# golog.lua: Towards a Non-Prolog Implementation of Golog for Embedded Systems

**Alexander Ferrein***
The Robotics and Agents Research Lab
University of Cape Town, South Africa
alexander.ferrein@uct.ac.za

### Abstract

Among many approaches to address the high-level decision making problem for autonomous robots and agents, the robot programming and plan language Golog follows a logic-based deliberative approach, and its successors were successfully deployed in a number of robotics applications over the past ten years. Usually, Golog interpreter are implemented in Prolog, which is not available for our target platform, the bi-ped robot platform Nao. In this paper we sketch our novel prototype implementation of a Golog interpreter in the scripting language Lua. With the example of the elevator domain we discuss how the basic action theory is specified and how we implemented fluent regression or backtracking in Lua. One possible advantage of the availability of a Non-Prolog implementation of Golog could be that Golog becomes available on a larger number of platforms, and also becomes more attractive for roboticists outside the Cognitive Robotics community.

## Introduction

To address the problem of high-level decision making for autonomous robots or agents, a number of different robot programming languages have been developed. Each of these follow a particular paradigm or technique how the problem of decision making could be solved. Among them are for instance the Procedural Reasoning System (PRS) (Ingrand et al. 1996), the Saphira architecture (Konolige et al. 1997), Reactive Action Packages (RAP) (Bonasso et al. 1997), or the Reactive Plan Language (RPL) (McDermott 1991) and Structured Reactive Controller (SRC) (Beetz 2001). These approaches mostly follow a reactive paradigm or deploy hierarchical task networks.

The robot programming and plan language Golog, on the other hand, follows a logic-based deliberative approach (Levesque et al. 1997), and its successors were successfully deployed in a number robotics applications over the past ten years. The applications range from service robotics to even robotic soccer applications. Golog was used as the high-level decision-making component on a number of different robot platforms, ranging from the RWI B14/B21 over the Sony Aibo to Lego Mindstorms, and many more tailor-made platforms. During the course of the last decade it was extended with useful features like integrating sensing and exogenous actions (De Giacomo, Levesque, and Sardiña 2001), continuous change (Grosskreutz and Lakemeyer 2003), or decision-theoretic planning (Boutilier et al. 2000) to name just a few. It emerged into an expressive robot programming and plan language and is used in the Cognitive Robotics community.

Golog interpreter are in general implemented in Prolog. This is straight-forward as the semantic of the language constructs is described in the situation calculus (McCarthy 1963), a first order action logic which allows for reasoning about actions and change. The implementation, or better the specification of Golog in Prolog is just a page long, and it was shown that the implementation is correct, having a proper Prolog interpreter (Reiter 2001).

Until now, we did not face any problems to run Golog on our robots (Ferrein and Lakemeyer 2008), though one has to note that it always took some extra computational resources to do the action selection with Golog. However, for our current robot project, it seems that no Prolog interpreter is available. We want to use Golog on the bi-ped robot Nao from French Aldebaran, which is running Open Embedded Linux, for which, to the best of our knowledge, no Prolog system is currently available.

This motivated to start with a reimplementation of Golog in a language different from Prolog. We came up with using the the scripting language Lua (Ierusalimschy, de Figueiredo, and Filho 1999), which we also used for the Behaviour Engine that we are running on the robot (Niemüller, Ferrein, and Lakemeyer 2009). In this paper, we present our prototype implementation of a Vanilla Golog interpreter in Lua. We first briefly introduce the hardware platform and the software system which we are running on the Nao, as this motivates our decisions to try a re-implementation of Golog, and to use Lua for this purpose. Then, we introduce Golog and Lua, and show in some detail the Prolog implementation of Golog, before we give details of our Lua re-implementation. In particular, we show how the basic action theory is specified, how regression or backtracking is

*The author is currently a Feodor Lynen fellow supported by a grant of the Alexander von Humboldt Foundation. This research is also partly supported by the International Bureau of the German Ministry of Education and Resarch with grant no. SUA 07/031.

implemented in our interpreter. As a proof-of-concept we implemented the elevator domain (Levesque et al. 1997; Reiter 2001). We do not have any run-time results or comparisons of our implementation with the Prolog implementation yet, but hope to have them by the time of the Symposium. Finally we conclude with discussing the preliminary state of this work and give an outlook to our future work.

## Our Embedded System: The Nao Robot

### Hardware Platform

In the past, we used and extended Golog for several robotics applications ranging from service robotics to robotic soccer applications (Ferrein and Lakemeyer 2008). We learnt to value the flexibility in modelling the application domain and expressing control knowledge in an elegant way. Our Golog implementation was always based on Prolog, and we could run a Prolog engine on our robots so far. It is worth noting that running Golog on a mobile robot platform requires some extra computational resources. For our latest robotics project, however, no Prolog system, to the best of our knowledge, seems to be available. Moreover are the computational resources of our new mobile robot platform quite restricted.

Currently, we are developing robotic soccer applications for the bi-ped humanoid robot Nao built by French Aldebaran (Aldebaran Robotics 2008). The platform is the successor of the Sony Aibo in Robocup's Standard Platform League. It is a 21 degree-of-freedom humanoid robot about 58 cm tall and is equipped with two VGA resolution cameras, ultrasonic sensors as well as infrared sensors, an inertial measurement unit, tactile sensors and force resistance sensors in the feet. The robot has microphones, loudspeakers, and it has a number of LEDs with which it can display status information. It is powered by an AMD Geode 500 MHz CPU and equipped with 256 MB of memory. Furthermore, it has 1 GB flash memory for hard disk space.

### Software Framework

The programming framework we are using on the Nao is the Fawkes framework. The Fawkes robot software framework (Niemüller 2009) provides the infrastructure to run a number of plug-ins which fulfil specific tasks. Each plug-in consists of one or more threads. The application runs a main loop which is sub-divided into certain stages. Threads can be executed either concurrently or synchronised with a central main loop to operate in one of the stages. All threads registered for the same stage are woken up and run concurrently. The software architecture of Fawkes follows a component-based approach. A component is defined as a binary unit of deployment that implements one or more well-defined interfaces to provide access to an inter-related set of functionality configurable without access to the source code. Components are implemented in Fawkes as a plug-in. For communication between the components we use a blackboard infrastructure which serves well-defined communication interfaces.

Another building block of Fawkes is the use of a Lua-based Behaviour Engine (Niemüller, Ferrein, and Lakemeyer 2009). The idea of this behaviour engine is to provide a behaviour middle-ware between the low-level robot system and high-level decision-making modules. The behaviour engine deploys extended hybrid state machines for monitoring the execution of action patterns or primitive actions. We decided to deploy Lua for the Behaviour Engine, as this scripting language is lightweight with a small memory footprint, though expressive enough for the task. Moreover, Lua showed its potential in a number of successful AI applications so far.[1] The behaviour middle-ware was designed with a high-level decision-making module such as a Golog-based deliberative component in mind. The good experiences with Lua influenced our decision for developing a Lua-based Golog interpreter.

## Situation Calculus

The situation calculus is a first order language with equality which allows for reasoning about actions and their effects. The world evolves from an initial situation due to primitive actions. Possible world histories are represented by sequences of actions. The situation calculus distinguishes three sorts: *actions*, *situations*, and domain dependent *objects*. A special binary function symbol $do : action \times situation \rightarrow situation$ exists, with $do(a, s)$ denoting the situation which arises after performing action $a$ in the situation $s$. The constant $S_0$ denotes the initial situation, i.e. the situation where no actions have yet occurred. The state the world is in is characterised by functions and relations with a situation as their last argument. They are called *functional* and *relational fluents*, respectively. As an example, consider the position of a robot navigating in an office environment. One aspect of the world state is the robot's location $robotLoc(s)$. Suppose the robot is in an office with room number 6214 in the initial situation $S_0$. The robot now travels to office 6215. The position of the robot then changes to $robotLoc(do(goto(6215), S_0)) = 6215$. $goto(6215)$ denotes the robot's action of travelling to office 6215, and the situation the world is in after the action is described by $do(goto(6215), S_0)$.

For each action one has to specify a *precondition axiom* stating under which conditions it is possible to perform the respective action and an *effect axiom* formulating how the action changes the world in terms of the specified fluents. An action precondition axiom has the form $Poss(a(\vec{x}), s) \equiv \Phi(\vec{x}, s)$ where the binary predicate $Poss \subseteq action \times situation$ specifies when an action can be executed, and $\vec{x}$ stands for the arguments of action $a$. For our travel action the precondition axiom may be $Poss(goto(room), s) \equiv robotLoc(s) \neq room$. After having specified when it is physically possible to perform an action it remains to state how the respective action changes the world.

In the situation calculus the effects of actions are formalised by so-called successor state axioms of the form $F(\vec{x}, do(a, s)) \equiv \varphi_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\varphi_F^-(\vec{x}, a, s)$, where $F$ denotes a fluent, $\varphi_F^+$ and $\varphi_F^-$ are formulae describing under which conditions $F$ is true, or false resp. This axiom simply states that $F$ is true after performing action $a$ if $\varphi_F^+$ holds, or the fluent keeps its former value if it was

[1] See http://lua.org for a list of applications.

not made false. Successor state axioms describe Reiter's solution to the frame problem (Reiter 2001), the problem that all the non-effects of an action have to be formalised as well. As an example, consider the following successor state axiom for the fluent $robotLoc(s) : robotLoc(do(a, s)) = y \equiv a = goto(room) \wedge y = room \vee a \neq goto(room) \wedge y = robotLoc(s)$. Note that free variables in the occurring formulae are meant to be implicitly universally quantified. The background theory (also called basic action theory, or BAT for short) is a set of sentences $\mathcal{D}$ consisting of $\mathcal{D} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$, where $\mathcal{D}_{ssa}$ contains sentences about the successor state axioms, $\mathcal{D}_{ap}$ contains the action precondition axioms, $\mathcal{D}_{una}$ states sentences about unique names for actions, and $\mathcal{D}_{S_0}$ consists of axioms stating what holds in the initial situation. Additionally, $\Sigma$ contains a number of foundational axioms defining situations. For details we refer to (Pirri and Reiter 1999; Reiter 2001).

## Golog

The high-level programming language Golog (Levesque et al. 1997) is based on the situation calculus. As planning is known to be computationally very demanding in general, which makes it impractical for deriving complex behaviours with hundreds of actions, Golog finds a compromise between planning and programming. The robot or agent is equipped with a situation calculus background theory. The programmer can specify the behaviour just like in ordinary imperative programming languages but also has the possibility to project actions into the future. The amount of planning (projection) used is in the hand of the programmer. With this, one has a powerful language for specifying the behaviours of a cognitive robot or agent. While the original Golog is well-suited to reason about actions and their effects, it has the drawback that a program has to be evaluated up to the end before the first action can be performed. It might be that the world changed between plan generation and plan execution so that the plan is not appropriate or is invalid.

The original Golog was extended over recent years and has become an expressive robot programming language. Dialects of Golog feature online execution, sensing facilities (De Giacomo, Levesque, and Sardiña 2001), continuous change (Grosskreutz and Lakemeyer 2003), or decision-theoretic planning (Boutilier et al. 2000), to name just a few. Golog and it's derivatives were used in a number of successful cognitive robotics applications such as (Hähnel, Burgard, and Lakemeyer 1998; Levesque and Pagnucco 2000; Soutchanski, Pham, and Mylopoulos 2006; Ferrein and Lakemeyer 2008; Eyerich et al. 2006).

In this paper we approach a re-implementation of the original language design, also called Vanilla Golog. Therefore we describe the specification and its Prolog implementation in the remainder of this section.

### Specification

As stated above is the semantic of the different programming statements defined as situation calculus formulae. Having this formal semantics is very convenient for proving program properties and for program verification. In the following we overview the formal semantics of the different program statements. Programs are expanded into situation calculus formulae with the a predicate $Do(p, s, s')$. If the basic action theory entails the behaviour of the program $p$, it is proven that program $p$ leads to the goal situation $s$ starting from the initial situation $S_0$. As a side effect of the constructive proof one yields an executable Golog program. Formally,

$$\mathcal{D} \models (\exists p, s).Do(p, S_0, s) \wedge Goals(s),$$

meaning that it is entailed that program $p$ leads from the initial situation $S_0$ to some situation $s$. The action sequence $s$ which resulted from interpreting program $p$ fulfils $Goals(s)$. At the end of this section we illustrate this with an example for a Golog elevator controller.

The semantics is defined as follows.

1. *Sequence:* $Do([\delta_1; \delta_2], s, s') \stackrel{def}{=} \exists s''.Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')$.
2. *Test action:* $Do(\varphi?, s, s') \stackrel{def}{=} \varphi[s] \wedge s' = s$. The notation $\varphi[s]$ means that in the formula $\varphi$ the suppressed situation arguments are restored.
3. *Non-deterministic choice of action argument:* $Do((\pi x)\delta(x), s, s') \stackrel{def}{=} \exists x.Do(\delta(x), s, s')$.
4. *Non-deterministic iteration:* $Do(\delta^*, s, s') \stackrel{def}{=} \forall P.(\forall s_1.P(s_1, s_1) \wedge \forall s_1, s_2, s_3.(P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3))) \supset P(s, s')$
5. *Conditional:* $Do(\textbf{if } \varphi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif}, s, s') \stackrel{def}{=} Do([\varphi?; \delta_1 | \neg\varphi?; \delta_2], s, s')$.
6. *Loops:* $Do(\textbf{while } \varphi \textbf{ do } \delta \textbf{ endwhile}, s, s') \stackrel{def}{=} Do(((\varphi?; \delta)^*; \varphi?), s, s')$.
7. *Primitive Action:* $Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s)$. Similar to tests, $a[s]$ means that in action terms the situation argument is restored. In the implementation of Golog this is done via a predicate `restoreSitArg` (see below).
8. *Non-deterministic choice of actions:* $Do((\delta_1 | \delta_2), s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$.

Note that we left out the second-order specification of procedures for space reasons here. It can be found for instance in (Levesque et al. 1997; Reiter 2001).

To access a fluent value in a particular situation we make use of the regression operator $\mathcal{R}$. The situation term of the fluent in the situation in question is regressed to the initial situation by repeatedly applying successor state axioms until $S_0$ is reached and the initial value can be read off from $\mathcal{D}_{S_0}$ for this particular fluent. The fluent value in the situation in question then follows from the regressed formula.

### Prolog Implementation

Golog interpreter are usually based on Prolog, as it is straight-forward to implement the logical situation calculus

specification of the language in a logic programming framework. In the following, we present the implementation of Vanilla Golog.[2] The following Prolog clauses implement the predicate $Do(p, s, s')$ and should easily be understood also by readers not too familiar with Prolog.[3]

1. *Sequence*: each sequence of actions or program statements are evaluated from left to right;

```
do(E1 : E2,S,S1) :- do(E1,S,S2),
  do(E2,S2,S1).
```

2. *Test action*: a test actions evaluates the truth value of a logical formula. The predicate `holds` is introduced below;

```
do(?(P),S,S) :- holds(P,S).
```

3. *Pick*: a variable `V` is non-deterministically chosen and each occurrence of `V` is substituted in program `E` resulting in `E1`;

```
do(pi(V,E),S,S1) :- sub(V,_,E,E1),
  do(E1,S,S1).
```

4. *Star*: implements the non-deterministic repetition of a program;

```
do(star(E),S,S1) :- S1 = S ;
  do(E : star(E),S,S1).
```

5. *Conditional*: if the test on the condition holds, the then-branch is evaluated, otherwise, the else-branch is taken into account;

```
do(if(P,E1,E2),S,S1) :- do((?(P) : E1) #
  (?(-P) : E2),S,S1).
```

6. *Loop*: the star operator is conditioned on `P`;

```
do(while(P,E),S,S1):- do(star(?(P) : E) :
  ?(-P),S,S1).
```

7. *Non-deterministic choice of actions*: either program `E1` or `E2` is evaluated, making use of Prolog's disjunction operator ";"; [4]

```
do(E1 # E2,S,S1) :- do(E1,S,S1) ;
  do(E2,S,S1).
```

8. *Procedure*: for a procedure, it is simply checked if a declaration of a procedure in Prolog's database with the same name exists, if so, the body of the procedure is further evaluated;

```
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
```

9. *Primitive Action*: similar to procedures, it is checked whether the action is declared. Furthermore, it is checked if the precondition axiom `poss` in the current situation holds.

```
do(E,S,do(E,S)) :- primitive_action(E),
  poss(E,S).
```

**Variable Substitution and Regression** The first clause of `sub` addresses the case that the term `T1` is a variable. In this case, the resulting term `T2` is unified with `T1`, the first two arguments can be ignored in this case (denoted by the underscores).

```
sub(_,_,T1,T2) :- var(T1), T2 = T1.
```

The idea of $\texttt{sub}(\texttt{V1}, \texttt{V2}, \texttt{T1}, \texttt{T2})$ is to substitute every occurrence of `V1` in the term `T1` with the value `V2`, resulting in the substitution term `T2`:

```
sub(X1,X2,T1,T2) :- \+ var(T1), T1 = X1,
  T2 = X2.
```

If `X1` is not unifiable with the term `T1` then it must be an argument of the term. Hence, the term `T1` is broken up into the functor (unified with `F`) and the arguments (given by the list `L`). Recursively, each argument is substituted appropriately by calling the predicate `sub_list`. The first clause of `sub_list` specifies the end condition of the recursion.

```
sub(X1,X2,T1,T2) :- \+ T1= X1, T1 =..[F|L1],
  sub_list(X1,X2,L1,L2),T2 =..[F|L2].
sub_list(_,_,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :-
  sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).
```

In order to evaluate logical formulae, the predicate *holds* is needed. Similar to the evaluation of a program, the logical formula is straight-forwardly broken up into its subformulae, which in turn are interpreted by `holds`.

```
holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
% similar clauses for <=>, =>, -&, -v, ...
```

Next, we look at the case for $-$`P`. The negation makes use of Prolog's mechanism of negation by finite failure, i.e. when a goal cannot be proved in a finite number of steps, it is assumed that its negation holds. (`isAtom` is an auxiliary predicate defining how atomic formulae are built.)

```
holds(-P,S) :- isAtom(P), \+ holds(P,S).
```

`some` stands for the logical existential quantifier, and is implemented by evaluating the sentence `P1` after having substituted all occurrences of `V` in `P`.

```
holds(some(V,P),S) :- sub(V,_,P,P1),
  holds(P1,S).
```

The final instance of holds is used to restore situation terms in logical formulae in programs. For example, the last argument of a fluent, i.e. the situation term, is dropped when using the fluent in a program. However, for regressing the fluent, we need to establish the fluent value based on the current situation. Hence, in the specification one has to define two versions of a fluent predicate, one with and one without the situation term argument.

[2] It is based on a version, which was adopted by S. Sardiña to run under SWI-Prolog and is available at http://www.cs.toronto.edu/cogrobo/main/systems/index.html.

[3] Note that the head of the clause is left of the ":-" the body is right of it. When evaluating a clause, the head will be replaced by its body. The clauses can also be read as reverse implications.

[4] Note that the evaluation strategy is as follows: `E1` is evaluated, and only if the goal `do(E1, S, 1)` fails, then `E2` is tried.

```
holds(A,S) :- restoreSitArg(A,S,F), F ;
          \+ restoreSitArg(A,S,F),
          isAtom(A), A.
```

**The Elevator Example** In the following we restate the elevator example from (Levesque et al. 1997). First, we need to define the actions in our basic action theory:

```
primitive_action(turnoff(N)).
primitive_action(up(N)).
poss(up(N),S) :- currentFloor(M,S), M < N.
```

Other primitive actions which are required for the elevator application are actions for going one storey down, opening, and closing the elevator doors, and can also be found in (Reiter 2001). As an example for a control procedure we give the $goFloor(n)$ procedure and the main control procedure $control$.

```
proc(goFloor(N), ?(currentFloor(N))
    # up(N) # down(N)).
proc(control, while(some(n, on(n)),
    serveAfloor) : park).
```

$goFloor$ tests the actual floor, and either chooses the $up$ or $down$ action. Note that Golog here depends on Prolog's backtracking mechanism to choose either to go up or down. The $control$ procedure simply calls the procedure $serveAFloor$ (which we omit here) until there is no more clause instance of the fluent $on(n)$ in the clauses database. The fluent $on(n)$ becomes true, if an elevator call button on storey $n$ was pressed. Initially, the call buttons on storey 3 and 5 are pressed, meaning that the facts $on(3, S_0)$ and $on(5, S_0)$ are added to the database as being valid in $S_0$.

As a final example we want to show the successor state axiom for the fluent $currentFloor$:

```
currentFloor(M,do(A,S)) :- A = up(M) ;
    A = down(M) ; not A = up(N),
    not A = down(N), currentFloor(M,S).
```

$currentFloor(m, do(a, s))$ is true if either the action performed in situation $s$ was $up(m)$ or $down(m)$, otherwise the fluent value of $currentFloor$ remains unchanged. A successful execution of this program leads to the situation, where all buttons are turned off and the elevator is in its parking position, i.e.

$$s^* = ([down(3), turnoff(3), open, close, up(5), turnoff(5), open, close, down(0), open], S_0).$$

## Lua

Lua (Ierusalimschy, de Figueiredo, and Filho 1999) is a scripting language designed to be fast, lightweight, and embeddable into other applications. These features make it particularly interesting for the Nao platform. The whole binary package takes less then 200 KB of storage. When loaded, it takes only a very small amount of RAM. This is particularly important on the constrained Nao platform and the reason Lua was chosen for our Behaviour Engine over other scripting languages that are usually more than an order of magnitude larger (Ierusalimschy, de Figueiredo, and Filho 2007). In an independent comparison Lua has turned out to be one of the fastest interpreted programming languages (Ierusalimschy, de Figueiredo, and Filho 2007; The Debian Project ). Besides that Lua is an elegant, easy-to-learn language (Hirschi 2007) that should allow newcomers to start developing behaviours quickly. Another advantage of Lua is that it can interact easily with C/C++. As most robot software is written in C/C++, there exists an easy way to make Lua available for a particular control software.

Lua is a dynamically typed language, attaching types to variable values. Eight different types are distinguished: *nil*, *boolean*, *number*, *string*, *table*, *function*, *userdata*, and *thread*. For each variable value, its type can be queried.

The central data structure in Lua are tables. Table entries can be addressed by either indices, thus implementing ordinary arrays, or by string names, implementing associative arrays. Table entries can refer to other tables allowing for implementing recursive data types. For example `t["name"] = value1` stores the key-value pair (name, value1) in table t, while `t[9] = value2` stores the value2 at position 9 in array `t`. Special iterators allow access to associative tables and arrays. Note that both index methods can be used for the same table.

Function are first-class types in Lua and can be created at run-time, assigned to a variable, or passed as an argument, or be destroyed. Lua provides proper tail calls and closures to decrease the needed stack size for function calls. Furthermore, Lua offers a special method to modify code at run-time. With the `loadstring()` statement chunks of code (one or more instruction of Lua code is called chunk) can be executed at run-time. This comes in handy to modify code while you are running it.

Lua deploys a register-based virtual machine to run its code. Although it is an interpreted language, a program will be pre-compiled. For code chunks that are created at run-time, the above mentioned `loadstring` function precompiles the chunk at run-time. As the virtual machine is register-based the code size is decreased. Furthermore, Lua uses an efficient mark-and-sweep garbage collection which, for example, frees unused values in associative arrays efficiently. Finally, we want to mention the explicit support for threads and co-routines in the Lua specification, which can be particularly useful for robotics applications.

## Implementing `golog.lua`: A First Approach

In the following we show some details of our prototypical implementation of Golog in Lua. One of the very pleasant features of Prolog is that it is very easy to work with terms and formulae. Creating instances of terms or atoms even at run-time of a program to modify the code is very helpful for dealing with dynamic domains. For example, the initial value of the fluent $on$ in our elevator example above was kept by adding the instances $on(3, S_0)$ and $on(5, S_0)$ to the internal clauses data base as atomic formulae. Unification is built in, substituting variable values comes for free and using list structures is comfortable.

In Lua, these concepts are not directly available. As opposed to terms and lists, Lua has its associative table structures and is good in dealing with string values. In our first implementation of Golog in Lua, we mainly use tables and

strings to implement a function `Do` which interprets Golog programs. On a technical side, note that our implementation resembles more the transition semantics as proposed in ConGolog (De Giacomo, Lesperance, and Levesque 2000), as each interpreted statement is consumed from the input program, leaving the rest program to be interpreted. This is however not a problem as it has been shown that the transition semantics is equivalent to the evaluation semantics that is used by Vanilla Golog, but it requires some special treatment when features such as backtracking are needed. Also, the way we encode action effects is slightly different. We address these topics in the next section.

## Programs and Situation Terms as Nested Tables

In `golog.lua`, a program is a table which is defined in a Lua environment, and the program is run by calling a function `Do(p,s)`

```
prog = {{a_1, {}}, {a_2, {x_1, x_2}},
  {if, {fluent}, {a_3, {}}, {a_4, {}}}}
local s_2, failure = Do(prog, {})
```

The program above consists of an 0-ary action $a_1$ in sequence with $a_2(x_1, x_2)$ and a conditional which, depending on the truth value of *fluent*, chooses $a_3$ or $a_4$, resp. The program is executed with calling the interpreter function `Do` which takes a program and a situation term, and returns the resulting situation after executing the program, or, if the program trace lead to a failure, i.e. the failure variable is true, $s_2$ contains the last possible action. Assuming that *fluent* holds, the resulting execution trace of the `prog` will be

```
s_2 =  {{"a_1",{}},
   {"a_2", {"x_1", "x_2"},{"a_3,{}}}}
```
[5]

We use the empty table or *nil* to represent $S_0$. Therefore, the above situation term has to be interpreted as $do(a_3, do(a_2(x_1, x_2), do(a_1, S_0)))$. Similarly, we represent logical formulae as tables, with the connectives in prefix notation, i.e. $\{\texttt{and}, \phi, \{\texttt{or}, \psi, \theta\}\}$ represents the formula $\phi \wedge (\psi \vee \theta)$.

## Axioms as Tables and Functions

The domain specification and the basic action theory are defined using special associative arrays. Each fluent name in the domain description has to be inserted into the special table `D_fluents`, which, for the elevator example, means:

```
D_fluents=Set{on, currentFloor}
```

`Set` is one of our auxiliary functions to store the values `on` and `currentFloor` in the associative array `D_fluents`. Similarly, we need to keep track of our primitive actions and procedures:

```
D_act = Set{turnoff, open, close, up, down}
D_proc = Set{proc_goFloor, proc_serve,
  proc_park,proc_control}
```

[5]Note that all program statements, actions, and fluent names must be given as strings. For reasons of readability, we omit the quotation marks throughout this paper. Note also that Lua supports to return multiple value, the situation term and the failure condition in this case.

We need these sets to be able to distinguish user-defined actions, procedures, and fluents from Golog keywords when interpreting a program. Next, we show the definition of fluent *on*.

```
on = {["name"]=on, ["arity"] = 1 }

function on.initially(N)
   return {{"3"}, {"5"}}
end
```

For the fluent *on* from our elevator domain, we define a table called `on`. To refer to it in the Golog program, the field `["name"]` needs to be filled, as well as the arity of the fluent. Next, we specify the initial value, i.e. the value in $S_0$. We here use Lua's facility to define unnamed tables. The function returns an associative array with the fields `table["3"]=true` and `table["5"]=true`. The intended meaning is that in the initial situation $on(3, S_0) \equiv on(5, S_0) \equiv \top$. For 0-ary fluents, we would simply return the value `true`.

For defining the effects of an action, the user of the Prolog implementation of Vanilla Golog needs to specify successor state axioms. In our Lua implementation, we use effect axioms similar to the way they were implemented in Indigolog (De Giacomo, Levesque, and Sardiña 2001):

```
function on.turnoff(N, prev_val)
   local list = Retract(tostring(N),
     prev_val[1])
   return prev_val
end
```

Note that `Retract(value, array)` is one of our helper functions that deletes `value` from `array`. This means, to evaluate the value of fluent $f$ in a particular situation $s$ we apply the effect axioms of those actions that are mentioned in the situation term and that change $f$'s value. For example, consider the elevator domain with $s' = ([down(3), turnoff(3), open, close, up(5), turnoff(5), open, close], S_0)$. To evaluate the value of fluent *on* we have to apply the following effect axioms:

```
on.turnoff(5,on.turnoff(3,on.initially(n)))
```

as the actions *up*, *down*, *open*, *close* do not change the value of the fluent *on*. The above string is generated at run-time by the interpreter and Lua's facility to apply code at run-time using the `loadstring()` command and is executed to evaluate the effects of an action. Similarly, we use `loadstring` to check whether precondition axioms or effects axioms are defined. For example, the code fragment

```
local action = "turnoff"
if loadstring("return type(" ..
  action .. ".Poss)")() == "function" then
  ...
else error("Precondition axiom for action
    \%s undefined\\n", action) end
```

checks at run-time whether the precondition axiom for action `turnoff` is defined.[6] In the above example our evaluation routine for checking the effects of action `turnoff`

[6]In our current naive and not optimised proof-of-concept implementation, we check for the axiom each time the action is called.

returns the value *nil*, meaning that no instances of $on$ are currently valid. To be able to evaluate fluent values this way, we follow the convention that the last argument of an effect axiom always takes the value from a successor situation. Another requirement is that all action effects changing a fluent value are defined per action and that the closed world assumption holds. We therefore require that effect axioms are defined as part of the fluent definition. (`on.turnoff` means that the function *turnoff* is defined in the namespace of `on` and can only be used in this namespace.) An example for a precondition axiom is:

```
function turnoff.Poss(N, s)
  return has_fval({on, {N}}, s)
end
```

The action `turnoff(n)` is only possible, if the call button on the respective storey is pressed, i.e iff `(on(n), s) = true`. To access the fluent value the user can apply the function `holds` or `has_fval`, which we address below.

## Holds, Pick, Some and other Variable Substitutions

To evaluate logical formulae, we provide a function `holds(f,s)`, which evaluates if `f` holds in situation `s`. As stated above, we use an prefix notation for logical connectives. We evaluate sub-formulae recursively, just as Golog's Prolog implementation does.

```
function holds(f, s)
  if type(f) == "table" then
    if Member(f[1], binop)  --binary op
    then return holds_binop(f, s)
    ...
end

function holds_binop(f, s)
  local op = table.remove(f, 1)
  local result

  -- traverse formula and evaluate each
element
  if op == "and" then result = true
    while f[1] do
      local eval=holds(table.remove(f,1),s)
      result = result and eval end
   ...
  return result
end
```

We call the respective evaluation function depending on the operator type. The example above shows the evaluation for the operator `and`. More complicated is the implementation of quantifiers. The current reasoning engine in our prototype implementation is somewhat restricted. Existential quantifiers are only allowed in fluent formulae. To this end, we introduce a function `has_fval`, to query fluent formulae.

```
has_fval({"on", {"3"}}, {}) → true
has_fval({"on", {nil}}, {}) → {{"3", "5"}}
```

With the help of `has_fval` it is straight-forward to the operator $some(n, fluent(n))$, which refers to the logical formula $\exists x.fluent(x)$. In the Prolog implementation, the evaluation via the predicate $holds$ is successful, if $fluent(n, s)$ follows from subsequently applying the fluent's successor state axiom on $s$ given its initial value. Similarly, we check with `has_fval(f,s)` if there is an instantiation of `f` in `s` by subsequently applying the effect axioms on `f`. Vanilla Golog also offers the $pi$ operator, which binds the variable in the formula $\exists x.fluent(x)$: $pi(n, ?(fluent(n))$. We omitted the $pi$ operator in our current prototype implementation. We achieve the variable binding with applying $some$ to a fluent, whose arguments are void, i.e. the arguments of the fluent contain nil values (second case of `has_fval(f,s)` above). A more general reasoning engine is subject to future work. One possibility might be to use the constraint system CLIPS (Giarratano and Riley 2004), for which also a Lua interface is available.

Finally we need to address argument substitution to implement call-by-value functionality. This is needed for procedure arguments, but also for the aforementioned case of substitutions for quantifiers. The substitution algorithm for procedures is quite simple, for each argument value, we get the variable name from the procedure prototype, and substitute each occurrence of the variable reference in the procedure body with the value as given in the procedure call. To allow nested procedure calls, we hold the substitutions on each call level on a stack. Similarly, we substitute each occurrence of a variable in a $some$ statement in the subsequent program.

## Executing Actions And Simple Backtracking

As primitive actions need to be declared first, it is easy to distinguish them from other constructs. As we have mentioned above follows our implementation the transition semantics idea of ConGolog. The current program statement is consumed while being interpreted. Hence, it is particularly easy to execute actions immediately by using the `loadstring()` functionality.[7]

One complication of Lua comes with the tail recursion. the closure of variables and global and local variables. Lua supports in general call-by-reference, meaning that you alter the original data object given as an argument, and not a local copy of it. If you need a local copy of a Golog sub-program such as a procedure, you have to iterate through the table representing the sub-program and copy each sub-table to a new table. Although, the programs are not large and Lua is fast with accessing tables, it seems to be overhead. Here, we might need to find a different way to deal with this. This means also that for backtracking as needed for Golog's "#" operator, we need to copy not only the different branches of the non-deterministic choice, but also the situation terms, so that we can determine the correct situation term for the

A more clever way would be to check these things once before executing the program. Note that ".." is the string concatenation operator in Lua.

[7] With adding sensing and exogenous actions to the Lua specifications of the interpreter together with guarded action theories (De Giacomo, Levesque, and Sardiña 2001), it should be quite straight-forward to extend our current implementation to an on-line interpreter.

successful branch.

```
function Do_ndet(prog1, prog2, s)
  local s1, s2 = copy(s), copy(s)
  local p1, p2 =  copy(prog1), copy(prog2)

  local s_res1, fail1 = Do(p1, s1)
  if fail1 then
    local s_res2, fail2 = Do(p2, s2)
     -- branch 2 successful
     if not fail2 then return s_res2,fail2
     -- both branches fail
     else return s, true end
     -- branch 1 successful
  else return s_res1, fail1 end
end
```

Finally, we sketch the implementation of our function `Do`:

```
function Do(program, s1)
  local s2, failure, instr
  repeat
    instr = table.remove(program, 1)
    -- process next instruction
    if type(instr) == "table" then
      -- pop first statement from program
      local statement=table.remove(instr, 1)
      -- process the first instruction
      if statement == nil then return
         s1, true
       -- non-det.  choice
      elseif statement == "#" then
        local ndet_1=table.remove(instr, 1)
        local ndet_2=table.remove(instr, 1)
        s2,failure=Do_ndet(ndet_1,ndet_2,s1)
      ...  -- other statements ...
      else       -- unknown action
        error("Unknown statemet\n"))
        failure = true
      end
    else
      error("Program invalid\n")
      s2 = {}; failure = true
    end
    s1 = s2
    if failure then break end
  until not program[1]
  return s2, failure
end
```

The main loop iterates over the program, popping the first statement from the program and processing it. If there are no more statements in the input program, and no failure occurred, the execution of the program was successful, if at any point during the interpretation of the program a failure occurs, the further execution is immediately terminated.

### The Elevator in Lua

The main control loop for the elevator in Lua looks like:

```
proc_control={["name"]=control,{},
  {{while,{some,{n},{on,{n}}},
    {{proc_serve,{n}}}},{proc_park}}}

proc_serve ={["name"]=proc_serve,{N},
  {{proc_goFloor,{N}},{turnoff,{N}},
    {open,{}}, {close,{}}}}

proc_goFloor={["name"]=proc_goFloor,{N},
  {{#,{#,{?,{{currentFloor,{N}}}},
        {up,{N}}},{down,{N}}}}}
```

As long as there are still instances of the fluent `on`, the procedure `proc_serve` is executed. As we discussed above note that we get a value for the argument $n$ of $some(n)$. The argument for the procedure $proc\_serve(n)$ is also substituted by this value as the procedure is the body of the $while$ instruction. The execution trace of the elevator program in Lua is

```
*** SUCCESS! No (more) solution (lol):
s2={{down, {3}}, {turnoff, {3}}, {open, {}},
  {close,{}}, {up,{5}}, {turnoff, {5}},
  {open,{}},{close,{}},{down,{0}},{open,{}}}
```

leading to the same solution as the Prolog implementation of Vanilla Golog.

## Discussion

Why do we believe this work is useful? Our motivation to begin with this work was the unavailability of a Prolog system on our target platform. The Open Embedded Linux system, to the best of our knowledge, does not offer a Prolog system so far. As we still want to make use of Golog for the high-level decision making of the robot, we need to provide an interpreter by other means. However, as mentioned several times throughout this paper and also the title suggests is the state of this work preliminary. We yet have to show, for the general applicability of this work, that our implementation is competitive with known implementations in Prolog. As for our application on the Nao, we seem to have no other choice than to re-implement Golog. Besides our first results and the fact that the elevator examples works with our interpreter, we have to show that our implementation is correct. Also note that this implementation is naive, and a first quick approach to develop a Golog interpreter in Lua. In particular, we did not yet consider to use meta tables or Lua's closure mechanism for defining the BAT. In future implementations, these features may be taken into account as well as the possibility of integrating Golog language features directly into the Lua specification using Metalua (Fleutot and Tratt 2007), a meta language based on Lua.

As already mentioned, for our future work we need to enhance the reasoning engine and develop an interpreter for online Golog, incorporating features that have proved useful (e.g. cf. (Ferrein and Lakemeyer 2008)). Furthermore, we have to show that our implementation is competitive with the Prolog implementation. Another important issue for the usability of Golog is an easy and neat syntax. The syntax presented here is contributed to the syntax of the associative arrays as provided by Lua. We think that this representation resembles rather an abstracted syntax tree, and should become the representation for the back-end of our new interpreter. The front-end should make use of a regular programming syntax without "lots of silly curly brack-

ets". Here we aim at using the LPEG library which is available for Lua (Medeiro and Ierusalimschy 2008). This package provides interpreting *parsing expression grammars* (PEGs), which could be used to generate the intermediate code, which we presented in this paper. Also, in this step several optimisations can be undertaken to speed up the execution time of a Golog program such as pre-processing loop invariants, or generating tables for often used fluent values, to speed up regression. Another advantage of Lua is the availability of a fast C/C++ interface. It is rather easy to connect Lua with the rest of your robot system. Finally, we aim at using Golog as the standard high-level control language in the Fawkes framework which was recently released (`www.fawkesrobotics.org`). The idea with that is to find a larger robotics community that might be using Golog for encoding control programs.

## References


Aldebaran Robotics. 2008. Website. http://www.aldebaran-robotics.com/.

Beetz, M. 2001. Structured reactive controllers. *Journal of Autonomous Agents and Multi-Agent Systems* 2(4):25–55.

Bonasso, R.; Firby, R.; Gat, E.; Kortenkamp, D.; Miller, D.; and Slack, M. 1997. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artifical Intelligence* 9(2-3):237–256.

Boutilier, C.; Reiter, R.; Soutchanski, M.; and Thrun, S. 2000. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, 355–362. AAAI Press.

De Giacomo, G.; Lesperance, Y.; and Levesque, H. J. 2000. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1-2):109–169.

De Giacomo, G.; Levesque, H.; and Sardiña, S. 2001. Incremental execution of guarded theories. *Computational Logic* 2(4):495–525.

Eyerich, P.; Nebel, B.; Lakemeyer, G.; and Claßen, J. 2006. Golog and pddl: what is the relative expressiveness? In *PCAR '06: Proceedings of the 2006 international symposium on Practical cognitive agents and robots*. ACM.

Ferrein, A., and Lakemeyer, G. 2008. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems, Special Issue on Semantic Knowledge in Robotics* 56(11):980–991.

Fleutot, F., and Tratt, L. 2007. Contrasting compile-time meta-programming in metalua and converge. In *Workshop on Dynamic Languages and Applications*.

Giarratano, J., and Riley, G. 2004. *Expert Systems: Principles and Programming*. Course Technology, fourth edition edition.

Grosskreutz, H., and Lakemeyer, G. 2003. ccgolog – A logical language dealing with continuous change. *Logic Journal of the IGPL* 11(2):179–221.

Hähnel, D.; Burgard, W.; and Lakemeyer, G. 1998. GOLEX - bridging the gap between logic Golog and a real robot. In Herzog, O., and Günter, A., eds., *KI-98: Advances in Artificial Intelligence*, volume 1504 of *Lecture Notes in Computer Science*, 165–176. Springer.

Hirschi, A. 2007. Traveling Light, the Lua Way. *IEEE Software* 24(5):31–38.

Ierusalimschy, R.; de Figueiredo, L. H.; and Filho, W. C. 1999. Lua - An Extensible Extension Language. *Software: Practice and Experience* 26(6):635 – 652.

Ierusalimschy, R.; de Figueiredo, L. H.; and Filho, W. C. 2007. The Evolution of Lua. In *Proceedings of History of Programming Languages III*, 2–1 – 2–26. ACM.

Ingrand, F.; Chatila, R.; Alami, R.; and Rober, F. 1996. PRS: A high level supervision and control language for autonomous mobile robots. In *Proc. ICRA-96*.

Konolige, K.; Myers, K.; Ruspini, E.; and Saffiotti, A. 1997. The Saphira architecture: A design for autonomy. *JETAI* 9(1):215–235.

Levesque, H. J., and Pagnucco, M. 2000. Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings of CogRob-00*.

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. 1997. GOLOG: A Logic Programming Language for Dynamic Domains. *J. of Logic Programming* 31:59–84.

McCarthy, J. 1963. Situations, Actions and Causal Laws. Technical report, Stanford University.

McDermott, D. 1991. A reactive plan language. Technical Report YALEU/DCS-RR-864, Yale University, Department of Computer Science.

Medeiro, S., and Ierusalimschy, R. 2008. A parsing machine for PEGs. In *Proceedings of the 2008 Symposium on Dynamic Languages*, 1–12. ACM.

Niemüller, T.; Ferrein, A.; and Lakemeyer, G. 2009. A lua-based behavior engine for controlling the humanoid robot nao. In *2009 RoboCup Symposium*.

Niemüller, T. 2009. Developing A Behavior Engine for the Fawkes Robot-Control Software and its Adaptation to the Humanoid Platform Nao. Master's thesis, Knowledge-Based Systems Group, RWTH Aachen University.

Pirri, F., and Reiter, R. 1999. Some contributions to the metatheory of the situation calculus. *Journal of the ACM* 46(3):325–361.

Reiter, R. 2001. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Soutchanski, M.; Pham, H.; and Mylopoulos, J. 2006. Decision making in uncertain real-world domains using dtgolog. In *Proc. AAAI-06*.

The Debian Project. The Computer Language Benchmarks Game. http://shootout.alioth.debian.org/. retrieved Jan 30th 2009.