# Shopper: a system for executing and simulating expressive plans

**Robert P. Goldman** and **John Maraist**
{rpgoldman, jmaraist}@sift.info
SIFT, LLC
211 N. First St.
Minneapolis, MN 55401 USA

## Abstract

We present Shopper, a plan execution engine that facilitates experimental evaluation of plans and makes it easier for planning researchers to incorporate replanning. Shopper interprets the LTML plan language, which extends PDDL in two major ways: with more expressive control structures, and with support for semantic web services modeled on OWL-S. LTML's command structures include not only conventional ones such as branching, iteration, and procedure calls, but also features needed to handle HTN plans, such as precondition-filtered method choice. Unlike conventional programming languages, LTML supports interaction with the agent's belief store, so that its execution semantics line up with those assumed by planners. LTML actions extend PDDL actions in having outputs as well as effects, which means that they can support actions that sense the world; an important special case of this is semantic web services, which reveal information about a state hidden from the agent. To support experimentation as well as action in the real world, Shopper accommodates multiple, swappable implementations of its primitive action API. For example, one may interact with real web services through SOAP and WSDL, or with simulated web services through local procedure calls. We describe novel features of LTML, the interpretation strategy, swappable back-ends, and the implementation.

## Introduction

Planning techniques are evolving in scale and practicality to be able to tackle real-world problems. In order for the techniques to be broadly accepted, the community must evaluate not only plan generation, but also how generated plans will perform in execution. We have developed the Shopper system to interpret plans encoded in LTML, an extension of PDDL. Shopper is designed to provide support to planning researchers, providing error detection, signaling and packaging, to make it easier to develop and test replanning techniques. Shopper provides swappable interfaces to execution and simulation backends for experimentation. For example, one may interact with real web services through SOAP and WSDL (Englander 2002), or with simulated web services through local procedure calls.

We have used LTML and Shopper for planning and execution problems that correspond to *web service composition* (Sirin et al. 2004; Traverso and Pistore 2004), and we draw this paper's examples from that domain. Our framework is generally suitable for software application domains like the Unix softbot (Etzioni 1993) and for limited cases of planning for physical systems as well.



```
(Class trans@Airport
   (subClassOf loc@Location)
  (restrict loc@airportLocationID
     (cardinality 1)
       - loc@AirportLocationCode))
```

Figure 1: LTML class definition.

## LTML

The Learnable Task Modeling Language, or LTML (Burstein et al. 2009), is a plan language that provides additional features necessary for real-world planning — especially in the domains of web services composition and softbots — in a framework comfortable to planning researchers familiar with PDDL. LTML is an extension to PDDL that brings in (1) compatibility with OWL (the web ontology language) and OWL-S to support semantic web service composition, and (2) variables[1] and (3) more expressive control constructs (branches, looping). 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. Finally, LTML adds higher-level control flow constructs to compose these atomic actions into programs.[2] LTML has some other unusual features for serialization on the semantic web and support for multiagent learning which we will not discuss here.

LTML allows us to describe the types and properties of entities. For example, the definition in Figure 1 specifies that an airport is a kind of location, and that it has an air-

[1] PDDL has parameters, and quantified variables in actions, but no program-style variables.

[2] Often referred to as "workflows" in the semantic web domain.

```
(AtomicProcess reserveSeat
  (inputs missionId - trans@ID
          patientID - med@PatientID
          specialNeeds - med@SpecialNeedsCode)
 (outputs outcomeFlag - res@ResultCode
           reservationID - med@ReservationID)
  (precondition
   (exists (m - trans@Mission)
           (trans@missionID m missionId)))
 (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.

port location ID property that has exactly one value of type airport location code. The "@" here is a namespace separator;[3] the airport location code class is defined in the `loc` namespace.

A primitive action in LTML is referred to as an `AtomicProcess`.[4] LTML atomic processes are an extension of PDDL actions that supports semantic web service markup (see Figure 2 for an example). Like PDDL actions, LTML atomic processes have parameters (`inputs`) and preconditions. Unlike PDDL actions, LTML atomic processes also have `outputs`. These are values that will be returned to the agent, typically a reflection of the internal state of some entity or process that is not accessible to the agent (e.g. an online bookstore's inventory).

The LTML `result` component corresponds to PDDL's effects. However, when working with domains like web services, the effects specification fulfills an important additional function — it tells the agent what it can infer about the state of the world based on its prior knowledge of the world *and the outputs of the web services*. The results provide a superset of OWL-S markup and PDDL effects. Note the use of the `Thereis` and `referent` construct. They give an "exists uniquely" quantifier — 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.[5] In Figure 2, the service allows us to reserve a seat for a patient on a mission. If the service call succeeds, the results tell us first that there is a patient, `p`, who is the referent of the input patient identifier, and a mission, `m`, the referent of the input mission id. We can infer that `p` has a reserved seat on `m`, with a particular reservation ID, and we should increment the number of passengers (`trans@PAX`).

LTML permits us to compose the atomic processes into complex procedures (`Methods`) with conventional control structures like conditionals and loops. LTML is

[3]LTML namespaces are XML namespaces.

[4]LTML processes are not processes in the sense of temporally-extended periods during which variables change continuously.

[5]This form of quantification is analogous to that of Golden's run-time variables, in his PUCCINI planner (Golden 1997).

```
(Method Exp@TransportAllPatients
 (inputs date - time@Date)
 (body
  (seq seq0
    (links recordSet
           - med@SetOfPatientReqRecord)
    (acts
      (perform step1
        (luRqts@lookupRequirements
           (luRqts@ReqDate <= date))
        (put
          (luRqts@lookupRequirementsOut
            => recordSet)))
       (loop step2
        (links
          (thisRecord (over recordSet)
            - med@PatientReqRecord))
        (body
          (perform step3
            (Exp@TransportPatients
              (record <= thisRecord)))))))))
```

Figure 3: Sample LTML method definition.

```
(perform p1
  (links x - xsd@integer m - trans@Mission)
   (when (exp@and (trans@missionID mid m)
                  (trans@PAX m x)))
   (notifyFlightComplement
      (missionID <= mid)
      (PAXCount <= x)))
```

Figure 4: Querying the knowledge state.

purely functional: its variables (referred to as "links") are lexically-scoped, write-once entities. LTML features logic-programming style queries in addition to conventional expressions. Predications such as `(trans@scheduled patient flight)` are treated as queries against the agent's beliefs in constructs like `test`.

The sample method definition in Figure 3 describes a part of a plan to process patients in a medical management system. The method is a sequence whose first step is to get a set of patient requirement records by `perform`ing a `lookupRequirements` web service for the current `date`. The input date is supplied to the named parameter of the service, and the (named) output parameter's value is `put` to the variable `recordSet`. Then we loop over the set of records, passing each one in turn to the `Exp@TransportPatients` method.

Figure 4 shows a small example of how an LTML method call interacts with the knowledge state. In this example, we assume that we have bound the link `mid` to a mission identifier. The `perform` statement `p1` establishes two local variables, `x` and `m`, and queries the agent's belief state to bind `m` to the mission with the identifier `mid`, and `x` to the number of passengers on the flight. We next invoke the web service `notifyFlightComplement`, passing as arguments the mission identifier and the number of passengers.

One important divergence between LTML and PDDL is

that the LTML semantics, unlike PDDL semantics, is not based on a finite, known domain of quantification. In the domains of web services and software, typically the set of objects is not known and fixed in advance (e.g. the agent will not know all possible ISBN numbers). Furthermore, in many domains entities will be created during the course of execution (e.g. a new "flight reservation" might be created).

## Shopper

Shopper is an interpreter for the executable subset of LTML. In a fully implemented scenario, a controlling executive process invokes Shopper with the LTML plan generated by the planner component(s). Shopper executes the plan, accumulating and eventually returning a list of event records. Typically the event records are details of the actions taken by the plan, but additional detail may be required in particular situations, for example when debugging interactively.

The executable fragment of LTML is essentially a standard, non-nested procedural language with single-assignment variables, so Shopper implements a straightforward stack-based abstract machine. There is an aspect of logic programming in that a query to the knowledge state can bind variables, but since these variables are not subject to revision through backtracking, they can be stored on the stack just as variables bound by explicit assignments. The choice of single-assignment variable semantics does require some additional complexity for loops: for iteration over sequences of values, and for accumulating result values as combinations of values from different iterations. Nonetheless these complications are entirely compatible with the standard stack model.

Shopper's LTML interpreter is engineered to be agnostic about much of its environment,[6] and requires very little besides the use of LTML itself. In particular, Shopper places no requirements on the sort of planner (if any) which generates its LTML plan: the plan could be completely linear, as from a conventional IPC planner, or could be the output of an HTN planner. Shopper further makes no assumption of any particular set of actions. As we discuss in the sections below, this agnosticism is key to Shopper's usefulness as an experimental platform, but it also requires consideration of the possible arbitrary failures within these extensions.

A key part of the implementation strategy is to separate out an API for implementing atomic process invocations, to permit these backends to be changed. The key parts of the backend are the mapping from atomic process names to LTML markup, and the implementation of atomic processes that applies inputs and reads outputs. For semantic web services, the backend must include handling the grounding of semantic objects into conventional inputs (e.g. the translation of an LTML `transairport` object into a string by extraction of its ICAO airport code) and the inverse lifting of outputs into semantic objects (e.g. recognition that a four-letter string in the output is the ICAO code corresponding to a `trans@airportID` object). Currently, Shopper supports two back-ends to semantic web services through alternate APIs to SOAP and WSDL, a number of very simple backends for testing (e.g. one where the services perform simple arithmetic operations), and a simulated web service backend for experimentation (see below).

[6] We presented the software engineering of Shopper, with a focus on the advantages gained by our choice of Lisp as an implementation language, in a previous paper (2009).

## Replanning and exceptions

A key part of any real-world plan execution is reacting to unexpected events and replanning to handle them. From the point of view of a planning researcher, a substantial advantage of Shopper is that it lifts the burden of error detection and handling, and leaves the researcher free to concentrate on the replanning research. Shopper was designed to interact smoothly with a planning system when a plan fails. These failures can arise for several reasons: failure to meet an action's preconditions; an exception raised by the plan itself; the arbitary errors which may be thrown by external action implementations; or program errors arising from flaws in the plan itself, such as a violation of the single-assignment semantics. Shopper catches errors thrown during workflow and web service execution, and returns serializations of the error itself, of the dynamic state of execution, and of the knowledge state. Of course, Shopper's ability to catch and serialize errors will not bring all errors into the scope of any particular replanner; planning may still fail.

```
proc solveIt
  state := initState;
  loop
     let prog = plan(state, goal);
     unless prog return false;
     let <success, newState> =
           shopper(prog, state)
     if success return true;
     else state := newState;
```

Figure 5: Simple replanning loop.

The pseudo-code in Figure 5 shows how easy Shopper makes replanning. The return of the knowledge state from Shopper is the critical component for replanning. It provides support for the most basic replanning tactic: determining the current world state and replanning *ab initio*. Additional support may be supplied in the exception object Shopper returns for planners that can exploit it. An HTN-style replanner (e.g. HotRIDE (Ayan et al. 2007)) might need to know something about the state of the program when the plan failed, and a more domain-specific replanning strategy would want to extract from the exceptions information about the specifics of action failures. If the planning system is able to replan, Shopper can resume execution of a residual plan, calculated to achieve overall goals from the context of the partial execution of the original plan.

The ability of the planner to generate a plan featuring exception signals is very important to supporting replanning. In some cases it will be impossible or impractical for a planner to generate a complete program (policy) to solve a problem. In such cases, the planner can write into a generated plan any "simplifying assumptions" it makes, such as that a

given web service will be able to provide an acceptable answer. The planner can do this by writing into the plan a conditional branch that checks the acceptability of this answer, and raises an exception if the answer is unacceptable. On this exception the planner can then update the plan, incorporating the additional information arising from the exception and the particular execution context.

## Facilities for experimentation

There are a number of practical challenges to experimenting with planning in the context of web services. First, one may not control the real web services, meaning that it is hard to vary them for testing and research purposes. Even if a researcher does control bona fide web services, these are often very difficult to manipulate experimentally — a web service of the kind that interests us is often a front end to a relational database of some sort, and it is laborious to manipulate the contents of these relational databases for experimental purposes. It is especially difficult to do so when multiple databases (in multiple web services) must be coordinated to create a consistent experimental scenario. We would often like to be able to inject failure cases, or stochastic variations.

Our experience with such services has led us to augment Shopper with a specialized simulation capability. Shopper provides a mechanism for associating a simulated implementation with a set of LTML `AtomicProcesses`. The simulated implementation is built on the state model and theorem-prover of the SHOP2 planner.[7] The simulator permits a logic-programming style of specification for the web services, providing a very terse and elegant specification, and allowing for quick loading of different starting states. Unlike conventional logic-programming systems, and because of its planner lineage, our simulator provides first-class support for state transitions. The Horn clauses also provide a convenient mechanism for stochastic variation.

The simulation capability uses Shopper's swappable backends so that one can interact with simulated web services using exactly the same plans as with real ones.

## Conclusions

Shopper lowers the barriers for researchers wishing to experiment with replanning and planning in software domains. It does so by supporting a language, LTML, whose expressive power has been expanded to meet the needs of such domains. Because of LTML's expressive power, Shopper can execute linear or HTN plans.[8] We also support replanning research and systematic experimentation by easing the implementation requirements for replanning and providing facilities to support systematic experimentation in simulated domains.

Our work on Shopper is ongoing; we are addressing issues in incomplete information, partial ordering, and the practicalities of debugging LTML plans. Currently, Shopper assumes that it has complete knowledge of propositions in its queries, and (unsafely) makes the closed-world assumption. We have been working on a design for better handling partial knowledge, based on the local closed world framework of Etzioni et al. (1997), on which we are building an approach for open world planning (Goldman 2009). Currently Shopper makes a STRIPS-like assumption when projecting the effects of actions. This is problematic when combined with the consistency constraints imposed by an OWL-style description logic ontology; it can be difficult to capture all the implied ramifications in PDDL-style effects expressions. Hoffman et al. (2009) discuss this in their work on web services composition. As this paper was written, we were working to develop a graphical debugger for Shopper.

[7] We have modified SHOP2 to make it possible to use its theorem-prover and state data structures as a stand-alone library.

[8] LTML can capture partially-ordered plans, but Shopper cannot yet execute them.

## Acknowledgments

This article was supported by DARPA/IPTO and the Air Force Research Labratory, Wright Labs under contract number FA8650-06-C-7606. This paper does not represent the official position or opinions of DARPA/IPTO or Air Force Research Labratory, Wright Labs. Thanks to colleagues at BBN, especially Mark Burstein, David McDonald and Jacob Beal, and at the University of Maryland, especially Dana Nau and Ugur Kuter. Thanks to our anonymous reviewers.

## References

Ayan, F.; Kuter, U.; Yaman, F.; and Goldman, R. P. 2007. HOTRiDE: Hierarchical ordered task replanning in dynamic environments. In *Proceedings of the ICAPS Workshop on Planning and Plan Execution for Real-World Systems*.

Burstein, M.; Goldman, R. P.; McDermott, D. V.; McDonald, D.; Beal, J.; and Maraist, J. 2009. LTML — a language for representing semantic web service workflow procedures. In *Proceedings ISWC workshop on Semantics for the Rest of Us*.

Englander, R. 2002. *Java and SOAP*. O'Reilly.

Etzioni, O.; Golden, K.; and Weld, D. S. 1997. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence* 89(1–2):113–148.

Etzioni, O. 1993. Intelligence without robots: A reply to Brooks. *AI Magazine* 14(4):7–13.

Golden, K. 1997. *Planning and knowledge representation for softbots*. Ph.D. Dissertation, University of Washington.

Goldman, R. P., and Maraist, J. 2009. SHOPPER: interpreter for a high-level web services language. In *Proc. Int'l Lisp Conference*.

Goldman, R. P. 2009. Partial observability, quantification, and iteration for planning: Work in progress. In *Proceedings ICAPS workshop on Generalized planning*.

Hoffmann, J.; Bertoli, P.; Helmert, M.; and Pistore, M. 2009. Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection. *Journal of Artificial Intelligence Research* 35:49–117.

Sirin, E.; Parsia, B.; Wu, D.; Hendler, J.; and Nau, D. 2004. HTN planning for web service composition using SHOP2. *Journal of Web Semantics* 1(4):377–396.

Traverso, P., and Pistore, M. 2004. Automated composition of semantic web services into executable processes. In *ISWC*.

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