planning competition language PDDL [16][5] and the SHOP-2 hierarchical task network planning language [17].

LTML encompasses the expressive power of OWL by providing syntactic forms that are directly translatable into OWL class and property definitions and descriptions of individuals. LTML's primitive service definitions capture both conventional planning operators (as in PDDL) and OWL-S atomic processes, while providing a few additional features to overcome shortcomings of OWL-S representations, especially for service effects. Finally, LTML adds higher-level control flow constructs to compose these atomic actions into *methods*, similar to OWL-S composite processes, and uses these forms to describe web service workflows.

All LTML surface syntactic forms for processes have translations into OWL using an OWL ontology very similar to the OWL-S Process ontology. Other OWL concepts are used to describe event sequences, specifically service execution traces. In the POIROT system, all LTML surface forms are translated, using these ontologies, into RDF triples for storage in an RDF triple-store (SESAME), and all ontologies are translated to and represented in OWL Full. This allows these forms to be easily combined with other OWL representations for such things as sets of methods (subprocedures) that comprise a learner's hypothesis about how to perform an observed procedure.

---

[5] Note that, as an extension of PDDL, LTML is also able to capture conventional AI planning problems.

3   LTML Language Overview

LTML has a Lisp-like syntax, or rather two syntaxes. LTML has a *surface* form intended for human readability and writeability. LTML also may be written in *striped* notation. Striped notation is a Lisp-like notation for RDF triples, essentially equivalent to N3 [8]. The syntax for striped descriptions is simply:

```
(Classname instanceName (propertyName value)*)
```

where the *value* is another description, and *Classname*, *instanceName* and *property-Name* are LTML *symbols*. LTML symbols consist of a namespace abbreviation, an '@', and a local *namestring*. The '@' replaces the OWL ':'.

Striped syntax translates into RDF in the obvious way. The pattern creates an RDF description for *instanceName*, declaring it of type *Classname*, with properties specified by *propertyName* and with corresponding *values*. So, for example,

```
(animals@Dog Fido (pet@belongsTo Mark)
                  (pet@chases (animals@Cat Trixie)))
```

translates to the RDF/XML:

```
<animals:Dog rdf:ID="Fido">
  <pet:belongsTo rdf:resource="Mark">
  <pet:chases>
      <animals:Cat rdf:resource="Trixie"/>
  </pet:chases>
</animal:Dog>
```

The rest of the paper focuses on the compact surface form.

Classes and properties LTML provides a syntax for describing classes and properties. Again, the "@" is a namespace separator. LTML namespace abbreviations map to XML namespace abbreviations in the obvious way.

```
(in-namespace trans)
(Class Airport
   (subClassOf loc@Location)
   (subClassOf top@Entity)
   (restrict airportLocationID (cardinality 1) - LocationID))
```

For comparison, here is the definition of Airport translated into OWL/XML[6]

```
<owl:Class rdf:ID="Airport">
  <rdfs:subClassOf rdf:resource="&loc;#Location"/>
```

---

[6] For brevity, we use entity abreviations for the namespaces.

```
    <rdfs:subClassOf rdf:resource="&top;#Entity"/>
    <rdf:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="&loc;#airportLocationID"/>
            <owl:allValuesFrom rdf:resource="&loc;#LocationID"/>
            <owl:cardinality
             rdf:datatype=
                "&xsd;#nonNegativeInteger">1</owl:cardinality>
        </owl:Restriction>
    </rdf:subClassOf>
</owl:Class>
```

The main "syntactic sugar" here is the simplified form for property restrictions on classes. The `restrict` keyword is followed by a property name, an optional cardinality description and a type restrictions in a single ordered expression. As in other places in the language, type restrictions are indicated by a dash '-' followed by a type. The *restict* clause also declares a new property if it was not previously defined.

Properties are defined in a similar way. ObjectProperties and DatatypeProperties are recognized based on their range.

```
(Property airportLocationID
    (domain Airport)
    (range LocationID))
```

Atomic process definition Figure 2 is an example of an LTML atomic process (semantic web service) definition. As with methods, services have inputs and outputs. In addition, however they have preconditions and results.[7]

```
 (AtomicProcess reserveSeat
   (inputs flightNo - travel@FlightID
           passengerID - travel@PassengerID
           confirmationCode - travel@ConfirmCode)
   (outputs outcomeFlag - svc@ResultCode
             reservationID - airSvc@ReservationID)
   (precondition
     (travel@paymentRcvd passengerID flightNo confirmationCode))
   (result
     (when (svc@Success outcomeFlag)
       (ThereIs ((p (referent travel@Passenger passengerID))
                 (f (referent travel@Flight flightNo)))
         (trans@hasReservedSeat p f reservationID)
         (increment-fluent (travel@passengers f) 1)))))
```

Fig. 2. Sample web service definition in LTML.

---

[7] Methods can also have preconditions and effects, but they are normally derived from their consitituents.

This definition includes what an invoking agent should provide to the web service (**inputs**), the service precondition, which is the condition that the agent should check and make sure is true before invoking the service, and what the agent can expect back (**outputs**). Other conditions on successful service execution that are based on information known to the service provider appear in conditional effects. For example, for the `reserveSeat` service, a reservation failure due to seat capacity being exceeded would be expressed as a conditional *effect* since the agent making the reservation would not be expected to know the plane's capacity or number of pre-existing reservations before attempting to call the service.

The markup also tells the agent what it can infer about the state of the world after invocation of the web service (**result**). Service results are typically a set of conditional statements, principly based on the service outputs and what the agent believes about the state of the world. In this case, that knowledge is limited to what happens if the service is invoked successfully; if it's unsuccessful, the agent can infer nothing. The language for expressing results in LTML is a superset of what is defined for OWL-S. Note particularly the use of the **Thereis** and `referent` construct. The combination of these gives an "exists uniquely" kind of quantification — if the agent knows the patient that is the referent of `patientID`, it should bind `p` to that patient; otherwise, the agent can infer the existence of such a patient with that ID, and a skolem for that entity is created.

Method definitions Figure 3 shows an example of an LTML `Method`. This example was crafted so as to demonstrate most key elements of the process language. Methods and atomic processes have sets of **inputs** and **outputs**, which are process variables that bind to entities of a given type. All variables preceding a dash are typed with the class following the dash. The inputs `passenger` are the arguments to the method, and the outputs (`loc1, loc2, deptime` are the variables bound to whatever is produced. A method has a body which consists of a single control construct.

An LTML **Workflow** is simply a top-level method definition. It has no inputs or outputs, and serves the same function as the `main()` routine in a conventional programming language.

Methods have a **body** consisting of an instance of a control construct. The main compositional control constructs are **seq**, **loop** and **branch**. In the figure, it is a *sequence*(**seq**) of four steps (the last being a loop). Each control construct provides for the declaration of **links** which are essentially write-once local variables over the scope of the construct. Single step control constructs are **perform**, used to invoke an atomic services or another method by name, **achieve**, used to invoke a method indexed by its goal and **values**, used to bind values to links or method output parameters.

The initial keyword (shown in bold) for all control constructs is immediately followed by a *tag* which is simply a name for the instance of the construct. In translation to RDF, each bare tag is composed with a context marking string to make a unique URL. For example, `seq0` becomes `example3@FindFlight.seq0` and `step0` becomes `example3@FindFlight.seq0.step0`. Parameter and link variables are automatically translated by the same mechanism, so that subsequent references to steps or variables in markup intended to annotate or comment on those descriptions will be unambiguous.

```
(in-namespace example3)
(Method FindFlight
  (inputs passenger - base@Person
     loc1 loc2 - base@Location
     deptime - base@Time)
  (outputs foundFlt - travel@Flight)
  (body
    (seq seq0
       (links apt1 apt2 - travel@Airport
              flights - travel@FlightSet
              flight - travel@Flight)
       (acts
         (perform step0  ;; find a departure airport
             (airSvc@findAirport (loc <= loc1))
             (put (found => apt1)))
         (perform step1  ;; find an arrival airport
             (airSvc@findAirport (loc <= loc2))
             (put (found => apt2)))
         (perform step2 ;; find flights from apt1 to apt2
             (when (exp@and (bound apt1) (bound apt2))
             (airSvc@findFlights (origin <= apt1)
                           (destination <= apt2)
                           (onDay <= (exp@propval date depTime)))
             (put (fltList => flights)))
         (loop step3 ;; loop over flights - find one early enough
             (links (flt (over flights) - travel@Flight))
             (while (exp@not (exp@bound foundFlt)))
             (body
                (branch branch1
                   (test (base@time< (exp@propval departTime flt)
                                            deptime))
                       (values step4 (foundFlt := flt)))))))))))
```

Fig. 3. Sample method definition in LTML.

The ability to reference elements of methods uniquely so that one could *annotate* them with information useful during learning, and relate them to alternative hypotheses was a key objective of LTML.

Branch **test**, loop **while** or **until** and perform **when** clauses all take expression forms, which are conventional prefix-notation formulas. Expressions are reified in translation to RDF, and are composed of instantiated exp@Predicate subclasses. **and**, **or**, and **not** keywords may be used in the normal way in expressions. Links declared *in the same control construct as the expression* are treated as variables to be bound by the expression, which are treated as a queries against a Prolog-like model of the current state of the world that is maintained during execution. This essentially follows the approach adopted by OWL-S, and does not resolve the problem that class names and

properties frequently appear in such expressions as unary and binary predicates, though these expressions are not normal OWL descriptions.

Trace descriptions LTML also represents *traces*, which capture sequences of service activations. Traces are the primary inputs to the POIROT learning components. Trace elements include the service called, its inputs and outputs, and its effects, which is a grounded instantiation of the result description from each service called. Figure 4 shows an initial fragment of an example consistent with Figure 3 having been called. The syntax of traces is close to striped except that `input` and `output` keywords are followed by lists of parameter-value pairs rather than properties. If present, the result keyword is followed by a list of predicate terms, resulting from the interpretation of the service result.

```
(Trace x@Trace1
   (agent x@Demonstrator1)
   (elt
      (TraceElt x@Trace1Elt1
         (eltPosition 1)
         (timestamp "0:00:00")
         (eventType airSvc@findAirport)
         (input
           (airSvc@findAirport.loc
               (base@Loc x@Loc1
                   (base@locationName "Boston"))))
         (output
           (airSvc@findAirport.found
               (trvl@Airport x@Airport1
                   (trvl@airportLocID
                       (trvl@AirportLocID x@AirportLocID1
                           (base@value "BOS")))))
              (svc@outcomeFlag (svc@Success svc@Success1)))
         (serviceResult
            ((trvl@Airport x@Airport1)
             (trvl@airportLocID x@Airport1 x@AirportLocID1)
             (trvl@nearTo x@Loc1 x@Airport1))))
   . . .
```

Fig. 4. Sample web service definition in LTML

Figure 5 shows the scale of examples that POIROT learned as LTML method sets during the first year of the project. As of the end of the second year, this had grown considerably, and POIROT was able to execute examples that were hundreds of steps long.
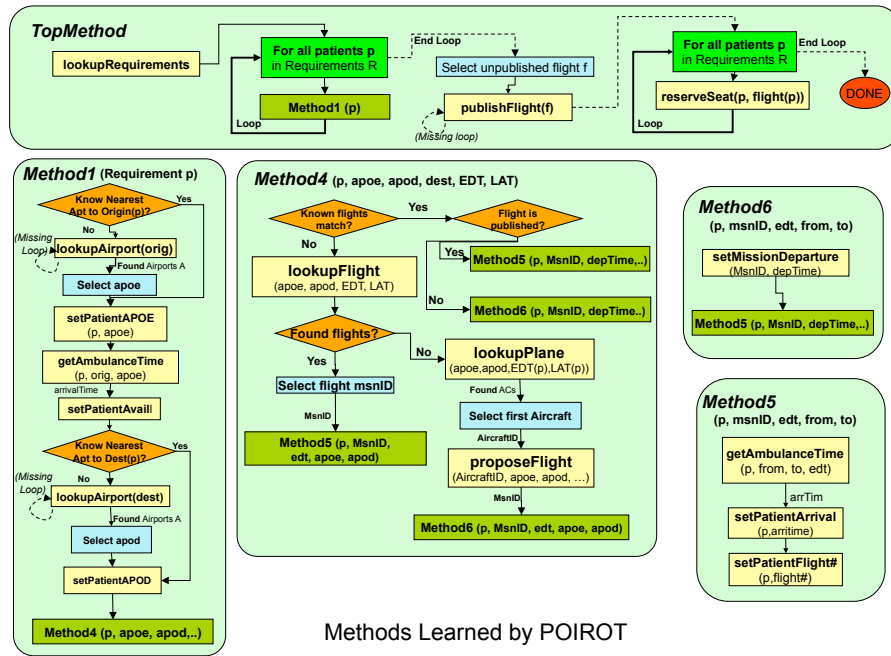
Fig. 5. Workflow Learned by POIROT using LTML

4 Execution Model

LTML provides an executable model of web-based workflow, and we have developed an LTML interpreter, SHOPPER[2], as one component of the POIROT system.

LTML is purely functional: its variables (referred to as "links" or "parameters") are write-once entities (except for loop links). LTML is unusual in featuring, in addition to conventional expressions (e.g., (islessthan *Number Number*)), logic-programming style queries. Any simple predication in a query context, for example (trans@scheduled patient flight), is treated as a boolean query against the agent's current beliefs. Queries are variable-binding operations if they refer to unbound links in the same LTML control construct. So if a branch declares links then its test may bind that link for use during the execution of the branch. Similarly, links declared in a perform may be bound by the when clause for use as inputs to the called action.

LTML has a state-based execution model like that assumed by AI planning systems [18]. Calls to web services return not only results (called outputs) in the usual sense of subroutine invocation, but also a *service result* or set of facts to be incorporated into the knowledge state. Subsequent queries are then based against the newly updated knowledge state, and built-in functions can also operate on this state.

5 Beyond OWL-S

LTML moves beyond OWL-S in features and expressivity in several ways. Most of these have resulted from our project's needs in representing complex workflows in a way that allowed them to be learned, annotated, compared, combined and executed.

A key feature of LTML models is their radical decomposability. The intent is that different authors (including programs) be able to combine their products into individual control structures. For example, one program might compose a loop that would process a collection, and a different program would dictate the order in which elements of the collection were processed. This required us to be careful about how tags were created to refer to these elements. We now expand tag names so that they are effectively scoped lexically.

The way `perform` steps bind their input and output parameters to and from *links* is different from OWL-S dataflow, where inputs were merely expressions on other step outputs. This construct makes it possible to follow the dataflow much like in a normal programming language. We introduced `when` conditions on steps (`performs`) that act as local step guards and bind step input parameters based on facts in the current state of the world. We also introdced the `values` construct to allow links to be bound to calculations based on other links. This is especially useful for drilling down into complex objects to find and bind values.

Though not discussed here for lack of space, we have also extended the process model to address HTN planning directly. `to-achieve` and `to-execute` forms associate methods with goals (with to-achieve) or abstract procedures (to-execute).

LTML *signals* support exception returns from service calls. This was still on the to-do list when OWL-S development ceased.

We have identified the need for the language to have atomic actions for *assert* and *query* to enable the workflows to record decisions and check for 'mental notes' that enable workflows to progress properly. These facts cannot be asserted by results of atomic service definitions if they are specific to the composite process and the atomic services are developed by other parties.

6 Conclusions and future work

LTML was developed in an attempt to make a more robust, usable web service procedure language using which we might even automatically learn these procedures, compare them, integrate them and critique them using automated learning and reasoning techniques. POIROT demonstrates the basic feasibility of this approach. LTML is continuing to evolve, although more slowly. We are moving toward applications of POIROT that will involve training naive users with procedures that POIROT has learned. This will likely require us to use partial order step constructs more extensively, so that following procedures during the training is not overly restrictive.

References

1. Burstein, M., Laddaga, R., McDonald, D., Cox, M., Benyo, B., Robertson, P., Hussain, T., Brinn, M., McDermott, D.: POIROT - integrated learning of web service procedures. In: Proceedings of the Twenty-Third AAAI Conference (AAAI-08), AAAI Press (July 2008)
2. Goldman, R.P., Maraist, J.: SHOPPER: interpreter for a high-level web services language. In: Proc. Int'l Lisp Conference. (March 2009)
3. Paolucci, M., Ankolekar, A., Srinivasan, N., Sycara, K.P.: The DAML-S virtual machine. In Fensel, D., Sycara, K.P., Mylopoulos, J., eds.: International Semantic Web Conference. Volume 2870 of Lecture Notes in Computer Science., Springer (2003) 290–305
4. Geib, C., Maraist, J., Goldman, R.P.: A new probabilistic plan recognition algorithm based on string rewriting. In: Proceedings of ICAPS-2008. (2008)
5. Cox, M.T., Burstein, M.H.: Case-based explanations and the integrated learning of demonstrations. Künstliche Intelligenz **22**(2) (2008) 35–38
6. Yaman, F., Oates, T., Burstein, M.: A context driven approach to workflow mining. In: Proceedings of IJCAI-09. (July 2009)
7. Winner, E., Veloso, M.: DISTILL: Learning domain-specific planners by example. In: Proceedings of ICML-2003. (August 2003)
8. Berners-Lee, T.: Primer: Getting into RDF & semantic web using N3. web page (August 2005)
9. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: OWL-S: Semantic markup for web services. W3C Member Submission (November 2004)
10. Martin, D., Burstein, M., McDermott, D., McIlraith, S., Paolucci, M., Sycara, K., McGuiness, D., Sirin, E., Srinivasan, N.: Bringing semantics to web services with owl-s. World Wide Web **10**(3) (2007) 243–277
11. McGuinness, D.L., van Harmelen, F.: Owl web ontology language overview. Technical report, W3C (February 2004) `http://www.w3.org/TR/owl-features/`.
12. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL). web page (March 2001)
13. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: Simple object access protocol (SOAP) 1.1. W3C Note (2000)
14. Clark, J., ed.: XSL transformations (XSLT). Technical report, W3C (November 1999)
15. McDermott, D.V.: Surface syntax for OWL-S. `http://www.daml.org/services/owl-s/1.2/owl-s-gram/owl-s-gram-htm.html` (2004)
16. McDermott, D.V.: The 1998 AI planning systems competition. AI Magazine **21**(2) (2000) 35–55
17. Nau, D., Muñoz-Avila, H., Cao, Y., Lotem, A., Mitchell, S.: Total-Order planning with partially ordered subtasks. In Nebel, B., ed.: Proceedings IJCAI, Morgan Kaufmann (2001) 425–430
18. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann (2004)