Adapting Golog for Composition of Semantic Web Services

Sheila McIlraith
Knowledge Systems Laboratory
Department of Computer Science
Stanford University
Stanford, CA 94305-9020
sam@ksl.stanford.edu

Tran Cao Son
Computer Science Department
New Mexico State University
PO Box 30001, MSC CS
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

Abstract

Motivated by the problem of automatically composing network accessible services, such as those on the World Wide Web, this paper proposes an approach to building agent technology based on the notion of generic procedures and customizing user constraint. We argue that an augmented version of the logic programming language Golog provides a natural formalism for automatically composing services on the Semantic Web. To this end, we adapt and extend the Golog language to enable programs that are generic, customizable and usable in the context of the Web. Further, we propose logical criteria for these generic procedures that define when they are *knowledge self-sufficient* and *physically self-sufficient*. To support information gathering combined with search, we propose a middle-ground Golog interpreter that operates under an assumption of reasonable persistence of certain information. These contributions are realized in our augmentation of a ConGolog interpreter that combines online execution of information-providing Web services with offline simulation of world-altering Web services, to determine a sequence of Web Services for subsequent execution. Our implemented system is currently interacting with services on the Web.

1 INTRODUCTION

Two important trends are emerging in the World Wide Web (WWW). The first is the proliferation of so-called *Web Services* – self-contained, Web-accessible programs and devices. Familiar examples of Web services include information-gathering services such as the map service at yahoo.com, and world-altering services such as the book-buying service at amazon.com. The second WWW trend is the emergence of the so-called *Semantic Web*. In contrast to today's Web, which is designed primarily for human interpretation and use, the Semantic Web is a vision for a future Web that is unambiguously computer-interpretable [2]. This will be realized by marking up Web content, its properties, and its relations, in a reasonably expressive markup language with a well-defined semantics. DAML+OIL, a description-logic based Semantic Web markup language, is one such language [10, 11].

Our interest is in the confluence of Web Services and the Semantic Web. In early work, we outlined a semantic markup for describing the capabilities of Web services, initially in first-order logic and a predecessor to DAML+OIL [19]. This effort evolved into a coalition of researchers from BBN, CMU, Nokia, SRI, Stanford and Yale who are developing a DAML+OIL ontology for Web services, called DAML-S [4]. Several metaphors have proved useful in developing this markup, including viewing Web services as functions with inputs and outputs, and alternatively as primitive and complex actions with (knowledge) preconditions and (knowledge) effects. While DAML-S is not yet complete, two versions of the ontology have been released for public scrutiny [3]. We will return to DAML-S towards the end of the paper.

The provision of, effectively, a knowledge representation of the properties and capabilities of Web services enables the automation of many tasks, as outlined in [19]. In this paper we focus on the task of automated Web service composition (WSC): Given a set of Web services and a description of some task or goal to be achieved (e.g., "Make the travel arrangements for my KR2002 conference trip."), find a composition of services that achieves the task. Disregarding network issues, WSC can be conceived as either a software synthesis problem, or as a planning and plan execution problem, depending upon how we represent our services. In either case, this application domain has many distinctive features that require and support tailoring. We identify and address many of these features in this paper.

In this paper we conceive WSC as a planning and execution task, where the actions (services) may be complex actions. In related work, we show how to compile service representations into operators that embody all the possible evolutions of a complex action, in order to treat complex actions as primitive action plan operators [18]. As a planning task, WSC is distinguished in that it is planning with very incomplete information. Several sequenced information-gathering services may be required, that culminate in the execution of only a few world-altering services. (Imagine making your travel plans on the Web.) Since our actions (services) are software programs, the input and output parameters of the program act as knowledge preconditions and knowledge effects in a planning context. Software programs can also have side-effects in the world (such as the purchase of a commodity), that are modeled as non-knowledge effects. Service preconditions are regularly limited to knowledge preconditions. Information-gathering services (aka sensors) don't fail, network issues aside. Exogenous events affect the things being sensed. Persistence of knowledge has a temporal extent associated with it (contrast stock prices to the price of a shirt at the Gap), which affects the sequencing of services. Services often provide multiple outputs, a subset of which must be selected to act as input for a subsequent service (consider picking flights).

Many services perform similar functions, so WSC must choose between several services, each sharing some of the same effects. Also, plans (compositions of services) are often short, so the plan search space is short and broad. WSC tasks may or may not be described in terms of a goal state. In some instances they are described as a set of loosely coupled goals, or constraints. Many plans may satisfy the WSC task. User input and user constraints are key in pruning the space of plans (e.g., choosing from the multitude of available flights) and in distinguishing desirable plans.

The unique features of WSC serve to drive the work presented in this paper. Rather than realizing WSC simply as planning, we argue that a number of the activities a user may wish to perform on the (semantic) WWW or within some networked service environment, can be viewed as customizations of reusable, high-level generic procedures. For example, we all use approximately the same generic procedure to make our travel plans, and this procedure is easily described. Nevertheless, it is difficult to task another person, less a computer, to make your travel plans for you. The problem lies not in the complexity of the procedure, but rather in selecting services and options that meet your individual constraints and preferences. Our vision is to construct reusable, high-level generic procedures, and to archive them in sharable (DAML-S) generic-procedures ontologies so that multiple users can access them. A user could then select a task-specific generic procedure from the ontology and submit it to their agent for execution. The agent would automatically customize the procedure with respect to the user's personal or group-inherited constraints, the current state of the world, and available services, to generate and execute a sequence of requests to Web services to perform the task.

We realize this vision by adapting and extending the logic programming language Golog (e.g., [14, 22, 6]). The adaptations and extensions described in the sections to follow are designed to address the following desiderata of our WSC task. **Generic:** We want to build a class of programs that are sufficiently generic to meet the needs of a variety of different users. Thus programs will often have a high degree of nondeterminism to embody the variability desired by different users. **Customizable:** We want our programs to be easily customizable by individual users. **Usable:** We want our programs to be usable by different agents with different a priori knowledge. As a consequence, we need to ensure that the program accesses all the knowledge it needs, or that certain knowledge is stipulated as a prerequisite to executing the program. Similarly, the program ensures the actions it might use are $Poss$-ible. The programs must be both knowledge and physically self-sufficient.

2 ADAPTING GOLOG

Golog (e.g., [14, 8, 6, 22]) is a high-level logic programming language, developed at the University of Toronto, for the specification and execution of complex actions in dynamical domains. It is built on top of the situation calculus (e.g.,[22]), a first-order logical language for reasoning about action and change. Golog was originally developed to operate without considering sensing (aka information-gathering) actions. For Web applications, we rely on a version of Golog built on top of the situation calculus with knowledge and sensing actions (e.g., [24, 22]), which we henceforth refer to as simply as the situation calculus.

In the situation calculus [6, 22], the state of the world is described by functions and relations (fluents) relativized to a situation s, e.g., $f(\vec{x}, s)$. To deal with sensing actions, a special knowledge fluent K, whose first argument is also a situation, is introduced. Informally, $K(s', s)$ holds if the agent is in the situation s but believes (s)he might be in s'. The function $do(a, s)$ maps a situation s and an action a into a new situation. A situation s is simply a history of the primitive actions performed from an initial, distinguished situation S_0. A situation calculus theory $\mathcal{D}$ comprises the following sets of axioms (See [22] for details.):

- domain-independent foundational axioms of the situation calculus, Σ;
- accessibility axioms for K, $\mathcal{K}_{init}$[1];

[1] At this stage, we do not impose any conditions on K such as

- successor state axioms, $\mathcal{D}_{SS}$, one for K and one for every domain fluent F;
- action precondition axioms, $\mathcal{D}_{ap}$, one for every action a in the domain, that serve to define $Poss(a,s)$;
- axioms describing the initial situation, $\mathcal{D}_{S_0}$ (including axioms about K);
- unique names axioms for actions, $\mathcal{D}_{una}$;
- domain closure axioms for actions, $\mathcal{D}_{dca}$[2].

Golog builds on top of the situation calculus by providing a set of extralogical constructs for assembling primitive actions, defined in the situation calculus, into complex actions that collectively comprise a program, δ. Constructs include the following.

a — primitive actions
$\delta_1;\delta_2$ — sequences
$\phi?$ — tests
$\delta_1 \vert \delta_2$ — nondeterministic choice of actions
$(\pi x)\delta(x)$ — nondeterministic choice of arguments
δ^* — nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf** – conditionals
while ϕ **do** δ **endWhile** — while loops

Note that the conditional and while-loop constructs are actually defined in terms of other constructs.

if ϕ **then** δ_1**else** δ_2**endIf** $\doteq [\phi?;\delta_1] \mid [\neg\phi?;\delta_2]$
while ϕ **do** δ **endWhile** $\doteq [\phi?;\delta]^*;\neg\phi?$

These constructs can be used to write programs in the language of a domain theory, e.g.,
$buyAirTicket(\vec{x})$;
if far **then** $rentCar(\vec{y})$ **else** $bookTaxi(\vec{y})$ **endIf**.

Given a domain theory, $\mathcal{D}$ and Golog program δ, program execution must find a sequence of actions $\vec{a}$ such that: $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$. $Do(\delta, S_0, do(\vec{a}, S_0))$ denotes that the Golog program δ, starting execution in S_0 will legally terminate in situation $do(\vec{a}, S_0)$, where $do(\vec{a}, S_0)$ abbreviates $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$.

2.1 CUSTOMIZING GOLOG PROGRAMS

In this section we extend Golog to enable individuals to customize a Golog program by specifying personal constraints. To this end, we introduce a new distinguished fluent in the situation calculus called $Desirable(a,s)$, i.e., action a is desirable in situation s. We contrast this with $Poss(a,s)$, i.e. action a is physically possible in situation s. We further restrict the cases in which an action is executable by requiring not only that an action a is $Poss(a,s)$ but further that it is $Desirable(a,s)$. This further constrains the search space for actions when realizing a Golog program. The set of $Desirable$ fluents, one for each action, is referred to as $\mathcal{D}_D$. $Desirable(a,s) \equiv true$ unless otherwise noted.

An individual specifies her personal constraints in our Semantic Web markup language. The constraints are expressed in the situation calculus as *necessary conditions for an action a to be desirable*, $\mathcal{D}_{necD}$ of the form:

$$Desirable(a,s) \supset \omega_i, \tag{1}$$

and *personal constraints*, $\mathcal{D}_{PC}$ which are situation calculus formulae, C.

For example, Marielle would like to buy an airline ticket from origin o to destination d, if the driving time between these two locations is greater than 3 hours. Thus $Desirable(buyAirTicket(o,d,dt),s) \supset gt(DriveTime(o,d),3,s)$ is included in $\mathcal{D}_{necD}$. Similarly, Marielle has specified dates she must be at home and her constraint is not to be away on those dates. Thus, $\mathcal{D}_{PC}$ includes: $\neg(Away(dt,s) \wedge MustbeHome(dt,s))$. Using $\mathcal{D}_{necD}$ and $\mathcal{D}_{PC}$, and exploiting our successor state axioms and domain closure axioms for actions, $\mathcal{D}_{SS}$ and $\mathcal{D}_{dca}$, we define $Desirable(a,s)$ for every action a as follows:

$$Desirable(A(\vec{x}),s) \equiv \Omega_A \wedge \bigwedge_{C \in \mathcal{D}_{PC}} \Omega_{PC}, \tag{2}$$

where $\Omega_A = \omega_1 \vee \ldots \vee \omega_n$, for each ω_i of (1). E.g.,
$\Omega_{buyAirTicket} = gt(DriveTime(o,d),3,s)$, and

$$\Omega_{PC} \equiv \mathcal{R}[C(do(A(\vec{x}),s))]$$

where $\mathcal{R}$ is repeated regression rewriting (e.g., [22]) of $C(do(A(\vec{x}),s))$, the constraints relativized to $do(A(\vec{x}),s)$, using the successor state axioms, $\mathcal{D}_{SS}$ from $\mathcal{D}$. E.g.,

$$\Omega_{PC} \equiv \mathcal{R}[\neg(Away(dt, do(buyAirTicket(o,d,dt),s)) \wedge MustbeHome(dt, do(buyAirTicket(o,d,dt),s)))]$$

We rewrite this expression using the successor state axioms for fluents $Away(dt,s)$ and $MustbeHome(dt,s)$. E.g.,

$$\begin{aligned} &Away(dt, do(a,s)) \equiv \\ &\quad [(a = buyAirTicket(o,d,dt) \wedge d \neq Home) \\ &\quad \vee (Away(dt,s) \wedge \\ &\quad \neg(a = buyAirTicket(o,d,dt) \wedge d = Home))] \end{aligned}$$

$$MustbeHome(dt, do(a,s)) \equiv MustbeHome(dt,s)$$

From this we determine:

$$\begin{aligned} &Desirable(buyAirTicket(o,d,dt),s) \equiv \\ &\quad gt(DriveTime(o,d),3,s) \\ &\quad \wedge (d = Home \vee \neg MustbeHome(dt,s)) \end{aligned} \tag{3}$$

Having computed $\mathcal{D}_D$, we include it in $\mathcal{D}$[3]. In addition to computing $\mathcal{D}_D$, the set of $Desirable$ fluents, we also modify the computational semantics of our dialect of Golog.

that K to be reflexive, symmetric, transitive or Euclidean in the initial situation. (See [22, pg. 302-308].)

[2] Not always necessary, but we will require it in 2.1.

[3] Henceforth, all reference to $\mathcal{D}$ includes $\mathcal{D}_D$.

In particular, we adopt the *computational semantics* for Golog. (See [6] for details.) Two predicates are used to define the semantics. $Trans(\delta, s, \delta', s')$ is intended to say that the program δ in situation s may legally execute one step, ending in situation s' with the program δ' remaining. $Final(\delta, s)$ is intended to say that the program δ may legally terminate in situation s. We require one change in the definition to incorporate $Desirable$. In particular, (4) is replaced by (5).

$$\begin{aligned} &Trans(a, s, \delta', s') \equiv \\ &\quad Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s) \end{aligned} \tag{4}$$

$$\begin{aligned} &Trans(a, s, \delta', s') \equiv \\ &\quad Poss(a, s) \wedge Desirable(a, s) \\ &\quad \wedge \delta' = nil \wedge s' = do(a, s) \end{aligned} \tag{5}$$

We can encode this more compactly by simply defining $Legal(a, s) \equiv Poss(a, s) \wedge Desirable(a, s)$, and replacing $Poss$ with $Legal$ in (4). This approach has many advantages. First it is elaboration tolerant [17]. An individual's customized $\mathcal{D}_D$ may simply be added to an existing situation calculus axiomatization. If an individual's constraints change, the affected $Desirable$ fluents in $\mathcal{D}_D$ may be elaborated by a simple local rewrite. Further, $Desirable$ is easily implemented as an augmentation of most existing Golog interpreters. Finally, it reduces the search space for terminating situations, rather than pruning situations after they have been found. Thus, it has computational advantages over other approaches to determining preferred sequences of actions. Our approach is related to the approach to the qualification problem proposed by Lin and Reiter [15]. There are other types of customizing constraints which we do not address in this paper (e.g., soft and certain temporal constraints). We address these constraints in future work.

2.2 ADDING THE ORDER CONSTRUCT

In the previous subsection we described a way to customize Golog programs by incorporating user constraints. In order for Golog programs to be customizable and generic, they must have some nondeterminism to enable a variety of different choice points to incorporate user's constraints. Golog's nondeterministic choice of actions construct ($|$) and nondeterministic choice of arguments construct (π) both provide for nondeterminism in Golog programs.

In contrast, the sequence construct (;) provides no such flexibility, and can be overly constraining. Consider the program: $buyAirTicket(\vec{x}); rentCar(\vec{y})$. The ";" construct dictates that $rentCar(\vec{y})$ must be performed in the situation resulting from performing $buyAirTicket(\vec{x})$ and that $Poss(rentCar(\vec{y}), do(buyAirTicket(\vec{x}), s))$ must be true, otherwise the program will fail. Imagine that the precondition $Poss(rentCar(\vec{y}), s)$ dictates that the user's credit card not be over its limit. If $Poss$ is not true, we would like for the agent executing the program to have the flexibility to perform a sequence of actions to reduce the credit card balance, in order to achieve this precondition, rather than having the program fail. The sequence construct ";" does not provide for this flexibility.

To enable the insertion of actions in between a sequence of actions, for the purposes of achieving preconditions, we define a new construct called *order*, designated by the ":" connective[4]. Informally, $a_1 : a_2$ will perform the sequence of action $a_1; a_2$ whenever $Poss(a_2, do(a_1, s))$ is true. However, when it is false, the ":" construct dictates that Golog search for a sequence of actions $\vec{a}$ that achieves $Poss(a_2, do(\vec{a}, do(a_1, s)))$. This can be achieved by a planner that searches for a sequence of actions $\vec{a}$ to achieve the goal $Poss(a_2, do(\vec{a}, do(a_1, s)))$. To simplify this paper, we restrict a_2 to be a primitive action. The definition is easily extended to an order of complex actions $\delta_1 : \delta_2$. Thus, $a_1 : a_2$ is defined as:

$$a_1; \textbf{while } (\neg Poss(a_2)) \textbf{ do } (\pi a)[Poss(a)?; a] \textbf{ endWhile}; a_2$$

It is easy to see that the while loop will eventually achieve the precondition for a_2 if it can be achieved.

We extend the computational semantics to include ":".

$$\begin{aligned} &Trans(\delta : a, s, \delta', s') \equiv \\ &\quad Trans((\delta; achieve(Poss(a)); a, s, \delta', s') \end{aligned} \tag{6}$$

$$\begin{aligned} &Final(\delta : a, s) \equiv \\ &\quad Final(\delta; achieve(Poss(a)); a, s) \end{aligned} \tag{7}$$

where $achieve(G) = \textbf{while } (\neg G) \textbf{ do } (\pi a)[Poss(a)?; a]$ **endWhile**. Since $achieve$ is defined in terms of existing Golog constructs, the definitions of $Trans$ and $Final$ follow from previous definitions.

Note that ":" introduces undirected search into the instantiation process of Golog programs and though well-motivated for many programs, should be used with some discretion because of the potential computational overhead. We can improve upon this simplistic specification by a more directed realization of the action selection mechanism used by $achieve$ using various planning algorithms.

Also note that the order construct has been presented here independently of the notion of $Desirable$, introduced in the previous subsection. It is easy to incorporate the contributions of Section 2.2 by replacing $achieve(Poss(a))$ with $achieve(Legal(a))$ in Axioms (6) and (7) above, plus any other deontic notions we may wish to include. Finally note that a variant of the order construct also has utility in expressing narrative as proposed in [21]. We can modify ":" to express that actions a_1 and a_2 are ordered, but that it is not necessarily the case that a_2 occurred in situation $do(a_1, s)$.

[4]Created from a combination of existing constructs.

2.3 SELF-SUFFICIENT PROGRAMS

Now that our Golog programs are customizable and can be encoded generically, we wish them to be usable. Sensing actions are used when the agent has incomplete knowledge of the initial state (often true for WSC), or when exogenous actions exist that change the world in ways the agent's theory of the world does not predict. Web service compositions often have the characteristic of sequences of information-gathering services, performed to distinguish subsequent world-altering services. In our work, we need to define Golog programs that can be used by a variety of different agents without making assumptions about what the agent knows. As such, we want to ensure that our Golog programs are self-sufficient with respect to obtaining the knowledge that they require to execute the program. Further, we wish our programs to ensure that all preconditions for actions the program tries to execute are realized within the program, or are established as an explicit precondition of the program.

To make this concrete, we define the notion of a Golog program δ being *self-sufficient* with respect to an action theory $\mathcal{D}$ and *kernel initial state*, $Init_\delta$. $Init_\delta$ is a formula relativized to (suppressed) situation s, denoting the necessary preconditions for executing δ. To characterize self-sufficiency, we introduce the predicate $ssf(\delta, s)$. $ssf(\delta, s)$ is defined inductively over the structure of δ.

$$ssf(nil, s) \equiv true \tag{8}$$

$$ssf(\phi?, s) \equiv \mathbf{KWhether}^5(\phi, s) \tag{9}$$

$$\begin{aligned} ssf(a, s) \equiv\ & \mathbf{KWhether}(Poss(a, s)) \wedge \\ & \mathbf{KWhether}(Desirable(a, s)) \end{aligned} \tag{10}$$

$$\begin{aligned} ssf(\delta_1; \delta_2, s) \equiv\ & ssf(\delta_1, s) \wedge \\ & \forall s'.[Do(\delta_1, s, s') \supset ssf(\delta_2, s')] \end{aligned} \tag{11}$$

$$ssf(\delta_1 | \delta_2, s) \equiv ssf(\delta_1, s) \wedge ssf(\delta_2, s) \tag{12}$$

$$\begin{aligned} ssf(\delta^*, s) \equiv\ & ssf(\delta, s) \wedge \\ & \forall s'.[Do(\delta, s, s') \wedge ssf(\delta^*, s')] \end{aligned} \tag{13}$$

$$ssf((\pi x)\delta(x), s) \equiv \forall x. ssf(\delta(x), s) \tag{14}$$

$$\begin{aligned} & ssf(\mathbf{if}\ \phi\ \mathbf{then}\ \delta_1\ \mathbf{else}\ \delta_2\ \mathbf{endIf}, s) \equiv \\ & \quad \mathbf{KWhether}(\phi, s) \wedge \\ & \quad\quad (\phi(s) \supset ssf(\delta_1, s)) \wedge \\ & \quad\quad (\neg\phi(s) \supset ssf(\delta_2, s)) \end{aligned} \tag{15}$$

$$\begin{aligned} & ssf(\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{endWhile}, s) \equiv \\ & \quad \mathbf{KWhether}(\phi, s) \wedge \\ & \quad (\phi(s) \supset ssf(\delta, s) \wedge \\ & \quad\quad (\forall s'.[Do(\delta, s, s') \supset \\ & \quad\quad\quad ssf(\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{endWhile}, s')]) \\ & \quad) \end{aligned} \tag{16}$$

Since ":" is defined in terms of existing constructs, $ssf(\delta_1 : \delta_2, s)$ follows from (8)–(14) above.

[5] $\mathbf{KWhether}(\phi, s)$ abbreviates a formula indicating that the truth value of ϕ is known in situation s [24].

Definition 1 (KSSF: Knowledge Self-Sufficient Program) *$KSSF(\delta, Init_\delta)$, Golog program δ is knowledge self-sufficient relative to action theory $\mathcal{D}$ and kernel initial state $Init_\delta$ iff $\mathcal{D} \models Init_\delta(S_0)$ and $\mathcal{D} \models ssf(\delta, S_0) \equiv true$.*

$KSSF(\delta, Init_\delta)$ ensures that given $Init_\delta$, execution of the Golog program δ will not fail for lack of knowledge. However, the program may fail because it may be im-$Poss$-ible to perform an action. $KSSF$ ensures that the agent knows whether $Poss$ is true, but not that it actually *is* true. To further ensure that our generic procedures are physically self-sufficient, we define $PSSF(\delta, Init_\delta)$.

Definition 2 (PSSF: Physically Self-Sufficient Program) *$PSSF(\delta, Init_\delta)$, Golog program δ is physically self-sufficient relative to action theory $\mathcal{D}$ and kernel initial state $Init_\delta$ iff $KSSF(\delta, Init_\delta)$ and $\mathcal{D} \models \exists s'.Do(\delta, S_0, s')$.*

Proposition 1 *For every Golog program δ and associate kernel initial state $Init_\delta$, $PSSF(\delta, Init_\delta) \supset KSSF(\delta, Init_\delta)$.*

This follows directly from Definitions 1-2.

Next we discuss how to verify KSSF and PSSF for a common subset of Golog programs.

We call a Golog program, δ *loop-free* if it does not contain the nondeterministic iteration and while-loop constructs. Note that we may preserve the loop-free nature of our programs, while using these programming constructs by defining a maximum iteration count, or time-out. It follows that.

Proposition 2 *For every loop-free Golog program $(\delta, Init_\delta)$ and associated situation calculus theory $\mathcal{D}$, there exist first-order situation calculus formulae ϕ_P and ϕ_K such that*

$$KSSF(\delta, Init_\delta) \equiv \phi_K \text{ and } PSSF(\delta, Init_\delta) \equiv \phi_P,$$

and ϕ_P and ϕ_K do not mention ssf.

The proof is inductive over the structure of δ.

From these propositions, it follows that $PSSF$ and $KSSF$ of loop-free programs, $(\delta, Init_\delta)$ can be verified using regression followed by theorem proving in the initial situation. For programs with potentially unlimited looping, $\phi_K(s)$ and $\phi_P(s)$ are not first-order definable, and hence are problematic.

Proposition 3 *Let δ be a loop-free Golog program, and let ϕ_K and ϕ_P be defined as in Proposition 2. Let $\mathcal{K}_{init}$ consist of any subset of the accessibility relations Reflexive, Symmetric, Transitive, Euclidean, then*

1. *$KSSF(\delta, Init_\delta)$ iff $D_{una} \cup D_{S_0} \cup \mathcal{K}_{init} \models \mathcal{R}[\phi_K]$*
2. *$PSSF(\delta, Init_\delta)$ iff $D_{una} \cup D_{S_0} \cup \mathcal{K}_{init} \models \mathcal{R}[\phi_P]$.*

This follows directly from Reiter's regression theorem with knowledge [22].

We wish to highlight the work of Ernie Davis on Knowledge Preconditions for Plans [5], which we became aware of when we first presented ssf [20]. There are many similarities to our work. One significant difference is that he makes no distinction between (what we distinguish as) knowledge sufficiency and physically sufficiency in his framework, i.e., for a plan to be executable, he requires that the agent has the knowledge to execute it and that it must be physically possible. Further, we construct the ssf condition from the situation calculus theory for primitive actions that can be regressed over situations and verified in the initial situation. He develops a set of rules that can be used to check for plan executability. The set of rules, is sufficient but not necessary.

3 EXECUTING GOLOG PROGRAMS

Now that we have defined customizable, generic and usable generic procedures for WSC, we must execute them. In building a Golog interpreter that incorporates sensing actions, the interplay between sensing and execution of world-altering actions can be complex and a number of different approaches have been discussed (e.g., [7, 13, 22]). While [7] and [22] advocate the use of an online interpreter to reason with sensing actions, [13] suggests the use of an offline interpreter with conditional plans. The trade-off is clear. An online interpreter is incomplete because no backtracking is allowed, while an offline interpreter is computationally expensive due to the much larger search space, and the need to generate conditional plans, if sensing actions are involved. The choice between an online and offline interpreter depends on properties of the domain, and in particular, since exogenous actions can affect the value of fluents, on the temporal extent of the persistence of the information being sensed. In a robotics domain, an online interpreter is often more appropriate, whereas an offline interpreter is more appropriate for contingency planning.

3.1 MIDDLE-GROUND EXECUTION

We define a middle ground between offline and online execution, which we argue is appropriate for a large class of Semantic Web WSC applications. Our middle-ground interpreter (MG) senses online to collect the relevant information needed in the Golog program, while only simulating the effects of world-altering actions. By executing sensing actions rather than branching and creating a conditional plan, MG reduces search space size, while maintaining the ability to backtrack by merely simulating world-altering actions, initially. The outcome is a sequence of world-altering actions that are subsequently executed[6]. Humans often follow this approach, collecting information on the Web (e.g., flight schedules) while only simulating the world-altering actions (buying tickets, etc.) in their head until they have a completed plan to execute.

Of course, the veracity of MG is predicated on an important assumption – that the information being gathered, and upon which world-altering actions are being selected, persists. We assume that the fluents MG is sensing persist for a reasonable period of time, and that none of the actions in the program cause this assumption to be violated. This assumption is generally true of much of the information we access on the Web (e.g., flight schedules, store merchandise), but not all (e.g., stock prices). This assumption is much less pervasive in mobile robotic applications where we may assume persistence for milliseconds, rather than minutes or hours. We formalize this assumption as follows.

Definition 3 (Conditioned-on Fluent) *Fluent C is a conditioned-on fluent in Golog program δ iff δ contains the Golog construct $\phi?$ and C appears in formula ϕ.*

Recall that the $\phi?$ construct is used to define the conditional (if-then-else) and the while-loop constructs. It is also commonly used within the δ of $(\pi x)\delta(x)$.

Definition 4 (Invocation and Reasonable Persistence (IRP) Assumption) *Golog program and kernel initial state $(\delta, Init_\delta)$ adhere to the invocation and reasonable persistence assumption if*

1. Non-knowledge preconditions for sensing actions are true in $\mathcal{D}_{S_0} \cup Init_\delta(S_0)$.

2. Knowledge of preconditions for actions and conditioned-on fluents C in δ, once established, persists[7].

Condition 1 ensures that all sensing actions can be executed by the MG interpreter. Condition 2 ensures that decisions are predicated on correct information. Condition 1 may seem extreme, but, as we argued earlier in this paper, by their nature, Web services generally only have knowledge preconditions. The persistence of knowledge in Condition 2, trivially holds from the frame assumption for knowledge. This condition addresses change by subsequent or exogenous actions.

We claim that under the IRP assumption, MG does the right thing for programs that are physically self-sufficient.

Claim 1 (Veracity of MG) *Given an action theory D and*

[6] At this stage they can alternately be shown to a human for approval before execution. Our interpreter can also generate and present multiple alternate courses of action.

[7] I.e., no subsequent actions inside or outside the program change the value of sensed fluents.

Golog program δ such that $PSSF(\delta, Init_\delta)$, and $(\delta, Init_\delta)$ adheres to IRP, let $\vec{\alpha}_w$ be the sequence of world-altering actions selected by the middle-ground interpreter, MG, for subsequent execution. Assuming no exogenous actions and no sensor errors[8]*, it follows that executing $\vec{\alpha}_w$ yields the same truth value for all fluents F in $\mathcal{D}$ as an online interpreter with an oracle that chooses $\vec{\alpha_w}$ at the appropriate branch points in its interpretation of δ.*

Let $do(\vec{\alpha}, S_0)$ be the terminating situation, following execution of Golog program δ with theory $\mathcal{D}$, using the interpreter MG, starting in S_0. Then $\mathcal{D} \models Do(\delta, S_0, do(\vec{\alpha}, S_0))$, and we denote the sequence of actions $\vec{\alpha}$ by the relation MG$(\mathcal{D}, \delta, Init_\delta, \vec{\alpha})$.

$\vec{\alpha}$ is comprised of both sensing actions and world-altering actions (e.g., $[s_1, a_1, a_2, a_3, s_2, a_4]$). Let $\vec{\alpha}_s$ be the sequence of sensing actions in $\vec{\alpha}$ (i.e., $[s_1, s_2]$), and likewise let $\vec{\alpha}_w$ be the sequence of world-altering actions in $\vec{\alpha}$ (i.e., $[a_1, a_2, a_3, a_4]$). MG executes the sensing actions $\vec{\alpha}_s$, interleaved with the simulation of world-altering actions, searching to find the appropriate terminating situation, $\vec{\alpha}$. MG then outputs $\vec{\alpha}$. $\vec{\alpha}_w$, the subsequence of world-altering actions, are extracted from $\vec{\alpha}$ and executed in the world.

Theorem 1 *Given an action theory D and Golog program δ such that $PSSF(\delta, Init_\delta)$, and $(\delta, Init_\delta)$ adheres to IRP, suppose MG$(\mathcal{D}, \delta, Init_\delta, \vec{\alpha})$ holds for some $\vec{\alpha}$. Assume that there are no sensor errors, and that no exogenous actions affect fluents in $\mathcal{D}$, then for all fluents F in $\mathcal{D}$*

$$\mathcal{D} \models F(\vec{x}, do(\vec{\alpha}_w, do(\vec{\alpha}_s), S_0)) \equiv F(\vec{x}, do(\vec{\alpha}, S_0)).$$

In cases where the IRP Assumption is at risk of being violated, the full sequence of sensing and world-altering actions generated by MG, $\vec{\alpha}$, could be re-executed with an online execution monitoring system. The system would reperform sensing actions to verify that critical persistence assumptions were not violated. In the case where the IRP Assumption does not hold for some or all conditioned-on fluents in a Golog program, MG could be integrated with an interpreter that builds conditional plans for branch points that do not adhere to IRP, following the approach proposed in [13]. The explicit encoding of search areas in a program, as proposed by [7] through the addition of their Σ search construct, can achieve some of the same functionality as our middle-ground interpreter. Indeed, the principle defined above, together with an annotation of the temporal extent of conditioned-on fluents within the action theory provides a means of automatically generating programs with embedded search operators Σ, as proposed in [7]. We leave a formal account of this to a future paper.

[8] Trivially true of virtually all current-day information-gathering Web services.

3.2 MIDDLE-GROUND PROLOG INTERPRETER

In Section 2, we proposed extensions to Golog to enable programs to be generic, customizable and self-sufficient. In Section 3, we proposed a strategy for middle-ground execution that enables an efficient and thorough combination of sensing and search. We have modified the ConGolog offline interpreter in [7, 6] to realize these enhancements. We describe the necessary code modifications in the subsections to follow, and prove the correctness of our implementation. We adopt notational conventions for Prolog code, that differ from those used for theories of action. To avoid confusion, all Prolog code is listed in courier font. Prolog variables are uppercase, and constants are lowercase, contradicting the situation calculus notational convention.

3.2.1 User customizing constraints

Personal constraints were added to the ConGolog interpreter by the following straightforward and elegant modification to the code, that accounts for the addition of the $Desirable$ predicate. We replaced the following code:

```
trans(A,S,R,S1) :-  primAct(A),
  (poss(A,S), R=nil, S1=do(A,S)); fail.
```

of the ConGolog interpreter with

```
trans(A,S,R,S1) :- primAct(A),
  (poss(A,S), desirable(A,S),
     R=nil, S1=do(A,S));  fail.
```

This ensures that every action selected by the interpreter is also a desirable one.

In Section 3.3, we will show how to encode the $Desirable(a, s)$ predicate in our Prolog action theory. The domain-independent Prolog rules for the MG interpreter include the following rule to ensure that actions are desirable unless proved otherwise.

```
desirable(A,S):- \+ not_desirable(A,S).
```

3.2.2 Order Construct

To include the order construct ":", we added the following rules to our interpreter:

```
final(P:A, S):-
    action(A),
    final([P,achieve(poss(A),0),A],S).
trans(P:A,S,R,S1):-
    action(A),
    trans([P,achieve(poss(A),0),A],S,R,S1).
```

where achieve(Goal,0) is an A*-planner, adapted from the so-called *World Simplest Breath First Planner* (wsbfp) developed by Reiter [22, pg. 234]. We appeal to its simplicity and the soundness and completeness of the A* algorithm. Obviously any planner can be used to accomplish this task. We are investigating the effectiveness of other planners.

3.2.3 Sensing Actions

We incorporate sensing actions and their effects into our interpreter, using an approach that eliminates the need for the situation calculus K fluent. Recently, Soutchanski [25] proposed a somewhat similar transformation, articulating conditions under which his representation was correct. This is also similar in spirit to the approach in [8].

To accommodate both backtracking and sensing, we assume that the truth value of a certain fluent, say $F(\vec{x}, s)$, can be determined by executing *an external function call*, A. The call is denoted by $exec(A(\vec{x}), s)$. Whenever the execution succeeds, F is true; otherwise, it is false. Note that because Prolog answers queries with free variables by returning possible values for these variables, this technique is equally suitable for sensed functional fluents. This is illustrated in Example 2. The use of external function calls, together with the IRP Assumption, allows us to write equations of the following form, which are embodied into the successor state axiom of a fluent $F(\vec{x})$:

$$F(\vec{x}, do(A(\vec{x}), s)) \equiv exec(A(\vec{x}), s) \tag{17}$$

Equation (17) is translated into Prolog as follows

```
holds(f(X), do(a(X), S)):- exec(a(X), S).
```

In addition, we need to provide the set of rules that call the action a externally.

```
exec(a(X), S):- <external function call>
```

These rules are domain dependent and may be unique to the specific Prolog encoded situation calculus domain theory. In the following section, we discuss how to translate our situation calculus theories into Prolog, and illustrate how we make external function calls for WSC.

3.3 TRANSLATING SITCALC TO PROLOG

To translate a specific situation calculus theory, $\mathcal{D}$ into a set of Prolog rules, $\mathcal{D}^P$, we follow the description provided in [22], with the following additions, predominantly to accommodate sensing actions and the $Desirable$ predicate.

- For each propositional fluent $F(\vec{x}, s)$, create a corresponding propositional fluent `f(X,S)` in D^P.
- For each functional fluent $F(\vec{x}, s) = \vec{y}$, create a corresponding propositional fluent `f(X, Y, S)` in D^P.
- Successor state axioms for non-sensed fluents are translated into Prolog following the description in [22]. For fluents whose truth value can change as the result of a sensing action, the normal successor state axiom encoding is augmented to include the suitable external function calls. Further, we add the necessary code to realize those calls.
- For each action $A(\vec{x})$ for which $Desirable(A(\vec{x}), s) \equiv \Omega(A(\vec{x}), s)$ is defined following Equation (2), create a Prolog rule `not_desirable(a(X),S):- \+ omega(a(X),S)`. We use Prolog's negation as failure to infer that by default an action is desirable in a situation, as per the MG code in Section 3.2.1.

We illustrate the points above with the following examples.

Example 1: We return to our simple travel theme. Many of the major airlines offer WWW programs that allow a user to buy an air ticket online. We conceive these services as world-altering actions in the situation calculus, e.g., $buyAirTicket(o, d, dt)$. To simplify the example, we disregard all parameters except origin, destination, and date (o, d, dt, respectively). This service has a number of effects including asserting that the user will own a ticket after its execution. The successor state axiom for $ownAirTicket(o, d, dt, s)$ is as follows.

$$\begin{aligned}
&ownAirTicket(o, d, dt, do(a, s)) \equiv \\
&\quad (a = buyAirTicket(o, d, dt) \wedge ticketAvail(o, d, dt)) \vee \\
&\quad (a \neq buyAirTicket(o, d, dt) \\
&\quad\quad \wedge\ ownAirTicket(o, d, dt, s))
\end{aligned}$$

which is encoded in Prolog as follow.

```
holds(ownAirTicket(O, D, DT, do(E,S)):-
    E = buyAirTicket(O, D, DT),
    holds(ticketAvail(O, D, DT), S);
    \+ E = buyAirTicket(O, D, DT),
    holds(ownAirTicket(O, D, DT), S).
```

Recall that MG does not execute world-altering actions. Hence there is no external function to execute `buyAirTicket`. To create an interpreter that executed (some) world-altering actions immediately, the Prolog code would be modified analogously to the sensing-action code below.

Example 2: Now consider the map service offered at www.yahoo.com. Simplified, this service takes as input the origin of a trip o, and the destination d, and returns, among other things, the driving time between o and d. In the context of WSC, we view this service as the sensing action, $getDrivingTime(o, d)$ which, after its execution, tells us the value of the functional fluent $DriveTime(o, d, s)$. In Prolog, we create a corresponding propositional fluent, `driveTime(O,D,T,S)`. For simplicity, we assume $DriveTime(o, d, s)$ is only sensed by $getDrivingTime(o, d)$, but the extension to multiple actions is trivial.

The successor state axiom for `driveTime(O,D,T)` is as follows. Note the call to execute `ex_get_driving_time(O, D, T)`.

```
holds(driveTime(O, D, T), do(E,S)):-
    E = getDrivingTime(O, D),
```

```
        exec(ex_get_driving_time(O, D, T)) ;
    \+ E = getDrivingTime(O, D),
    holds(driveTime(O, D, T), S).
```

To specify how the action `ex_get_driving_time(O, D, T)` is executed, we need additional code. In our implementation, our sensing actions are all calls to Web services. We do this via a call to the Open Agent Architecture (OAA) agent brokering system [16]. OAA in turn requests a service named `get_directions`.

We first write a general rule for the generic `exec` call.

```
exec(A):- A.
```

We also write a rule for the actual call.

```
ex_get_driving_time(O, D, T):-
  oaa_Solve(
    get_directions(O, ' ', D, ' ', 0, X),[]),
  drvTime(X, T).
```

`oaa_Solve(get_directions(...),[])` requests the yahoo service that provides the driving time from `O` to `D`. As the information provided by the service (`X`) contains additional information including images and driving directions, we have written some additional code to extract what we need from what the service returns. The code extraction is performed by `drvTime(X, T)`. We omit the details in the interest of space. As we will discuss in the following section, when our vision of semantic Web services is realized, such extra code will be unnecessary.

Example 3: In Section 2.1, we discussed Marielle's personal preferences and showed how they translated into Equation (3). Here we show how we represent this axiom in our Prolog encoding of the domain theory.

```
not_desirable(buyAirTicket(O, D, DT), S):-
    holds(driveTime(O, D, T), S),
    T <= 3 ;
    \+ D = ``home'', holds(mustBeHome(DT, S)).
```

3.4 CORRECTNESS OF THE INTERPRETER

We complete Section 3 with a theorem that proves the correctness of our interpreter.

Theorem 2 *Given an action theory $\mathcal{D}$ and Golog program δ such that $PSSF(\delta, Init_\delta)$, and $(\delta, Init_\delta)$ adheres to IRP, if $D^P \cup MG^P \vdash Do(\delta, S_0, S)$ then there exists a model $\mathcal{M}$ of $\mathcal{D}$ such that $\mathcal{M} \models Do(\delta, S_0, S)$, where D^P is the set of Prolog rules representing $\mathcal{D}$, MG^P is the set of Prolog rules representing our MG Golog interpreter, and $\vdash$ is proof by our Prolog interpreter.*

Following [22], the action theories in this paper are definitional theories when $\mathcal{D}_{S_0}$ is complete, i.e., when $\mathcal{D}_{S_0}$ contains a definition for each fluent in the theory. This can be achieved by making the closed-world assumption (CWA). Since our programs are self-sufficient, this seems less egregious. Proposition 4 follows immediately from the Implementation Theorem of [22].

Proposition 4 *Given an action theory $\mathcal{D}$ and Golog program δ such that $PSSF(\delta, Init_\delta)$, and $(\delta, Init_\delta)$ adheres to IRP. Then, for all situations S,*

$$D^P_{CWA} \cup MG^P \ \vdash \ Do(\delta, S_0, S) \text{ iff } \mathcal{D} \cup CWA(S_0) \models Do(\delta, S_0, S),$$

where $CWA(S_0)$ is the closed-world assumption on S_0, defined as $\{F(S_0) \equiv false \mid$ there exists no definition of F in $\mathcal{D}_{S_0}\}$, MG^P is the set of Prolog rules for our MG Golog interpreter, D^P_{CWA} is the set of Prolog rules representing $\mathcal{D} \cup CWS(S_0)$, and $\vdash$ is proof by our Prolog interpreter.

4 COMPOSING WEB SERVICES

A significant aspect of our contribution is that the research described to this point is implemented and has been tested on a running system that interacts with services on the Web. In this section, we step back and situate the agent technology we've been describing in the context of our system architecture for *Semantic Web Service Composition*. We also discuss further details of our implementation. Finally, we conclude this section with an example generic procedure that illustrates the use of our work.

4.1 ARCHITECTURE

Figure 1 illustrates the key components of our semantic WSC architecture [19]. Of course, the Semantic Web does not exist yet – www.yahoo.com does not use semantic markup such as DAML+OIL to describe its services nor to disseminate information. We describe both the architecture for our system, and in the section to follow discuss how we've accommodated for the pieces of the architecture that are not yet realizable in an elegant way.

The key features of this architecture follow.

Semantic Markup of Web Services: Individual Web services are described in a semantic Web markup language. The programs, their control structure and data flow, are described using a declarative process modeling language. Processes are either atomic or composite. Each process has inputs, outputs, preconditions and effects. It also has a grounding that describes the communication-level properties of the service. A service profile is created for describing and locating the service. Collectively, this semantic markup provides a declarative API for the service so that programs/agents can read this markup and understand how to interact with a service.

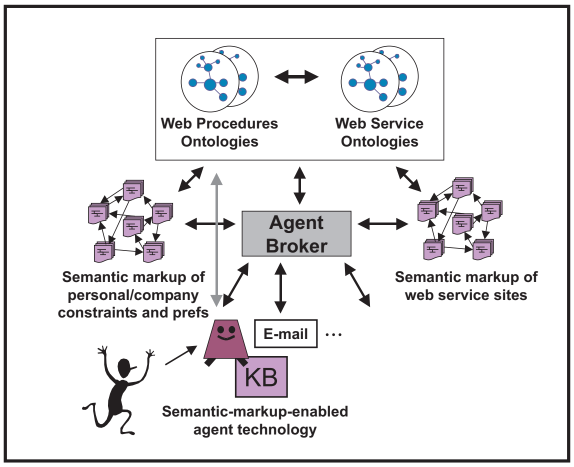

Figure 1: Semantic WSC Architecture

Ontologies of Web Services: To encourage reuse of vocabulary, parsimony, and shared semantic understanding, individual Web service descriptions are organized into Web service ontologies. For example, an ontology might contain a service class called `buy`, with subclasses `buyTicket`, `buyBook`, etc. `buyTicket` might in turn be subclassed into `buyAirTicket`, etc. Thus, a Web service provider wishing to describe a new service can simply subclass an existing service, inheriting vocabulary, and ensuring some degree of integration with existing systems.

Semantic Markup and Ontologies of Generic Procedures: Generic procedures can be described using the same semantic markup constructs used to describe Web services. After all, they are just programs. Similarly, generic procedures are stored in ontologies to facilitate sharing and reuse. The ability to share generic procedures is what motivated our desire to make procedures knowledge and physically self-sufficient.

Semantic Markup of Personal/Company Constraints: In addition to semantic markup of services, people can archive their personal preferences and constraints as semantic markup. These profiles can likewise be stored in ontologies, so that users can inherit constraints from family, their place of work, or other affiliations.

Semantic-Markup-Enabled Agent Technology: The architecture also includes a variety of agent technologies that communicate with Web services through an Agent Broker. Our Golog interpreter is one such agent technology.

Agent Broker: The agent broker accepts requests for services from the agent technology or other services, selects an appropriate service and directs the request to that service. Likewise it relays responses back to the requester.

4.2 IMPLEMENTATION

To realize our agent technology, we started with a simple implementation of an offline ConGolog interpreter in Quintus Prolog 3.2. We have modified and extended this interpreter as described in Section 3. Agent brokering is performed using the Open Agent Architecture (OAA) agent brokering system [16]. We have modified our Golog interpreter to communicate with Web services via OAA. Unfortunately, commercial Web services currently do not utilize semantic markup. In order to provide a computer-interpretable API, and computer-interpretable output, we use an information extraction program, World Wide Web Wrapper Factory[9] (W4). This program extracts the information we need from the HTML output of Web services. All information-gathering actions are performed this way. For obvious practical (and financial!) reasons, we are not actually executing world-altering services.

All the core infrastructure is working and our Golog interpreter is communicating with services on the Web via OAA. We first demoed our Golog-OAA-WWW system in August, 2000 [19]. Since then, we have been refining it and working on Semantic Web connections. Where our architecture has not been fully realized is with respect to full automation of semantic markup. We originally constructed rudimentary service and procedure ontologies in first-order logic. We are migrating these to DAML-S, as we complete our DAML-S specification. Eventually our Golog interpreter, will populate its KB from the DAML-S ontologies and from DAML+OIL ontologies of user's customizing constraints. These declarative representations will be compiled into a situation calculus representation. We have constructed partial compilers for DAML-S to PDDL[10], and for PDDL to the situation calculus, but we are still predominantly hand-coding situation calculus theories at this time.

4.3 EXAMPLE

We complete this section with an example generic procedure. Consider the example composition task given at the beginning of this paper, "Make the travel arrangements for my KR2002 conference trip." If you were to perform this task yourself using services available on the Web, you might first find the KR2002 conference Web page and determine the location and dates of the conference. Based on the location, you would decide upon the most appropriate mode of transportation. If traveling by air, you might then check flight schedules with one or more Web services, book flights, and arrange transportation to the airport through another Web service. Otherwise, you might book a rental car. You would then need to arrange transportation and accom-

[9] db.cis.upenn.edu/W4/

[10] Plan Domain Definition Language.

modations at the conference location, and so on.

We have created a generic procedure for arranging travel that captures many aspects of this example. Our generic procedure selects and books transportation (car/air), hotel, local transportation, emails the customer an itinerary, and updates an online expense claim. As noted previously, these generic procedures are not that complex – they are indeed generic. It is the interplay with user constraints that makes our approach powerful.

In what follows we provide Prolog code for a subset of our generic travel procedure. We have simplified the program slightly (particularly the number of parameters) for illustration purposes. We have also used informative constant and term names to avoid explanation. `D1` and `D2` are the departure and return dates of our trip. `pi` is the nondeterministic choice of action arguments construct, π. Sensing actions, such as `searchForRFlight()` have associated execution code, not included here. Recall that to interpret this generic procedure, Golog will look for actions that are `desirable` as well as possible.

The following is a generic procedure for booking a return airline ticket.

```
proc(bookRAirTicket(O, D, D1, D2),
[
  poss(searchForRFlight(O, D, D1,D2)) ?,
  searchForRFlight(O, D, D1, D2),
  [ pi(price,
         [ rflight(ID, price) ?,
           (price < usermaxprice) ?,
           buyRAirTicket(ID, price) ])
  ]
]).
```

Note the choice of flight based on price using π. Procedures for booking a car or hotel can be written in a similar fashion. We compose such procedures to make a Golog travel program.

```
proc(travel(D1, D2, O, D),
[
   [  bookRAirticket(O, D, D1, D2),
      bookCar(D, D, D1, D2)
   ] |
   bookCar(O, O, D1, D2),
   bookHotel(D, D1, D2),
   sendEmail,
   updateExpenseClaim
]).
```

Note the use of nondeterministic choice of actions. If booking a return air ticket or booking a car at the destination prove undesirable, Golog tries to book a car at the origin so the user can drive to the destination and back.

We have tested our generic travel procedure with different tasks and a different user constraints. These tests have confirmed the ease and versatility of our approach to WSC.

5 SUMMARY & RELATED WORK

In this paper we addressed the problem of automated Web service composition and execution for the Semantic Web. We developed and extended theoretical research in reasoning about action and cognitive robotics, implemented it and experimented with it. We addressed the WSC problem through the provision of high-level generic procedures and customizing constraints. We proposed Golog as a natural formalism for this task. As an alternative to planning, our approach does not change the computational complexity of the task of generating a composition. Nevertheless, most Web service compositions are short, and the search space is broad. Consequently, our approach has the potential to drastically reduce the search space, making it computationally advantageous. Additionally, it is compelling, and easy for the average Web user to use and customize.

Our goal was to develop Golog generic procedures that were easy to use, generic, customizable, and that were usable by a variety of users under varying conditions. We augmented Golog with the ability to include customizing user constraints. We also added a new programming construct called *order* that relaxes the notion of *sequence*, enabling the insertion of actions to achieve the precondition for the next action to be performed by the program. This construct facilitates customization as well as enabling more generic procedures. Finally, we defined the notion of knowledge and physically self-sufficient programs that are executable with minimal assumptions about the agent's initial state of knowledge, or the state of the world. We showed that these criteria could be verified using regression and theorem proving. Adherence to these criteria makes our generic procedures amenable to wide-spread use. To execute our programs, we defined a middle-ground approach to execution that performed online execution of necessary information-gathering Web services with offline simulation of world-altering services. Thus, our MG interpreter determined a sequence of world-altering Web Services for subsequent execution. We proved that our approach to execution had the intended consequences, under the IRP assumption.

These contributions were implemented as modifications to an existing ConGolog interpreter and we proved the correctness of our implementation. Further they have been intergrated into a Semantic Web Architecture, that includes an agent broker for communication with Web Services, and a variety of service-related ontologies. We have tested our results with a generic procedure for travel and a variety of different customizing constraints that showcase the effectiveness of our approach. Though our work was focused on Web service composition, the work presented in this paper has broad relevance to a variety of cognitive robotic tasks.

Much related work was identified in the body of this paper, with the work in [7, 22], and more recently, [23], being most closely related. Several other agent technologies deserve mention. The topic of agents on the internet has been popular over the years. Some of the first and most related work is the softbot work done at the University of Washington [9]. They also use action schemas to describe information-providing and world-altering actions that an agent can use to plan to achieve a goal on the internet. More recently, [26, 1, 12] have all developed some sort of agent technology that interacts with the Web.

ACKNOWLEDGEMENTS

We thank the Cognitive Robotics Group at the University of Toronto for providing an initial ConGolog interpreter that we have extended and augmented, and SRI for the use of the Open Agent Architecture software. We would also like to thank Honglei Zeng for his work on the OAA interface to our Golog code [19]. Finally, we gratefully acknowledge the financial support of the US Defense Advanced Research Projects Agency DAML Program grant number F30602-00-2-0579-P00001. The second author would also like to acknowledge the support of NSF grant NSF-EIA-981072 and NASA grant NCC2-1232.

References

[1] V. Benjamins et al. IBROW3: An Intelligent Brokering Service for Knowledge-Component Reuse on the World Wide Web. In *KAW'98, Banff, Canada*

[2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. In *Scientific American*, May 2001.

[3] DAML-S. http://www.daml.org/services, 2001.

[4] DAML-S Coalition: A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S: Semantic markup for Web services. In *Proc. Int. Semantic Web Working Symposium (SWWS)*, 411–430, 2001.

[5] E. Davis. Knowledge Preconditions for Plans. *Journal of Logic and Computation*, 4(5):721–766, 1994.

[6] G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus. *AIJ*, 121(1-2):109–169, 2000.

[7] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing. In *Logical Foundations for Cognitive Agents, Contributions in Honor of Ray Reiter*, pages 86–102, 1999.

[8] G. De Giacomo and H. Levesque. Projection using regression and sensors. In *IJCAI'99*, 160–165, 1999.

[9] O. Etzioni and D. Weld. A softbot-based interface to the internet. *JACM*, pages 72–76, July 1994.

[10] J. Hendler and D. McGuinness. The DARPA agent markup language. In *IEEE Intelligent Systems Trends and Controversies*, November/December 2000.

[11] I. Horrocks, F. van Harmelen, P. Patel-Schneider, T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, D. Fensel, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, and L. Stein. DAML+OIL, March 2001. http://www.daml.org/2001/03/daml+oil-index.

[12] C. Knoblock, et. al. Mixed-initiative, multi-source information assistants. In *Proceedings of the 10th International Conference on the World Wide Web*, pages 697–707, 2001.

[13] G. Lakemeyer. On sensing and off-line interpreting in Golog. In *Logical Foundations for Cognitive Agents, Contr. in Honor of Ray Reiter*, pages 173–187, 1999.

[14] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, April-June 1997.

[15] F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994. Special Issue on Action and Processes.

[16] D. L. Martin, A. J. Cheyer, and D. B. Moran. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence*, 13:91–128, January-March 1999.

[17] J. McCarthy. Mathematical logic in artificial intelligence. *Daedalus*, pages 297–311, Winter, 1988.

[18] S. McIlraith and R. Fadel. Planning with complex actions. Submitted for publication, 2002.

[19] S. McIlraith, T. Son, and H. Zeng. Semantic Web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 16(2):46–53, March/April 2001.

[20] S. McIlraith and T. C. Son. Adapting ConGolog for Programming the Semantics Web. In *Working Notes of The Fifth International Symposium on Logical Formalization of Commonsense Reasoning*, pages 195–202, 2001.

[21] R. Reiter. Narratives as programs. In *Proc. of the Seventh International Conference on Knowledge Representation and Reasoning (KR2000)*, pages 99–108, 2000.

[22] R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Specifying and Implementing Dynamical Systems*. The MIT Press, 2001.

[23] S. Sardiña. Local conditional high-level robot programs. In *Proceedings of the 4th Workshop on Nonmonotonic Reasoning and Action, IJCAI, August 2001*, pages 195–202, 2001.

[24] R. Scherl and H. Levesque. The frame problem and knowledge producing actions. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 689–695, 1993.

[25] M. Soutchanski. A Correspondence Between Two Different Solutions to the Projection Task with Sensing. In *Working Notes of Common Sense 2001*, pages 235–242, 2001.

[26] R. Waldinger. Deductive composition of Web software agents. In *Proc. NASA Wkshp on Formal Approaches to Agent-Based Systems, LNCS*. Springer-Verlag, 2000.