# Action-Based Imperative Programming with YAGI

**Alexander Ferrein**
FH Aachen University of Applied Sciences
Aachen, Germany
ferrein@fh-aachen.de

**Gerald Steinbauer**
Graz University of Technology
Graz, Austria
steinbauer@ist.tugraz.at

**Stavros Vassos**
University of Athens
Athens, Greece
stavrosv@di.uoa.gr

## Abstract

Many tasks for autonomous agents or robots are best described by a specification of the environment and a specification of the available actions the agent or robot can perform. Combining such a specification with the possibility to imperatively program a robot or agent is what we call the action-based imperative programming. One of the most successful such approaches is Golog.

In this paper, we draft a proposal for a new robot programming language YAGI, which is based on the action-based imperative programming paradigm. Our goal is to design a small, portable stand-alone YAGI interpreter. We combine the benefits of a principled domain specification with a clean, small and simple programming language, which does not exploit any side-effects from the implementation language. We discuss general requirements of action-based programming languages and outline YAGI, our action-based language approach which particularly aims at embeddability.

## Introduction

Many tasks for autonomous agents or robots are best described by a specification of the environment and a specification of the available actions the agent or robot can perform. Consider a simple office delivery robot delivering items. The task of the robot is defined by keeping track of the robot's position and the objects the robot should carry around. Furthermore, it needs to know the office layout and the recipient of the items it is delivering. The robot should be able to perform a set of actions like moving to the offices and picking up and putting down objects. Further, it should be able to perform some sensing actions, say, to ensure that an object was really delivered and the reception was acknowledged by the recipient. It is obvious that encoding all this in a fixed pre-programmed robot controller is harder than using a declarative description of the world, describing the actions with their effect and let the controller work out the details.

In the field of cognitive robotics, such controllers are intensively studied and there is a number of well-working approaches. One of the most successful ones is Golog (Levesque et al. 1997) and its descendants (see e.g. (Levesque and Lakemeyer 2008) for an overview of different Golog derivates). Golog applications range from robotic soccer via service robots to gamebots in interactive computer games and webagents (e.g. (Ferrein and Lakemeyer 2008; McIlraith and Son 2002)). All these languages share that they have a formal domain specification and some form of a programming component which could be used to control a robot or agent. We call the combination of a formal domain specification with a programming component *action-based programming*. In the case of Golog, the programming component is based on the imperative programming paradigm, hence we speak of *action-based imperative programming*.

In this paper we present the outline of the language YAGI which stands for *Yet Another Golog Interpreter*. But why would we need yet another Golog interpreter? As experienced Golog programmers having taught Golog in class as well, the shortcomings of Golog's run-time environment for real-world robotics and agent applications become obvious. Run-time systems for Golog are usually implemented in Prolog. The problem is that features of Prolog are implicitly used inside Golog, and the distinction between Golog and Prolog is only obvious to the expert. Students are struggling with that distinction. Another drawback is that it is not easy to integrate a Golog interpreter into one's own agent project. Advanced knowledge about how the interpreter works is required. This, in particular, is holding many potential users back from giving such an action-based programming language a chance in their projects.

While previous related work was concerned with extending the language (e.g. (Levesque and Pagnucco 2000; Giacomo et al. 2009; Boutilier et al. 2000; De Giacomo, Levesque, and Sardiña 2001; Pham 2006; Grosskreutz and Lakemeyer 2003)) or defining a formal semantics with superior properties (Lakemeyer and Levesque 2004; Claßen and Lakemeyer 2006), in this paper we define an easy-to-use new robot programming language following the action-based programming approach. The aim is to define a stand-alone interpreter similar to Golog which moreover integrates useful features from other action-based languages. With YAGI, we want to combine the benefits of a principled domain specification with a clean, small and simple programming language, which does not exploit any side-effects from the embedding host language. The contributions of this paper are the following: (1) we define a (non-exhaustive) list of requirements that categorizes useful features for action-based programming languages; (2) we give the syntax and

semantics for YAGI, a Golog-based interpreter that aims for some of the basic requirements listed in (1) while also providing the ground for meeting more of these requirements in future extensions.

While there exists no implementation yet, the syntax and semantics show that we can express a restricted class of situation calculus basic action theories and Golog-like programs in a uniform language with familiar programming constructs. To the best of our knowledge, it is the first time that a non logic-based language for specifying both a Golog domain and a Golog program is defined in a formal way.

Also, the semantics of YAGI are motivated by recent results showing that progression of basic action theories can be practical under restrictions that are common in application domains, such as the local-effect assumption (Vassos, Lakemeyer, and Levesque 2008). The idea then is that YAGI functions as an "on-line" interpreter, parsing and handling each line of code separately, and progressing, that is, updating, the internal representation after each step. As each line of code may be a code listing that includes the usual nondeterministic features of Golog, the standard regression semantics are used in order to identify an appropriate execution before executing and progressing.

The rest of the paper is organized as follows. In the next section, we formulate a number of general requirements for action-based agent and robot programming languages. We proceed to give some prerequisites for our work, namely the situation calculus and Golog. Then, we discuss the functionality of YAGI using some simple examples, and proceed to introduce the formal syntax and semantics. Finally, we conclude with a discussion on the requirements met by this first version of YAGI, and related and future work.

## General Requirements

In this section we give a list of different requirements for a language following the action-based programming paradigm. It is not expected that a particular language satisfies all the listed requirements as some may be competing or incompatible. The idea is that this list provides a formal ground for discussing and comparing action-based imperative-deliberative programming languages, including YAGI, the language we introduce in the next sections.

### Non-functional requirements

Q1: *Familiarity*: the language is based on concepts that are widely used in software programming frameworks such as variables, functions/methods, and objects.

Q2: *Embeddedness*: a program can be easily embedded into the familiar software programming frameworks such as Java and C++.

Q3: *Interoperability*: a program of the language can easily exchange information in a standard format or through familiar software programming constructs.

Q4: *Transparency*: no concepts of the underlying semantics, e.g., fluents or states, are apparent in the language unless they account for a practical requirement.

### Functional requirements

F1: *Action-based*: the language features constructs that allow the specification of a set of available actions along with their preconditions and effects.

F2: *Imperative*: the language features constructs known from general-purpose imperative programming such as loops and conditionals.

F3: *Goal-oriented*: the language features constructs that allow the specification of deliberative execution that follows a planning approach for achieving a specified goal.

F4: *Arithmetic*: the language features constructs that handle the four basic binary operations of arithmetic, i.e., addition, subtraction, multiplication, division.

F5: *Projection*: the language features constructs for specifying *projection conditional expressions* that test whether a condition is true after the execution of a sequence of actions.

F6: *Queries*: the language features constructs for specifying queries. These are similar to conditional expressions but instead of testing whether the condition is true, they identify all the values of an included variable for which the condition is true.

F7: *Null values*: the language features constructs for specifying unknown values for variables.

F8: *Probabilistic values*: the language features constructs for specifying that a variable value is supported with a degree of probability.

F9: *Disjunctive values*: the language features constructs for specifying that the value of a variable is not known but can only be one of a list of possibilities.

F10: *Interval-based values*: the language features constructs for specifying that the value of a variable is not known but can only be one of the values included in an interval.

F11: *Sensing:* the language features an account to perform knowledge-gathering actions (active sensing) or supports some form of updating sensor values in the background (passive sensing).

F12: *Decision-theoretic*: the language features constructs for allowing the specification of deliberative execution that follows a decision-theoretic approach.

## Situation calculus prerequisites

The situation calculus is a second order language with equality which allows for reasoning about actions and their effects (McCarthy and Hayes 1969; Reiter 2001). The world evolves from an initial situation due to primitive actions. A possible world history, which is simply a sequence of actions, is represented by a first-order term called a *situation*. The constant $S_0$ is used to denote the *initial situation* and a distinguished binary function symbol $do(a, s)$ is used to denote the successor situation to $s$ resulting from performing action $a$. The changing properties of the world are represented with *fluents*, that is, relations with a situation term as their last argument and whose truth value vary from situation to situation. There is also a special predicate $Poss(a, s)$ used to state that action $a$ is executable in situation $s$.

Within this language, one can specify action theories that describe how the world changes as the result of the available actions. A *basic action theory* $\mathcal{D}$ has the following form (Reiter 2001): $\mathcal{D} = \Sigma \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$.[1]

- $\Sigma$ is a set of domain independent foundational axioms which formally define legal situations.
- $\mathcal{D}_{ssa}$ is a set of successor state axioms (SSAs), one for each fluent symbol $F$, of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$. SSAs specify the conditions under which the fluent holds at situation $do(a, s)$ as a function of $s$.
- $\mathcal{D}_{ap}$ is a set of action precondition axioms, one per action symbol $A$, of the form $Poss(A(\vec{y}), s) \equiv \Pi_A(\vec{y}, s)$.
- $\mathcal{D}_{una}$ holds unique-names axioms for actions: $A(\vec{x}) \neq A'(\vec{y})$, and $A(\vec{x}) = A(\vec{y}) \supset \vec{x} = \vec{y}$, for each pair of distinct action symbols $A$ and $A'$.
- $\mathcal{D}_{S_0}$ describes the initial situation.

On top of these action theories, logic-based programming languages can be defined, which, in addition to the primitive actions of the situation calculus, allow the definition of complex actions. Golog (Levesque et al. 1997), the first situation calculus agent language, offers all the control structures known from conventional programming languages (e.g., sequence, iteration, conditional, etc) plus some nondeterministic constructs. It is due to these last control structures that programs do not stand for complete solutions, but only for sketches of them whose gaps have to be filled later, usually at execution time.

| | |
|---|---|
| $\alpha$, | primitive action |
| $\phi?$, | wait or test for a condition |
| $\delta_1; \delta_2$, | sequence |
| $\delta_1 \mid \delta_2$, | nondeterministic branch |
| $\pi\, x.\, \delta(x)$, | nondeterministic choice of argument |
| $\delta^*$, | nondeterministic iteration |
| **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$ **endIf**, | conditional |
| **while** $\phi$ **do** $\delta$ **endWhile**, | while loop |
| $p(\vec{\theta})$. | procedure call |

The execution of Golog programs is formalized using the special predicate $Do(\delta, s, s')$, which says that program $\delta$ when executed in situation $s$ has $s'$ as a legal terminating situation. Finding a legal execution for program $\delta$ then amounts to finding a sequence of actions $\vec{a}$ such that $\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$.

We now proceed to introduce YAGI.

## YAGI by example

The basic ingredients of YAGI are the constructs `fluent` and `action`. Before we introduce formally the syntax and semantics of YAGI, we will go over the basic use of the language using the simple elevator example from (Reiter 2001).

A fluent is similar to an *associative array* in that it associates multi-dimensional keys to a value. A fluent associates every *key* with a *flat list of single values* of either integer or string type. In the simplest case when the key has 0 dimensions, a fluent is very similar to a variable with weak typing. For example, in the following YAGI code the 0-dimensional fluent `currFloor` is declared and a singleton set is assigned to it's 0-dimensional key:

```
fluent currFloor;
currFloor = {4};
```

This is intended to hold the current location of the elevator. Similarly, `on` holds information about the floors where the elevator button is pressed.

```
fluent on;
on = {3, 5};
```

The `action` construct is more like a function definition with some structured parts that resemble a class definition. The interoperability between YAGI programs and the host programming environment is achieved using string signals. The following YAGI code declares the action of turning off the button of the elevator at the given floor.

```
action turnoff($n)
precondition:
  $n in on
effect:
  on -= {$n};
signal:
 "Turn-off button at floor " + $n
end action
```

Note that the arguments of the action are essentially local variables that can be used in the specification of the action. Local variables can be introduced by other constructs, too, as we will see later. They hold a single value of integer or string type, and always start with the dollar sign so as to distinguish them from fluents and facts, i.e., static fluents.

Except for normal assignment for fluents, YAGI features special operators with set-operational semantics such as `+=` and `-=`. For example in the effect above, `on` gets assigned the original set of values minus the value of the variable `$n`.

The following YAGI code declares the action of the elevator moving up to go to the given floor.

```
fact floors;
floors = {1, 2, 3, 4, 5, 6};
action up($n)
precondition:
  exists $i in floors
  such currFloor == {$i} and $i < $n
effect:
  currFloor = {$n};
signal:
  "Move up to floor " + $n
end action
```

Note that another local variable was introduced in the `exists` block, which also required a set that specifies its range. In the precondition we have two different comparisons. The left equivalence is set-based denoted by the brackets around the variable. The right comparison is an odinary value-value comparison. Action `down($n)` is declared in a similar way.

The elevator example also includes two actions with no precondition and effects that signal the opening and closing

[1] For readability we typically omit leading universal quantifiers.

of the elevator door. These are declared in YAGI simply as follows for the opening action and similarly for closing.

```
action open()
signal:
  "Open elevator door"
end action
```

Except for primitive actions of this form, the elevator example also specifies complex actions as procedures. The next one is the action of moving to a given floor by either going up or down.

```
proc goto($n)
  if currFloor != {$n} then
    choose
      up($n);
    or
      down($n);
    end choose
  end if
end proc
```

In this complex action, `choose` specifies a nondeterministic choice between alternatives.

The next complex action uses the other actions to specify how the elevator serves a given floor.

```
proc serve($n)
  goto($n);
  turnoff($n);
  open();
  close();
end proc
```

Finally, the next one is the main controller of the elevator that nondeterministically picks one floor where the button is pressed and serves it.

```
proc serveafloor()
  pick $n from on such
    serve($n);
  end pick
end proc
```

The YAGI code we have seen essentially declares elements of the YAGI world of fluents. The first few lines describe the current state of the world, while the rest specify primitive and complex actions. Now we will see how one interacts with a stand-alone YAGI interpreter. Assuming that the YAGI code we discussed is stored in "elevator.y", we invoke the interpreter, import the file, and execute some code.

```
YAGI>> import("elevator.y");
YAGI>> currentFloor
{4}
YAGI>> up(5);
"Move up to floor 5"
YAGI>> currentFloor
{5}
YAGI>>
```

For every *line* of YAGI code we execute, the interpreter *searches* for an appropriate realization of the code specified and then *executes* the corresponding primitive actions. As far as searching is concerned there is no difference with the Golog interpreter specified in (Reiter 2001) (or subsequent implementations). Nonetheless, the difference with YAGI is that after a sequence of actions is found, the underlying basic action theory that specifies the world of fluents gets *updated* in order to reflect the actions that have been performed.

So we can choose how the interpreter deliberates about finding which actions to execute by *grouping the code into lines* appropriately. The next example illustrates this.

```
YAGI>> import("elevator.y");
YAGI>> serveafloor();
"Move up to floor 5"
"Turn-off button at floor 5"
"Open elevator door"
"Close elevator door"
YAGI>> serveafloor();
"Move down to floor 3"
"Turn-off button at floor 3"
"Open elevator door"
"Close elevator door"
YAGI>>
```

Note that for each floor served, the interpreter *deliberated separately* about which actions to perform. Alternatively, the following YAGI code would require that a viable plan for serving *two floors* is found before execution takes place:

```
YAGI>> serveafloor(); serveafloor();
```

Finally, we can use declarations and assignments to change the current specification of the YAGI world of fluents.

```
YAGI>> import("elevator.y");
YAGI>> currentFloor
{4}
YAGI>> up(5);
"Move up to floor 5"
YAGI>> on += {1};
YAGI>> on -= {5};
YAGI>> on
{1, 3}
YAGI>>
```

This is not sensing in that an unknown fluent may be sensed to its actual value, and probably relates more to the so-called exogenous events. In any case, this can be useful for "altering" the value of fluents according to a passive or active use of sensors that constantly provide a feed of values.

## The syntax of YAGI

We now define the syntax of YAGI using BNF (Backus-Naur Form) while also using $\{\ldots\}$ to denote zero or more repetitions and $[\ldots]$ to denote zero or one repetition. The following BNF shows how to declare fluents and facts, and the syntax of basic expressions which evaluate to a single value or a set of values. Lexical $\langle ID \rangle$ tokens correspond to appropriate names for fluents, facts, and variables, while $\langle INT \rangle$,$\langle STRING \rangle$ correspond to values of that type.

$\langle setexpr \rangle$ ::= $\langle set \rangle$ { ('+' | '−') $\langle set \rangle$ }

⟨*set*⟩ ::= '{' ⟨*value*⟩ { ',' ⟨*value*⟩ } '}'
| ⟨*term*⟩
⟨*term*⟩ ::= ⟨*ID*⟩ { '[' ⟨*value*⟩ ']' }
⟨*valexpr*⟩ ::= ⟨*value*⟩ { ('+' | '−') ⟨*value*⟩ }
⟨*value*⟩ ::= ⟨*INT*⟩
| ⟨*STRING*⟩
| ⟨*var*⟩
⟨*var*⟩ ::= '$' ⟨*ID*⟩
⟨*fluent_decl*⟩ ::= 'fluent' ⟨*ID*⟩ {'['']'}
⟨*fact_decl*⟩ ::= 'fact' ⟨*ID*⟩ {'['']'}

The next BNF shows how logical expressions are built based on variables, terms, and logical connectives.

⟨*formula*⟩ ::= atom
| 'not' '(' ⟨*formula*⟩ ')'
| '(' ⟨*atom*⟩ ('and' | 'or') ⟨*formula*⟩ ')'
| 'exists' ⟨*var*⟩ 'in' ⟨*setexpr*⟩)
'such' ⟨*formula*⟩
| 'all' ⟨*var*⟩ 'in' ⟨*setexpr*⟩)
'such' ⟨*formula*⟩
⟨*atom*⟩ ::= ⟨*valexpr*⟩ ⟨*comp_op*⟩ ⟨*valexpr*⟩
| ⟨*setexpr*⟩ ⟨*comp_op*⟩ ⟨*setexpr*⟩
| ⟨*value*⟩ 'in' ⟨*setexpr*⟩
| ('true' | 'false')
⟨*comp_op*⟩ ::= '==' | '!=' | '<=' | '>=' | '<' | '>'

The next BNF shows the basic assignment statement of YAGI. Sets may be assigned to fluents and facts (indicated as terms), while variables may only hold a single value.

⟨*assign*⟩ ::= ⟨*term*⟩ ⟨*assign_op*⟩ ⟨*setexpr*⟩
| ⟨*var*⟩ ⟨*assign_op*⟩ ⟨*valexpr*⟩
⟨*assign_op*⟩ ::= '=' | '+=' | '−='

The basic assignment along with conditional statements and for-loops form blocks that are convenient for specifying the value of fluents.

⟨*assignment*⟩ ::= ⟨*assign*⟩ ';'
| ⟨*for_loop_assign*⟩
| ⟨*conditional_assign*⟩
⟨*for_loop_assign*⟩ ::= 'for' ⟨*var*⟩ 'in' ⟨*setexpr*⟩ 'do' ⟨*assignment*⟩ 'end for'
⟨*conditional_assign*⟩ ::= 'if' ⟨*formula*⟩ 'then' ⟨*assignment*⟩ [ 'else' ⟨*assignment*⟩ ] 'end if'

These assignment blocks are used to specify the effects of actions in the declaration of actions, as well as the current state of the YAGI world of fluents, as we will see shortly.

⟨*action_decl*⟩ ::= 'action' ⟨*ID*⟩ '('⟨*varlist*⟩')'
'precondition:' ⟨*formula*⟩
'effect:' { ⟨*assignment*⟩ }
'signal:' ⟨*valexpr*⟩ 'end action'
⟨*varlist*⟩ ::= ⟨*var*⟩ { ',' ⟨*var*⟩ }

The YAGI statements follow closely the Golog paradigm, including a construct for action execution, conditional control flow, testing whether a formula holds, performing a for-loop, as well as a construct for nondeterministic control flow and picking the value of a local variable from a set.

⟨*block*⟩ ::= { ⟨*statement*⟩ }
⟨*statement*⟩ ::= ⟨*action_exec*⟩ ';'
| ⟨*pick*⟩
| ⟨*test*⟩ ';'
| ⟨*for_loop*⟩
| ⟨*if_then_else*⟩
| ⟨*choose*⟩
⟨*action_exec*⟩ ::= ⟨*ID*⟩ '(' ⟨*arglist*⟩ ')'
⟨*arglist*⟩ ::= ⟨*value*⟩ { ',' ⟨*value*⟩ }
⟨*pick*⟩ ::= 'pick' ⟨*var*⟩ 'from' ⟨*setexpr*⟩ 'such' ⟨*block*⟩ 'end pick'
⟨*test*⟩ ::= 'test' ⟨*formula*⟩ ';'
⟨*for_loop*⟩ ::= 'for' ⟨*var*⟩ 'in' ⟨*setexpr*⟩ 'do' ⟨*block*⟩ 'end for'
⟨*if_then_else*⟩ ::= 'if' ⟨*formula*⟩ 'then' ⟨*block*⟩ ['else' ⟨*block*⟩ ] 'end if'
⟨*choose*⟩ ::= 'choose' ⟨*block*⟩ { 'or' ⟨*block*⟩ } 'end choose'
⟨*search*⟩ ::= 'search' ⟨*block*⟩ 'end search'

Finally, the next BNF shows procedure declaration and the definition of a line of code in YAGI.

⟨*line*⟩ ::= ⟨*declaration*⟩ | { ⟨*statement*⟩ }
⟨*declaration*⟩ ::= ⟨*fluent_decl*⟩ ';'
| ⟨*fact_decl*⟩ ';'
| ⟨*action_decl*⟩
| ⟨*proc_decl*⟩
| ⟨*assignment*⟩
⟨*proc_decl*⟩ ::= 'proc' ⟨*ID*⟩ '(' ⟨*varlist*⟩ ')' ⟨*block*⟩ 'end proc'

A working Java parser for YAGI that follows this BNF can be found at the Google code project for YAGI: `https://code.google.com/p/yagi/`. Note that this parser currently checks only for syntactical conformance.

## The semantics of YAGI

In this section we specify situation calculus semantics for well-formed YAGI code listings. The stand-alone YAGI interpreter (or a virtual machine that is embedded in some programming environment) operates as a *persistent object* that:

(i) holds a basic action theory that describes the YAGI world of fluents and a set of Golog procedures that describe the YAGI procedures;

(ii) responds to the execution of a YAGI declaration by *updating* the logical specification of the action theory and the procedures;

(iii) responds to the execution of a line of YAGI statements by *finding* an appropriate sequence of actions to perform as specified by Golog semantics, *progressing* the current state of the world, and *producing* signals.

In order to achieve this formalization we introduce a variant of the basic action theories that can represent the features of YAGI. In particular, a YAGI-BAT is defined as follows:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{pres} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{unc} \cup \mathcal{D}_{S_0}$$

where $\mathcal{D}_{pres}$ is the set of axioms of Presburger arithmetic (Enderton 1972), and $\mathcal{D}_{unc}$ is the set of unique-names axioms for some special constants that will be used to represent the string tokens of YAGI. We will now show how a YAGI code listing specifies a YAGI-BAT that has complete knowledge in the initial situation $S_0$ and range-restricted effects in the sense of (Vassos and Sardina 2011).

For simplicity, in our analysis we will omit the string signals that the interpreter produces, and we also assume that

each YAGI line of code does not raise any run-time errors. Consider a function *YAGISem* that takes as input a sequence of $\langle line \rangle$ instances $\langle l_1, \ldots, l_n \rangle$ and returns a theory $\mathcal{D}'$ over language $\mathcal{L}'$, defined recursively as follows.

The base case is when the interpreter is initialized. In this case we assume that *YAGISem*$(\langle\rangle)$ is a theory $\mathcal{D}'$ over a situation calculus language $\mathcal{L}'$ that includes no fluent, action, and constant symbols (except $S_0$), and such that $\mathcal{D}_{ssa}$, $\mathcal{D}_{ap}$, $\mathcal{D}_{una}$, and $\mathcal{D}_{unc}$ are empty.

For any arbitrary sequence of $\langle line \rangle$ instances $\langle l_1, \ldots, l_n \rangle$, *YAGISem*$(\langle l_1, \ldots, l_n \rangle)$ is the theory $\mathcal{D}'$ over $\mathcal{L}'$ that is composed by $\mathcal{D} =$ *YAGISem*$(\langle l_1, \ldots, l_{n-1} \rangle)$ over $\mathcal{L}$, as follows.

- If $l_n$ is a $\langle$*fluent_decl*$\rangle$ instance for an associative array $F$ with $m$ dimensions, then $\mathcal{L}'$ is $\mathcal{L}$ plus the $(m+1)$-ary fluent $F$, and $\mathcal{D}'$ is the same as $\mathcal{D}$ except that (i) all sentences that mention $F$ in $\mathcal{D}_0, \mathcal{D}_{ssa}$ are removed, (ii) the closure axiom $\forall \vec{x} \forall y. F(\vec{x}, y, S_0) \equiv false$ is added to $\mathcal{D}_0$, and (iii) the axiom $F(\vec{x}, y, do(a, s)) \equiv F(\vec{x}, y, s)$ is added to $\mathcal{D}_{ssa}$. Essentially, this declaration "initializes" $F$ to hold no information. The case of $\langle$*fact_decl*$\rangle$ is similar.
- If $l_n$ is an $\langle$*assignment*$\rangle$ instance, we specify $\mathcal{D}'$ and $\mathcal{L}'$ by induction on the construction of the line $l_n$. We show only how the base case of $\langle$*assign*$\rangle$ is treated, as this forms the main intuition, and the rest of the details follow similarly.

  For $\langle$*assign*$\rangle$ that assigns the set of integers $\{v_1, \ldots, v_k\}$ to key $\vec{c}$ for $m$-dimensional array $F$, theory $\mathcal{D}'$ is the same as $\mathcal{D}$ except that (i) sentences in $\mathcal{D}_0$ mentioning $F$ are removed, and (ii) the following closure axiom is added to $\mathcal{D}_0$:$\forall \vec{x} \forall y. F(\vec{x}, y, S_0) \equiv \bigvee_{i=1}^{k} (\vec{x} = \vec{c} \wedge y = v_i^*)$, where $v_i^*$ is the term of Presburger arithmetic for integer $v_i$.

  The case is similar with string values. In this case each string is represented as a constant with a name that is identical to the string, and the appropriate unique-name axioms for string constants are also added to $\mathcal{D}_{unc}$.
- If $l_n$ is an $\langle$*action_decl*$\rangle$ instance for an action $A$ with $m$ parameters, then $\mathcal{L}'$ is $\mathcal{L}$ plus the $m$-ary action symbol $A$, and $\mathcal{D}'$ is the same as $\mathcal{D}$ except that (i) all sentences that mention $A$ in $\mathcal{D}_{ap}, \mathcal{D}_{ssa}$ are removed, (ii) the action precondition axiom $Poss(A(\vec{y}), s) \equiv \Pi_A(\vec{y}, s)$ is added to $\mathcal{D}_{ap}$, where $\Pi_A(\vec{y}, s)$ is constructed by the corresponding $\langle$*formula*$\rangle$ instance in the obvious way, and (iii) the successor state axioms in $\mathcal{D}_{ssa}$ are updated with respect to the effect axioms specified by the $\langle$*assignment*$\rangle$ instances in the effect part of the action declaration.

  For each $\langle$*assignment*$\rangle$ instance, an effect axiom is specified by induction on the construction of $\langle$*assignment*$\rangle$. We show only how the base case of $\langle$*assign*$\rangle$ is treated, and the rest of the details follow similarly. For $\langle$*assign*$\rangle$ that assigns the set of integers $\{v_1, \ldots, v_k\}$ to key $\vec{c}$ for the $m$-dimensional array $F$, the following positive effect axioms is constructed: $\bigvee_{i=1}^{k} (\vec{x} = \vec{c} \wedge z = v_i^*) \supset F(\vec{x}, z, do(A(\vec{y}), s))$, and the negative axiom: $\vec{x} = \vec{c} \wedge F(\vec{x}, z, s) \wedge \bigwedge_{i=1}^{k} (z \neq v_i^*) \supset \neg F(\vec{x}, z, do(A(\vec{y}), s))$, where $v_i^*$ is a term of Presburger arithmetic as above. Note that the interpreter requires that only parameters of the action may be used as arguments for the $\langle$*term*$\rangle$ instance in $\langle$*assign*$\rangle$, which implies that $\vec{x}, z$ are included in $\vec{y}$ (or are bound to constants in the action definition or the fluent definition in the $\mathcal{D}_{S_0}$), imposing a range-restricted assumption (Vassos and Sardina 2011).
- If $l_n$ is a $\langle$*proc_decl*$\rangle$ instance, then $\mathcal{D}, \mathcal{L}$ remain unchanged and a procedure is added to the set of Golog procedures (replacing the previous description if there was one). Building the procedure is straightforward as YAGI statements are weaker than Golog statements and can be easily expressed using Golog constructs. Note also that similarly to Golog, procedures are treated as macros that expand into complex actions in a call-by-value manner.
- If $l_n$ is a sequence of $\langle$*statement*$\rangle$ instances, then $\mathcal{L}' = \mathcal{L}$ and $\mathcal{D}'$ is the same as $\mathcal{D}$ except that $\mathcal{D}_0$ is replaced by a progression of $\mathcal{D}_0$ with respect to an appropriate sequence of actions $\vec{\alpha}$. This sequence is specified by the outcome of the following entailment question: $\mathcal{D} \models \exists s Do(\delta, S_0, s)$,[2] where $\delta$ is the Golog program that corresponds to $l_n$.

  If the entailment holds then $\vec{\alpha}$ is extracted from the proof procedure. Note that progression is always feasible and the new $\mathcal{D}_0$ consists of axioms with the same structure as $\mathcal{D}_0$ (i.e., of the form $F(\vec{x}, y, S_0) \equiv \phi$). This follows from the fact that $\mathcal{D}$ is a special case of the theories examined in (Vassos and Sardina 2011), except for the special treatment of string constants and arithmetic terms.

  If the entailment question does not hold, then $\mathcal{D}'$ is $\mathcal{D}$ (and a signal indicating failure is produced).

## Meeting the requirements

In this section, we will briefly summarize which of the formal and non-formal requirements that we proposed in the beginning are met by our first version of YAGI and which of them will be met by the implementation that we envision, as well as future versions. We start with the non-functional requirements.

One of the main motivations for YAGI's syntax was to enable non-experts in the logical framework of the situation calculus or Golog to intuitively program robot controllers. In the approach taken here we focused on an intuitive syntax that unlike Golog hides details of the underlying theory. Therefore, we meet the requirements of *familiarity* (Q1) and *transparency* (Q4).

As for the requirement of *embeddedness* (Q2) we can only mention our future plans at the current stage, as we still have no implementation in place. However, we aim at developing a platform-independent run-time system that is as easy embeddable following the LUA or Java run-time system. This

[2] Note that YAGI has been specified so that the decidability of this entailment can be guaranteed. This follows by the following design choices: (i) there is complete knowledge in the initial situation, (ii) only bounded loops are allowed in the complex actions of YAGI, (iii) procedures are handled as macros that follow a call-by-value manner, (iv) YAGI includes a very simple notion of arithmetic that only uses equality and addition, and by means of (i) results in (Reiter 2001) that show that $\exists s Do(\delta, S_0, s)$ is equivalent to a regressable sentence for Golog programs that use procedures as macros (a first-order sentence in our case that only bounded loops are allowed), (ii) the decidability of Presburger arithmetic.

is different than the available Golog run-time systems which heavily depend on Prolog.

Similarly, in terms of *interoperability* (Q3), we envision that YAGI could interoperate with usual programming languages using a fixed set of basic types for variables that can be shared in a similar way that remote objects work in Common Object Request Broker Architecture (CORBA). Nonetheless, in this first version of YAGI we only include a generic type of interaction that is expressed with the `signal:` keyword. This is included in the action specification and allows the possibility to call operating/agent systems calls directly. The arguments of the signal slot will be handed over directly to the operating system. This is quite similar to available Golog run-time systems where such operating system calls are handed over via Prolog's C++ interface to the operating system.

As far as the functional requirements are concerned, the first version of YAGI that we present here provides a partial account for the most basic requirements F1, F2, F3, and F4. We intend to deal with the rest of the requirements in subsequent versions of YAGI, focusing first on requirements F5, F6, F7, F11 and later on the more advanced F8, F9, F10, F12. Note that the initial version of YAGI that we present here is indeed limited to some basic programming constructs and functionality, e.g., not including unbounded loops, in order to ease the process of implementing a robust first version of the system, which is our next immediate goal.

## Related work

A number of related works to ours exist. There is for one, a large number of extensions of Golog such as (Levesque and Pagnucco 2000; Giacomo et al. 2009; Boutilier et al. 2000; De Giacomo, Levesque, and Sardiña 2001; Pham 2006; Grosskreutz and Lakemeyer 2003) or (Lakemeyer and Levesque 2004; Claßen and Lakemeyer 2006). Recently, some work went into alternative Golog run-time systems (Ferrein 2010) or into approaches for embedding Golog (Ferrein and Steinbauer 2010).

Our first version of YAGI focuses on a set of features that relate to the basic specification of Golog (Levesque et al. 1997) and the on-line execution mode of Indi-Golog (De Giacomo, Levesque, and Sardiña 2001). As we aimed for a syntax and semantics that is compatible with the most basic constructs of familiar programming languages, YAGI offers a small subset of the full functionality offered by situation calculus basic action theories and Golog programs. In particular, this version focuses on complete information in the initial situation and actions with local-effects, among other restrictions imposed by the syntax of YAGI. We should note though that to the best of our knowledge, it is the first time that a non logic-based language for specifying both a Golog domain and a Golog program is defined in a formal way.

One important difference between YAGI and the Golog extensions that handle on-line execution is that the semantics of YAGI are specified with respect to progression. YAGI functions as an on-line interpreter, parsing and handling each line of code separately, and progressing the internal representation after each step without keeping track of the history. As each line of code may be a code listing that includes the usual nondeterministic features of Golog, the standard regression semantics are used in order to identify an appropriate execution before executing and progressing. Our intention is to arrive at equivalent semantics as the standard semantics that rely purely on regression, but offer an alternative view that could be better suited for specifying the behavior of implementations that rely on updating the internal representation after action execution.

Of course, Golog is not the only available action-based programming language. We briefly want to mention two other examples: 3APL and FLUX. The agent model used in 3APL (Hindriks et al. 1999) is strongly aligned along BDI agent architectures (Bratman 1987). The basic ingredients of 3APL are basic actions, achievement goals, and test goals which are classified as basic goals in the 3APL terminology. Complex goals are composed from basic goals with sequential composition or are connected via nondeterministic choices including loops and conditionals. They found their approach on logic programming, and the reasoning paradigm is that of practical or means-end reasoning (e.g. (Georgeff and Lansky 1986)).

Another related approach using action-based programming is FLUX (Thielscher 2005). FLUX, which stands for FLUent eXecutor, introduces a kind of run-time system for the fluent calculus. Constrained logic programs encode agent tasks based on the so-called FLUX kernel which implements the state update axioms. The implementation of the FLUX system relies on constrained logic programming, with the user language being tightly coupled with the functionality of the underlying Prolog environment. Note that similar to the case of Golog implementations, this makes it difficult for non-expert users to use.

## Conclusions

In this paper we sketched the syntax and semantics of YAGI, a novel approach to realize an action-based programming language that is mostly inspired and based on Golog. With YAGI we want to overcome several shortcomings that come with the available run-time systems for Golog, namely the fuzzy border between implementation and target language, problems with the embeddability of existing Golog interpreters, and difficulties in usage by people without knowledge in logic or Prolog. While there exists no implementation for YAGI yet, the syntax and semantics show that we can express a restricted class of situation calculus basic action theories and Golog-like programs in a uniform language with familiar programming constructs. Our next goal is to implement a robust system based on this specification.

## Acknowledgments

We thank the anonymous referees for their helpful comments.

## References


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.

Bratman, M. 1987. *Intentions, Plans, and Practical Reason*. Harvard University Press.

Claßen, J., and Lakemeyer, G. 2006. Foundations for knowledge-based programs using es. In Doherty, P.; Mylopoulos, J.; and Welty, C. A., eds., *Proceedings, Tenth International Conference on Principles of Knowledge Representation and Reasoning (KR2006)*, 318–318.

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

Enderton, H. B. 1972. *A mathematical introduction to logic*. Academic Press.

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.

Ferrein, A., and Steinbauer, G. 2010. On the way to high-level programming for resource-limited embedded systems with golog. In Ando, N.; Balakirsky, S.; Hemker, T.; Reggiani, M.; and von Stryk, O., eds., *Second International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2010)*, 229–240.

Ferrein, A. 2010. golog.lua: Towards a non-prolog implementation of golog for embedded systems. In Hoffmann, G., ed., *Proceedings of the AAAI Spring Symposium on Embedded Reasoning*, (SS-10-04). AAAI Press.

Georgeff, M., and Lansky, A. 1986. Procedural knowledge. *Proceedings of the IEEE, Special Issue on Knowledge Representation* 74(10):1383–1398.

Giacomo, G. D.; Lesprance, Y.; Levesque, H. J.; and Sardina, S. 2009. *Multi-Agent Programming: Languages, Tools and Applications*. Springer. chapter IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents, 31–72.

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

Hindriks, K.; de Boer, F.; van der Hoek, W.; and Meyer, J.-J. 1999. Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems* 4(2):357–401.

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.

Lakemeyer, G., and Levesque, H. J. 2004. Situations, si! situation terms, no! In Dubois, D.; Welty, C. A.; and Williams, M.-A., eds., *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR2004)*, 516–526.

Levesque, H., and Lakemeyer, G. 2008. Chapter 23 cognitive robotics. In Frank van Harmelen, V. L., and Porter, B., eds., *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*. Elsevier. 869 – 886.

Levesque, H. J., and Pagnucco, M. 2000. Legolog: Inexpensive experiments in cognitive robotics. In *Proceedings of the Second International Cognitive Robotics Workshop*.

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming* 31(1-3):59–84.

Liu, Y., and Levesque, H. J. 2005. Tractable reasoning in first-order knowledge bases with disjunctive information. In Veloso, M. M., and Kambhampati, S., eds., *Proceedings of the Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference*, 639–644.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence* 4:463–502.

McIlraith, S. A., and Son, T. C. 2002. Adapting golog for composition of semantic web services. In Fensel, D.; Giunchiglia, F.; McGuinness, D. L.; and Williams, M.-A., eds., *Proceedings of the Eights International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, 482–496. Morgan Kaufmann.

Pham, H. 2006. Applying DTGolog to large-scale domains. Master's thesis, Department of Electrical and Computer Engineering, Ryerson University, Toronot, Canada.

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

Thielscher, M. 2005. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming* 5(4-5):533–565.

Vassos, S., and Sardina, S. 2011. A database-type approach for progressing action theories with bounded effects. In Lakemeyer, G., and McIlraith, S., eds., *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque*. College Publications.

Vassos, S.; Lakemeyer, G.; and Levesque, H. J. 2008. First-order strong progression for local-effect basic action theories. 662–272.