Adapting Golog for Programming the Semantic Web*

Sheila McIlraith
Knowledge Systems Laboratory
Department of Computer Science
Gates Building, Wing 2A
Stanford University, Stanford, CA 94305-9020, USA
sam@ksl.stanford.edu

Tran Cao Son †
New Mexico State University
Computer Science Department
PO Box 30001
Las Cruces, NM 88003-8001
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 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 programming Web services. 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. We realize these extensions in an augmented 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

The Web, once solely a repository for documents, is evolving into a provider of services – information-providing services such as flight information providers, temperature sensors, and cameras; and world-altering services such as flight booking programs, sensor controllers, and a variety of E-Commerce and B2B applications. These services are realized by Web-accessible programs, databases, sensors, and eventually by a variety of other physical devices. Today's Web was designed primarily for human use. As the Web grows in size and diversity there is an increased need to automate aspects of Web services such as 1) automatic Web service *discovery* (e.g., "Find me a service that books airline tickets from San Francisco to Saskatoon, and that accepts En Route credit cards."), 2) automatic Web service *execution* (e.g., "Buy me 'Harry Potter and the Sorcerer's Stone' at www.acmebooks.com"), and 3) automatic Web service *selection, composition and interoperation* (e.g., "Make the travel arrangements for my IJCAI conference trip."). To realize wide-scale Web service automation we must make Web service properties and capabilities understandable to computer programs – we must create *semantic Web services* [15].

The so-called "Semantic Web" [1] is emerging with the development of *semantic Web markup languages* such as DAML-ONT [2], OIL [20], and most recently DAML+OIL [3]. These markup languages build on the foundations of XML and RDF(S) [9] to provide the taxonomic and logical expressiveness of description logics in a markup language with a clearly defined semantics. Over the next year, it is hoped that there will be a proposal for a semantic Web markup language that has much of the expressive capability of Horn clause logic. These markup languages, while still in their infancy, are being used to mark up Web pages. In our work, we are developing a knowledge representation scheme to mark up Web services in order to automate Web service discovery, execution, selection, composition and interoperation. Each service is conceived as a primitive or complex action. In using an action metaphor for marking up Web services, we are able to bring to bear much of the research on reasoning about action, to assist in developing technology for Web service composition and interoperation. We do not discuss the markup of Web services in this paper, nor the architecture that connects our markup to the technology developed here. The reader is referred to (e.g., [15]) for further details.

In this paper we focus on the issue of automating Web service composition by *programming* agents to compose and execute services on the Web. Our goal is to develop programs that are easy to use, generic, customizable, and that are usable by a variety of users under varying conditions. We argue that the logic programming language Golog (e.g., [11; 16; 4]) provides a natural formalism for this task. To this end, in Section 3, we augment Golog with the ability to include customizing user constraints. We also add a new programming construct called *order* that relaxes the notion of *sequence*, enabling the definition of more generic procedures. Finally, we define the notion of self-sufficient programs that are executable with minimal assumptions about the agent's initial state of knowledge, making them amenable to widespread use. In Section 4, we realize these ideas in an augmented ConGolog interpreter that combines online execution of information-gathering Web services with offline simulation of world-altering Web services, to determine a sequence of world-altering services for subsequent execution. Our ap-

*This paper will also be presented at the IJCAI 2001 Workshop on Nonmonotonic Reasoning, Action and Change (NRAC-01).

†Most of this work was done while the author was a postdoctoral fellow in the Knowledge Systems Lab, Stanford University.

proach to sensing and acting presents a middle ground between online and offline execution, leveraging properties of our Web domain. We describe our implementation briefly at the end of this paper.

2 Programming the Semantic Web

Consider the example composition task given in the previous section, "Make the travel arrangements for my IJCAI conference trip." If you were to perform this task yourself using services available on the Web, you might first find the IJCAI'2001 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 accommodations at the conference location, and so on.

While the procedure is lengthy and somewhat tedious to perform, what you have to do to make travel arrangements is easily described by the average person. Nevertheless, many of us know it is very difficult to get someone else to actually make your travel arrangements for you. What makes this task difficult to perform is not the basic steps but the need to make choices and customize the procedure in order to enforce your personal preferences and constraints. For example, you may only like to fly on selected airlines where you can collect travel bonus points. You may have scheduled activities that you need to work around. You may or may not prefer to have a car at the conference location, which will affect the selection of hotel. In turn, your company may require that you get approval for travel, they may place budget constraints on your travel, or they may mandate that you use particular air carriers. Constraints can be numerous and consequently difficult for another human being to keep in their head and to simultaneously satisfy. Fortunately, the enforcement of complex constraints is something a computer does well.

Our objective is to develop agent technology that will perform these types of tasks automatically by exploiting markup (knowledge representation) of Web services and markup of user constraints. Clearly we cannot simply program an agent to perform these tasks because there is so much variability in how a task is achieved, depending upon an individual's constraints. One approach to addressing the problem would be to exploit AI techniques for planning, and to build an agent that plans a sequence of Web service requests to achieve the goals of the task, predicated on the user's constraints. This is certainly a reasonable approach, but planning is computationally intensive and in this case, we actually know much of what the agent needs to do. We just need the agent to fill in the details and enforce the user's constraints.

We argue that a number of the activities a user may wish to perform on the semantic Web or within some networked service environment in the workplace or at home, can be viewed as customizations of reusable, high-level generic procedures. Our vision is to construct such reusable, high-level generic procedures, and to archive them in sharable 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. The adaptations and extensions described in the sections to follow are designed to address the following desiderata. **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 levels of knowledge. As a consequence, we need to ensure that the program accesses all the knowledge it needs itself, or that knowledge of certain things is a prerequisite to executing the program. Thus we want programs that are self-sufficient.

3 Golog

Golog (e.g., [11; 16; 4]) 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.,[16]), a first-order logical language for reasoning about action and change.

In the situation calculus, the state of the world is expressed in terms of functions and relations (fluents) relativized to a particular situation s, e.g., $f(\vec{x}, s)$. For the purpose of propositions and theorems at the end of this paper, we assume a finite number of fluent symbols, and no functional fluents. A situation s is a history of the primitive actions (e.g., a) performed from an initial, distinguished situation S_0. The function $do(a, s)$ maps a situation and action into a new situation. A situation calculus theory $\mathcal{D}$ comprises the following sets of axioms (See [16] for details):

- domain independent foundational axioms of the situation calculus, Σ,
- successor state axioms, $\mathcal{D}_{SS}$, one for every fluent F in the domain.
- 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, $\mathcal{D}_{S_0}$.
- unique names axioms for actions, $\mathcal{D}_{una}$,
- domain closure axioms for actions, $\mathcal{D}_{dca}$[1]

Note that in our work, we consider both world-altering actions and information-gathering or sensing actions. We follow the work of Levesque and others (e.g., [10]), and represent the effects of sensing actions by including a sensed fluent

[1]Not always necessary, but we will require it in 3.1.

axiom for each primitive action a in our domain, $SF(a,s) \equiv \phi_a(s)$.

Golog (alGOl in LOGic) [11] 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 (see [11; 16; 4] for details).

a — primitive actions
$\phi?$ — tests
$\delta_1; \delta_2$ — sequences
$\delta_1|\delta_2$ — nondeterministic choice of actions
$\pi(x)\delta$ — nondeterministic choice of parameters
if ϕ **then** δ_1 **else** δ_2– conditionals
while ϕ **do** δ — while loops
proc $P(\vec{v})$ δ **endProc** — procedure

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

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 $do(\vec{a}, S_0)$ is an abbreviation for $do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)))$.

3.1 Customizing Golog Programs

In this section we describe an extension to Golog that enables 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 is 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 in our theory, is referred to collectively 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 formulae, C in the situation calculus.

For example, Marielle's personal constraint is that she would like to book an airline ticket from origin, o to destination, d if the driving time between these two locations is greater than 3 hours. We specify this as the following constraint which is included in $\mathcal{D}_{necD}$:

$$Desirable(bookAirTicket(o,d,dt,s)) \supset gt(DriveTime(o,d),3,s)$$

Similarly, Marielle has specified dates she must be at home (her son's birthday, her daughter's hockey game,...) and her constraint is not to be away on those dates. These are include in $\mathcal{D}_{PC}$, e.g.,

$$\neg(Away(dt,s) \wedge MustbeHome(dt,s))$$

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:

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

where $\Omega_A = \omega_1 \vee \ldots \vee \omega_n$, for each ω_i of (1). E.g.,

$$\Omega_{bookAirTicket} = gt(DriveTime(o,d),3,s)$$

$$\Omega_{PC} \equiv \mathcal{R}^*[C(do(A(\vec{x}),s))] \tag{3}$$

where $\mathcal{R}^*$ is repeated regression rewriting (e.g., [16]) of $C(do(A(\vec{x}),s))$, the constraints relativized to $do(a,s)$, using the successor state axioms, $\mathcal{D}_{SS}$ from $\mathcal{D}$. E.g.,

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

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

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

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

From this we determine

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

Having computed $\mathcal{D}_D$, we include it in $\mathcal{D}$. In addition to computing $\mathcal{D}_D$, the set of $Desirable$ fluents, we also modify the computational semantics of our dialect of Golog. In particular, we adopt the *computational semantics* for Golog. (See [4] for details.) Two predicates are used to define the semantics. $Trans(\delta, s, \delta', s')$ is intented 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).

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

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

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 [14]. 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 [12]. There are other types of customizing constraints which we do not address in this paper (e.g., certain temporal constraints). We will address these constraints in future work.

Observe that this approach to defining $Desirable$ can be likewise adopted to incorporate other deontic constraints. For example, just as we have customized our Golog programs to the constraints of particular users in our Web-motivated application, we can similarly, customize our procedures to the nuances and constraints of particular agents, particular services, or particular end effectors, by encoding deontic notions such as $Able$ or $Should$ in a similar fashion and including them in $Legal$.

3.2 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 in order for them to be generic, they must have some nondeterminism and be fairly general in nature, enabling 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 flexibility, and can be overly constraining for the specification of many generic procedures. Consider the program $bookAirTicket(\vec{x}); bookCar(\vec{y})$. The sequence construct dictates that $bookCar(\vec{y})$ must be performed in the situation resulting from performing the action $bookAirTicket(\vec{x})$ and that $Poss(bookCar(\vec{y}), do(bookAirTicket(\vec{x}), s)$ must be true, otherwise the program will fail. Imagine that the precondition $Poss(bookCar(\vec{y}), s)$ 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 dictated sequence of actions for the purposes of achieving preconditions, we define a new construct called *order*, designated by the : connective[2]. 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 $Poss(a_2, do(\vec{a}, do(a_1, s)))$. For the purpose of this paper, we simplify the definition, restricting 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 equivalent to the program

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

It is easy to see that

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

will eventually achieve the precondition for a_2 if they can be achieved.

We extend the computational semantics of Golog as follows to include this construct.

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

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

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

Note that the order construct, ':' 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. We discuss this further in Section 4.1.

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 3.2 by replacing $achieve(Poss(a[s])$ with $achieve(Legal(a[s])$ in Axioms (6) and (7) above, where $Legal(a[s]) \equiv Poss(a[s]) \wedge Desirable(a[s])$, 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 [17]. 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 $do(a_1, s)$.

3.3 Self-Sufficient Golog Programs

Now that our Golog programs are customizable and can be encoded generically, we wish them to be usable. An agent needs sensing actions to execute a Golog program when the agent has incomplete knowledge of the initial situation or when exogenous actions exist that change the world in ways the agent's theory of the world does not predict. In our work, we need to define Golog programs that can be used by a variety of different agents without making assumptions about what they know. 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. That is, we wish to ensure that the agent knows whether actions

[2] Strictly speaking, this is a shorthand for a combination of existing constructs, rather than an extension.

are possible, and it also knows the truth or falsity of the *conditions* of conditional actions (if-then-else, while loops, pick) immediately prior to the situation in which those actions are executed. This can be achieved either by sensing immediately prior to the conditional, sensing earlier and the knowledge persisting, or by the condition being known in the initial situation and the knowledge persisting.

To capture this intuition, we define the notion of a Golog program that is *self-sufficient* with respect to a set of axioms $\mathcal{D}$ and *kernel initial situation*, $\mathcal{D}_{\delta_{S_0}}$. This kernel initial situation can be thought of as a necessary precondition for executing the Golog program. It is the conditions that must be true in S_0 (and that all agents know about). To characterize self-sufficiency, we define a predicate $ssf(\delta, s)$ that characterizes the conditions underwhich a Golog program has sufficient knowledge to be executable in situation s. As with the definition of the $Trans$ and $Final$ predicates, $ssf(\delta, s)$ is defined inductively over the structure of δ as follows.

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

$$ssf(\phi?, s) \equiv \mathbf{KWhether}^3(\phi, s) \tag{9}$$

$$ssf(a, s) \equiv \mathbf{KWhether}(Poss(a[s], s)) \wedge \mathbf{KWhether}(Desirable(a[s], s)) \tag{10}$$

$$ssf(if\ \phi\ then\ \delta_1\ else\ \delta_2, s) \equiv \mathbf{KWhether}(\phi, s) \wedge (\phi[s] \supset ssf(\delta_1, s)) \wedge (\neg\phi[s] \supset ssf(\delta_2, s)) \tag{11}$$

$$ssf(\delta_1; \delta_2, s) \equiv ssf(\delta_1, s) \wedge \forall s'.[Trans(\delta_1, s, nil, s') \supset ssf(\delta_2, s')] \tag{12}$$

$$ssf(\delta_1 | \delta_2, s) \equiv ssf(\delta_1, s) \wedge ssf(\delta_2, s) \tag{13}$$

$$ssf(while\ \phi\ do\ \sigma, s) \equiv \mathbf{KWhether}(\phi, s) \wedge (\phi[s] \supset \forall s'.[Trans(\sigma, s, nil, s') \supset ssf(while\ \phi\ do\ \sigma, s')]) \tag{14}$$

$$ssf(\Pi.x[\phi(x), \sigma], s) \equiv \exists x.[\mathbf{KWhether}(\phi(x), s) \wedge ssf(\sigma, s)] \tag{15}$$

Since : is defined in terms of existing constructs, $ssf(\delta_1 : \delta_2, s)$ follows from (8)–(15) above. We define a Golog program δ to be self-sufficient with respect to a kernel initial situation $D_{\delta_{S_0}}$, effectively the precondition for that program, if $ssf(\delta, S_0)$ is true and additionally that it can terminate given that the kernel initial situation holds in S_0. This second condition is more stringent, and we may choose to relax it at times.

Definition 1 (Self-Sufficient Program) *A Golog program δ is self-sufficient relative to a set of axioms $\mathcal{D}$ and kernel initial state $D_{\delta_{S_0}}$ if $\mathcal{D} \cup \mathcal{D}_{\delta_{S_0}}$ is satisfiable, and*

1. $\mathcal{D} \cup \mathcal{D}_{\delta_{S_0}} \models ssf(\delta, S_0) \equiv true$*, and*
2. $\mathcal{D} \cup \mathcal{D}_{\delta_{S_0}} \models \exists s'.[Trans(\delta, S_0, \sigma, s') \wedge Final(\sigma, s')]$.

[3] $\mathbf{KWhether}(\phi, s)$ is an abbreviation for a formula indicating that the truth value of ϕ is known. (See [19] for details.)

In addition to defining the notion of self-sufficient programs, we also define some simplifying assumptions. In particular, we assume that the fluents we are sensing persist for a reasonable period of time, and that none of the actions we perform in our program, cause this assumption to be violated. Many systems that sense make this assumption, they simply differ on the period of time for which they assume a sensed fluent persists or how drastically it changes when it changes. 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.

Definition 2 (Conditioned-on Fluent) *Fluent C is a conditioned-on fluent in Golog program δ iff for some conditional action A with condition ϕ of the form*

1. **if** ϕ **then** δ_1 **else** δ_2,
2. **while** ϕ **do** δ, *or*
3. $\Pi.x[\phi, \delta]$

C appears in the formula ϕ.

Definition 3 (Invocation and Reasonable Persistence (IRP) Assumption) *We assume that our Golog programs do not experience sensors errors. We say that a Golog program δ adheres to the invocation and reasonable persistence assumption if for every conditioned-on fluent in our Golog program δ,*

1. *If sensing actions have preconditions then they are true in the initial state.*
2. *The knowledge of conditioned-on fluents, once established, persists*[4].

We use this assumption in the development of our Golog interpreter in the section to follow. We also use it to help prove properties of self-sufficient programs and to verify that programs are self-sufficient. Our work in this area is preliminary. Hence, we postpone discussion to a longer version of this paper.

4 Middle-Ground Execution

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., [5; 8; 18]). While De Giacomo and Levesque [5] and Reiter [18] advocate the use of an online interpreter to reason with sensing actions, Lakemeyer [8] suggests the use of an offline interpreter with conditional plans. The trade-off is clear here. 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 or offline interpreter depends on properties

[4] i.e., no subsequent actions in the program change the value of sensed fluents.

of the domain, such as the ones discussed in the previous section. For instance, in a mobile robot domain, an online interpreter seems to be more appropriate because critical aspects of the world change quickly. On the other hand, for contingency planning, an offline interpreter that enables search may be more useful. We argue that for many applications that utilize semantic Web services, there is a middle ground.

Following the intuition in the previous section, we define a middle ground between offline and online execution that exploits both the merits of backtracking and search and the merits of online execution of sensing actions to reduce the search space size. Our interpreter senses online to collect the relevant information needed in the Golog program, while only simulating the effects of world-altering actions. By only simulating rather than executing, the interpreter can backtrack if it finds that a decision it has made does not lead to successful termination of the program. Humans often follow this approach when instantiating their generic procedures. They collect information (e.g., flight schedules, hotel availability) and choose a sequence of actions, relying on the information they have collected persisting until they actually execute the associated world-altering actions (e.g., booking the flight or the hotel). If this persistence assumption fails, they replan at execution time. In our middle-ground interpreter, following successful generation of a terminating situation by our interpreter, the sequence of world-altering actions that comprise the terminating situation can be executed, and will result in successful execution of the Golog program, if our persistence assumptions have not been violated. This is the approach we take right now. Alternately, the generated sequence of world-altering actions could be executed with an online execution monitoring system that potentially re-performs sensing actions to ensure our persistence assumptions have not been violated, and executes the world-altering actions that were finally selected by our middle-ground interpreter. The explicit encoding of search areas in a program, as proposed by [5] through the addition of their Σ search construct, can achieve some of the same functionality as our middle-ground interpreter.

Of course, as we discussed, the invocation and reasonable persistence (IRP) assumption upon which this approach is predicated may not always hold, or it may not hold for all fluents in a Golog program. In such cases, the interpreter we propose in the section to follow can be combined with an interpreter that builds conditional plans, following the approach proposed in [8]. The generation of condition plans can be used for actions whose conditioned-on-fluents do not adhere to the IRP assumption. We don't favour the generation of conditional plans in all cases because they are more computationally intensive, and can lead to the consideration of many irrelevant paths and the generation of large conditional plans.

4.1 The Middle-Ground ConGolog Interpreter

We have modified the ConGolog offline interpreter in [5; 4] to account for user constraints and the order construct. We have also provided a means of encoding the sensed fluent axioms that realizes the strategy for our middle-ground interpreter, described above. We discuss each of the modifications in further detail.

User customizing constraints: We have modified the ConGolog interpreter in [5; 4] to take into consideration personal constraints in a rather straightforward and elegant way. 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.

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 'World Simplest Breath First Planner' (wsbfp) developed by Reiter [16]. We appeal to its simplicity and the soundness and completeness of the A* algorithm. For completeness, we include the PROLOG code for $achieve$ and $wsbfp$.

```
proc(achieve(Goal,N),
   [(Goal)?$wsbfp(Goal, N)$achieve(Goal,N+1)]).
proc(wspbf(Goal, N), [plans(Goal, 0, N)]).
proc(plans(Goal, M, N), [ (M < N)?,
   [actionSequence(M),(Goal)?] $ plans(Goal, M + 1, N)
]).
proc(actionSequence(N), [ (N = 0)? $
  [(N > 0)? ,
   pi(a, [(poss(a))?, a]),
   actionSequence(N-1)
  ] ]).
```

Obviously any planner can be used to accomplish this task. We are currently investigating the effectiveness of other planners (e.g., regression planners).

Sensing Actions: A common approach to incorporating sensing actions into a situation calculus theory is through the use of the distinguished sensed-fluent predicate, SF. $SF(a, s)$ states that action a returns the binary sensing result $true$ when the fluent sensed by a, F is true in the situation s. Assuming each sensing action senses one fluent, we would include one sensed-fluent axiom for each action, $SF(a, s) \equiv F_a(s)$ [16].

To accommodate both backtracking and sensing, we assume that the truth value of $SF(a, s)$ can be determined by executing *an external function call*, denoted by $exec(a, s)$. Under this assumption, the truth value of F_a in s can be determined by making the external function call $exec(a, s)$. Whenever the execution succeeds, F_a is true; otherwise, it is false. This, together with the invocation and reasonable persistence assumption, allows us to write the successor state axiom of F_a in the following form

$$F_a(do(a, s)) \equiv exec(a, s) \tag{16}$$

Observe that guarded action theories [6] are similar to action theories encoded in this way in that they no longer contain the fluent SF. This allows a treatment of sensed fluents as ordinary fluents if the external function call $exec(a, s)$ can be made and its termination code can be incorporated. This is simple to implement in a PROLOG interpreter. Equation (16) is translated into PROLOG as follows

```
holds(F, do(A,S)):- exec(A,S).
```

where F denotes F_a. Thus, we only need to provide the set of rules that call the action A, i.e., for each sensing action A, the theory contains a rule of the form

```
exec(A,S):- execute A ...
```

Notice that the execution of action A is domain dependent, and hence, the above rule will be a part of the situation calculus action theory rather than a part of the Golog interpreter[5].

Theorem 1 *For every self-sufficient Golog program δ with associated axioms $\mathcal{D}$ and kernel initial condition $D_{\delta_{S_0}}$ that adheres to the invocation and reasonable persistence assumption and for all situations S, if $\mathcal{D} \cup \mathcal{D}_{\delta_{S_0}} \vdash_R Do(\delta, S_0, S)$ then there exists a model $\mathcal{M}$ of $\mathcal{D} \cup \mathcal{D}_{\delta_{S_0}}$ such that $\mathcal{M} \models Do(\delta, S_0, S)$, where $\vdash_R$ is proof by our Golog interpreter.*

We note that action theories in this paper are definitional theories in the sense of [16] if the initial situation axioms $\mathcal{D}_{S_0}$ is complete, i.e., for each fluent, it contains a definition. This can be achieved by making the closed-world assumption (CWA), which, since our programs are self-sufficient, is less egregious. Thus, the next proposition follows immediately from the Implementation Theorem of [16].

Proposition 1 *For every self-sufficient Golog program δ with associated axioms $\mathcal{D}$ and kernel initial condition $D_{\delta_{S_0}}$ that adheres to the invocation and reasonable persistence assumption and for all situations S,*

$\mathcal{D} \cup \mathcal{D}_{\delta_{S_0}} \vdash_R Do(\delta, S_0, S)$ *iff* $\mathcal{D} \cup \mathcal{D}^{CWA}_{\delta_{S_0}} \models Do(\delta, S_0, S)$,

where $\vdash_R$ is proof by our Golog interpreter and $\mathcal{D}^{CWA}_{\delta_{S_0}}$ is defined as $\mathcal{D}_{\delta_{S_0}} \cup \{F(S_0) \equiv false \mid$ there exists no definition of F in $\mathcal{D}_{\delta_{S_0}}\}$.

5 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 the previous section. The interpreter was also modified to communicate with the Open Agent Architecture (OAA) agent brokering system [13], to send requests for services, and to relay responses to Golog. Since 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 (W4), to extract 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, world-altering services are not actually executed.

[5]The offline interpreter with the search operator in [5] can also be modified and used for this purpose.

The work described was demonstrated in August, 2000 and we are continuing development. We have built several generic procedures including a travel scenario that books travel (car/air), hotel, local transportation, emails the customer an itinerary, and updates an online expense claim. The same procedure has generated numerous instantiations based on different user customization constraints, highlighting the versatility of our approach to programming the semantic Web. Integration with the semantic markup language DAML is ongoing as the language develops [15].

6 Summary and Related Work

In this paper we addressed the problem of automating Web service composition by *programming* agents to execute services on the Web. Our goal was to develop programs that were easy to use, generic, customizable, and that were usable by a variety of users under varying conditions. We argued that an effective way of programming the semantic Web was through the use of high-level reusable generic procedures and customizing user constraints, and we proposed the logic programming language Golog as providing a natural formalism for this task. To this end, 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 self-sufficient programs that are executable with minimal assumptions about the agent's initial state of knowledge, making them amenable to wide-spread use. These extensions were implemented as modifications to an existing ConGolog interpreter, along with an implementation of sensing actions (for information-gathering services). 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. The ConGolog interpreter communicates with Web services through an agent brokering systems and an HTML information extractor.

In ongoing work, we are extending the scope of $Desirable$. We are developing an execution monitoring system to execute the world-altering actions generated by our middle-ground interpreter. We are evaluating the use of regression-based search to realize the order construct more efficiently. Finally we are proving properties of self-sufficient programs.

Related Golog work was identified in the body of this paper, with the work in [5] being most closely related. Two other agent technologies that address aspects of Web or internet composition and that deserve mention are the work of Waldinger and others at SRI on Web agent technology [21], and the softbot work developed at the University of Washington (e.g., [7]). The latter shares in spirit some of what is proposed here and in [15] through the use of action schemas to describe information-providing and world-altering actions that an agent could use to plan to achieve a goal on the internet.

Acknowledgements

We would like to 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. 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.

References

[1] T. Berners-Lee and M. Fischetti. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. Harper, 1999.

[2] DAML-ONT. http://www.daml.org/2000/10/daml-ont.html, 2000.

[3] DAML+OIL. http://www.daml.org/2000/10/daml-oil, 2000.

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

[5] 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.

[6] G. De Giacomo and H. Levesque. Projection using regression and sensors. In *Proc. of the Sixteen International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 160–165, 1999.

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

[8] 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.

[9] O. Lassila and K. Swick. Resource Description Framework (RDF) model and syntax specification. Technical report, W3C, 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.html.

[10] H. Levesque. What is planning in the presence of sensing? In *AAAI 96*, pages 1139–1146, 1996.

[11] 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.

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

[13] 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.

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

[15] S. McIlraith, T. Son, and H. Zeng. Semantic Web services. In *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, March/April 2001. To appear.

[16] R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems*. Book draft, 2000. http://www.cs.utoronto.ca/ cogrobo.

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

[18] R. Reiter. On knowledge-based programming with sensing in the situation calculus. In *Proc. of the Second International Cognitive Robotics Workshop, Berlin*, 2000.

[19] R. Scherl and H. Levesque. The frame problem and knowledge producing actions. In *Proc. of AAAI*, pages 689–695, 1993.

[20] F. van Harmelen and I. Horrocks. FAQs on OIL: the ontology inference layer. *IEEE Intelligent Systems*, 15(6):3–6, Nov./Dec. 2000.

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