To Appear in *Proceedings of the 2009 International Lisp Conference*

SHOPPER: Interpreter for a High-level Web Services Language

Robert P. Goldman
SIFT, LLC
rpgoldman@sift.info

John Maraist
SIFT, LLC
jmaraist@sift.info

Categories and Subject Descriptors D [3]: 4

Keywords web service composition, interpreters, OWL, LTML, Lisp

1. Introduction

We have been working on the POIROT project[1] as part of DARPA's Integrated Learning program. POIROT is a large research program that aims to integrate multiple systems for learning workflows whose building blocks are semantic web services. These semantic web services are marked up in the Learnable Task Modeling Language (LTML). LTML (McDermott et al. 2008) is a radically composable web service markup and planning language with an s-expression syntax that extends both OWL-S (Martin et al. 2004), the W3C language for semantic web service markup, and the standard planning language PDDL (PDDL Resources). We have developed SHOPPER, a Common Lisp-based interpreter for the LTML language that is able to interpret unified LTML workflows, building and maintaining a complex state model based on information gathered from execution of semantic web services through the OWL Virtual Machine (OVM) (Paolucci et al. 2003).

2. POIROT Agent

SHOPPER is a part of the POIROT agent, whose structure is shown in Figure 1.[2] All POIROT components communicate through the system's blackboard, implemented on top of an RDF database. The learning components cooperate to develop parts of LTML workflows, based on training examples.Workflows are assembled out of these pieces by two additional POIROT components, the "Stitcher" and the SHOP2 (Nau et al. 2003) planner. When requested by the meta-control components of the architecture, these two POIROT components will assemble a workflow (which includes inlining all method invocations), and then dispatch it to SHOPPER for execution. SHOPPER will assemble an execution trace of its run and place it into the blackboard, allowing POIROT to further learn from experience. If SHOPPER encounters an error, it will write a description of the error into the blackboard in addition to the trace. It will also store its final state to permit restarting; it is intended that later work in POIROT feature replanning of workflows in response to errors and failures.

[1] Led by BBN Technologies, `bbn.com`.

[2] Although SHOPPER *can* run in isolation.

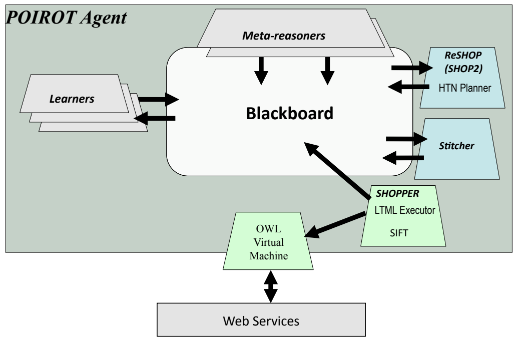

Figure 1. Simplified architecture of the POIROT agent.

3. LTML

The purpose of the LTML language is to be able to capture workflows whose components are semantic web services.[3] LTML extends the expressive power of OWL (van Harmelen and McGuinness 2004), allowing it to capture ontological information about classes and entities. LTML's primitive operations cover both conventional planning operators (as in PDDL), and semantic web services (as in OWL-S), with additional features to overcome shortcomings of OWL-S.

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

```
(AtomicProcess reserveSeat
  (inputs missionId - trans@ID
          patientID - med@PatientID
          specialNeeds - med@SpecialNeedsCode)
  (outputs outcomeFlag - res@ResultCode
           reservationID - med@ReservationID)
  (precondition
   (exists (m - trans@Mission
            capacity - xsd@integer
            num-passengers - xsd@integer)
           (and (trans@missionID m missionId)
                (trans@PAX m num-passengers)
                (trans@maxPAX m capacity)
                (islessthan num-passengers capacity))))
  (result
   (when (type outcomeFlag res@Success)
      (ThereIs ((p (referent med@Patient patientID))
                (m (referent trans@Mission missionID)))
                (trans@hasReservedSeat p m reservationID)
         (when (hasValue specialNeeds)
           (trans@supportsSpecialNeeds m specialNeeds))
         (increment-fluent (trans@PAX m) 1)))))
```

Figure 2. Sample web service markup in LTML.

Finally, LTML adds higher-level control flow constructs to compose these atomic actions into workflows.

A key feature of LTML models is their radical decomposability. 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. To support this, and provide for compatibility with the semantic web, LTML is defined as a superset of OWL and OWL-S, and the specification dictates how LTML constructs are to be represented as RDF triples.

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 another Lisp-like notation for RDF triples, similar to N3 (Berners-Lee 2005). All examples in this paper will be given in the surface notation.

LTML allows us to describe entity sorts and properties. For example:[4]

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

(Property exp@boundVar
   (domain exp@QuantFormula)
   (range ltml@Variable))
```

Note that the "`@`" here is a namespace separator. LTML namespace abbreviations map to XML namespaces.

LTML also supports semantic web service markup; see Figure 2. This defines what an invoking agent should provide to the web service (`inputs`), what the agent can expect back (`outputs`), the conditions under which the invocation should succeed (`precondition`). The markup also tells the agent what it can infer about the state of the world after invocation of the web service (`results`). 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 results provide a superset of OWL-S markup. Note particularly the use of the `Thereis` and `referent` construct. This 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.

[4] Examples taken from the LTML manual.

```
(Method Exp@Method0
  (inputs Exp@Method0.input0 - time@Date)
  (body
    (seq seq0
      (links Exp@Method0.link0 - med@SetOfPatientReqRecord)
      (acts
        (perform step0
          (luRqts@lookupRequirements
              (luRqts@ReqDate <= Exp@Method0.input0))
          (put (luRqts@lookupRequirementsOut
                              => Exp@Method0.link0))
          (occurence core@TraceElement1))
        (loop step2
          (links (looplink1 - med@PatientReqRecord))
          (body
            (perform step3
              (Exp@Method1
                (Exp@Method1.mInput0 <= looplink1)))))))))
```

Figure 3. Sample method definition

Finally, LTML permits us to compose the atomic processes into complex procedures (`Methods`) with conventional control structures like conditionals and loops. LTML is purely functional: its variables (referred to as "links") are write-once entities. LTML is unusual in featuring, in addition to conventional expressions (e.g., `(islessthan` *`Number Number`*`)`), logic-programming style queries. Any simple predication in program text, for example `(trans-@scheduled patient flight)`, is treated as a boolean query against the agent's beliefs. Queries can be extended to variable-binding operations using LTML constructs like `test`. A sample method definition is given as Figure 3. Note that all of the components are given names (e.g., the sequence construct is named `seq0`). This supports composability and translation into RDF triples (e.g., "the value of the `acts` property of `seq0` is...").

LTML has a state-based execution model like that assumed by AI planning systems (Ghallab et al. 2004). Calls to web services return not only results in the usual sense of subroutine invocation, but also a *service result* of facts to be incorporated into the knowledge state. Subsequent queries are then based against the new facts, and built-in functions can also operate on this state.

Figure 4 shows a small example of interaction with this knowledge state. In this example `noter` is a simulated web service whose sole effect is to add one fact `(noterCall`

```
(perform p1 (noter (arg <= 5)))
(perform p2
  (links x - xsd@anyType)
  (when (exp@and (noterCall 1 ?x) (< ?x 6)))
  (noter (arg <= 50))))))
```

Figure 4. Querying the knowledge state.

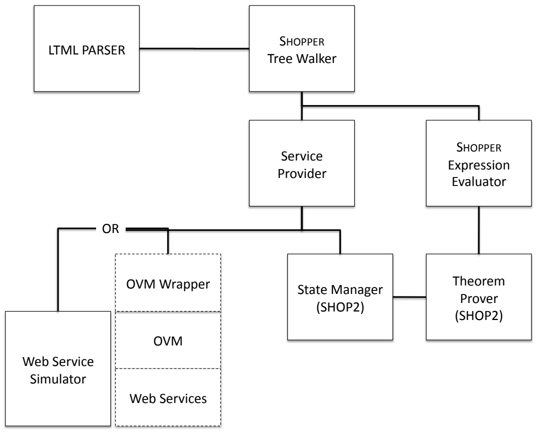

Figure 5. Architecture of the SHOPPER system.

$n\ z$) to the knowledge state, where z is its argument `arg`, and n is an integer which is 1 at the first invocation of `noter`, 2 at the second invocation, and so on. Of the two statements shown in the example, the first (`p1`) adds the fact `(noterCall 1 5)`.[5] Statement `p2` includes the guard that we must know of some x such that `(noterCall 1 ?x)` and `(< ?x 6)` hold. This precondition succeeds, and SHOPPER proceeds to invoke `(noter 50)`. In the final state, the system will know the two facts `(noterCall 1 5)` and `(noterCall 2 50)`.

4. SHOPPER Design

Figure 5 shows the SHOPPER architecture. SHOPPER receives its input by way of the LTML parser, based on CL-Yacc (Chroboczek 2005), which delivers to SHOPPER the abstract syntax tree (AST) of the LTML program, implemented as a network of CL structures.[6] The SHOPPER tree walker traverses the AST recursively, updating its execution state. SHOPPER's execution state has two components: a stack that contains link bindings, and a set of formulas that reflects the agent's beliefs about the world state.

We were able to build SHOPPER very quickly and efficiently by reusing components of the open source AI planning system SHOP2 (Nau et al. 2003).[7] SHOP2 contains a state manager component which maintains a trajectory of world states, supporting update and backtracking operations. SHOP2 also contains a theorem prover which allows Prolog-style querying of the world state (including the use of backward-chaining rules). We modified SHOP2 to make these free-standing ASDF systems.

When SHOPPER reaches a point in a program where it must invoke a semantic web service, it invokes the Service Provider. This is a shim layer that provides interfaces either to a web service simulator, written in Common Lisp, or to invocation of actual web services through the OVM. CL macros and CLOS were very helpful in building this intermediate layer, which is critical for SHOPPER development and testing. Use of the actual web services is undesirable for such purposes both because it is so heavyweight (invocation of a complex web service can take minutes, because of the layers of serializing, deserializing, and other overhead) and because the web services are backed by relational databases, which cannot be easily reconfigured for testing.

Invocation of web services can cause updates to SHOPPER's beliefs about the world, which are transmitted by the Service Provider to the State Manager. Evaluation of expressions is done by querying the world state through SHOPPER's expression evaluator. The built-in functions of LTML are encoded as special-purpose functions, which may be backed by inference rules in the theorem-prover.

5. Aspects of SHOPPER

The environment into which we deployed SHOPPER brought requirements for which Lisp's unique features were particularly useful. The LTML definition continued to evolve through the interpreter's development; developers not actively involved in the interpreter internals required low-level access to the implementation of built-in functions for experimentation and debugging; the interpreter was required in several contexts, including both online and offline execution of code referencing web services. Lisp's macros, sophisticated error handling, and CLOS each gave clear advantages.

We used macros in three distinct major roles in SHOPPER. Interestingly, we did *not* use macros in parsing: the syntax of LTML was not sufficiently stable in the parser development cycle. Since LTML syntax was s-expression based, we originally intended to simply use macros to analyze the language. However, the language syntax changed so frequently during development, and we found those syntactic changes so damaging to the stability of the code that we replaced our macros with a LL grammar of LTML, and used CL-Yacc (Chroboczek 2005) to parse LTML into an AST.

Macros provide linguistic support for ensuring complete case coverage in generic functions dispatching on AST objects. Rather than using a sequence of `defmethods`, all of the generic's primary methods are defined in a single macro invocation which also checks that no cases are inadvertently

[5] Assuming that these statements are the first uses of the service.

[6] In Section 5 we explain why we use a conventional parser in SHOPPER.

[7] SHOP2 was originally developed at the University of Maryland, and is now jointly maintained by UMd and SIFT.

omitted.[8] This check, familiar to users of algebraic data types in languages such as ML or Haskell, gave us compile-time error reporting for case omissions arising from changes as the language definition evolved. Our existing macro's applicability relies on two factors: a flat class hierarchy, and functions which are essentially single dispatch, or at least, multiply dispatched uniformly for each method. Where a flat class hierarchy is not possible, a macro for each level of decomposition of class into subclasses is a possibility.

Our second major use of macros in the interpreter is for defining offline versions or small test cases of web services. Web services differ from function libraries in ways which make their definitions rather tedious. They may be discovered only at runtime, so their names, parameter names, and properties cannot be processed at compile-time, but rather must be stored in runtime data structures. Although we do use `eql`-dispatch to hide many of these details, we nonetheless must still provide a number of different methods for each service definition. A service-defining macro encapsulates these coordinated definitions, eliminating confusing errors from omitting one of a family of related `defmethods`.

Finally, we use a macro to enumerate the built-in functions of LTML. LTML built-ins differ from the core functions of most languages because LTML expressions, by default, are queries to a belief state — the built-in functions are exceptions to this default. In SHOPPER we implement this behavior by associating each built-in with a number of methods, not only for executing a call to the built-in but also for determining that the name does refer to a provided function, and should not be handled by belief state query. As with web services, these multiple coordinated definitions are best generated from a single macro call.

Because the overall system studies workflow learning, failure is a phenomenon of interest: the failure of a workflow must trigger additional learning effort, and hopefully the generation of a updated workflow to repair damage arising from the failure, and then achieve the original goal. To provide information about workflow failure, and to allow an updated workflow to resume from the same dynamic environment where its predecessor failed, we rely on Lisp's robust error handlers. Each (recursive) invocation of the main tree-walking interpreter introduces a `handler-bind` wrapper which enriches errors objects with their context. Optionally, the innermost wrapper may allow a `break`, when the interactive user needs a more detailed look at the state of the interpreter itself. But in the normal phase of execution, the outermost error handler places this structured pointer into the LTML program, plus the lexical bindings at the time of failure, on the blackboard for examination by the learners.

Finally, we take advantage of multiple inheritance for defining aspects of system behavior. Many of SHOPPER's operations receive a parameter used for dispatch based on the overall system configuration — whether we seek to make external web service calls or rely on offline coded versions, which set of web service we address. This dispatch object allows behavior to be cleanly customized via key generic functions of the interpreter: the functions can be wrapped, given a `:before` or `:after` method, or simply overridden.

[8] Exceptions are permitted, but must be explicitly flagged.

A. Implementation Notes

SHOPPER is written in CL, but currently runs only under Allegro CL. LTML is case-sensitive and the current LTML libraries only run under Allegro's "modern," case-sensitive CL dialect. Furthermore, for compatibility with POIROT, SHOPPER relies on Allegro's proprietary Java interface.

Acknowledgments

This work was supported by DARPA IPTO under contract FA8650-06-C-7606. Approved for Public Release, Distribution Unlimited.

References

Tim Berners-Lee. Primer: Getting into RDF & semantic web using N3. web page, August 2005. URL `http://www.w3.org/2000/10/swap/Primer.html`.

Juliusz Chroboczek. CL-Yacc homepage. web page, 2005. URL `http://www.pps.jussieu.fr/~jch/software/cl-yacc/`.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, Inc., 2004.

David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic markup for web services. W3C Member Submission, November 2004. URL `http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/`.

Drew V. McDermott, Mark Burstein, Michael Cox, Robert P. Goldman, and David McDonald. The LTML manual pages. Unpublished manual, July 2008.

Dana Nau, T. C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003. ISSN 1076-9757.

Massimo Paolucci, Anupriya Ankolekar, Naveen Srinivasan, and Katia P. Sycara. The DAML-S virtual machine. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003. ISBN 3-540-20362-1.

PDDL Resources. PDDL resources. web page, April 2008. URL `http://ipc.informatik.uni-freiburg.de/PddlResources`.

Frank van Harmelen and Deborah L. McGuinness. OWL web ontology language overview. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.