Languages for goals and plans

Overview of the existing plan languages

Dmitry Shaparau

shaparau@itc.it

ITC-IRST
Motivation

One of the key components of the planning framework are

- a language to express planning goals.
- a language to specify resulting detailed plans.

We are going to combine goals with plans:

- Goal: "REACH g1 THEN RUN plan $\pi$ THEN REACH g2"
- Plan:

  ...
  do action1;
  if(battery == low) {
    goal (position == recharging room);
  }
  ...

Languages for goals and plans — p. 2/2
Introduction

Approaches to high-level planning task programming can roughly be classified into those making use of some form of deliberation and those which do not:

- Reactive architectures
  - Task Definition Language
  - Reactive Plan Language
- Logic-based architectures
  - Golog
  - Golog for WS composition
  - ReadyLog
Computer games industry is a huge application area for AI techniques. To make a virtual environment more realistic the game has to simulate a behavior of game-world objects.

- SimHuman is tool for creation animated three-dimensional scenes with an arbitrary number of objects, virtual agents and user-controlled avatars.
- TDL is a high-level procedural language for the definition of the virtual environment objects behavior.
The key concept in TDL is a **task**

A task defines the combination of virtual agent **actions** (It has a duration and is executed in continuous timeframes)

- **primitive actions**, which are a set of commands that an agent can perform in one timeframe.

- **predefined actions**, which are predefined sequence of actions: $a_1, a_2, \ldots, a_n$, where $a_i$ is the set of primitive actions performed in the timeframe $i$.

- **goal-oriented actions**: $a_i = f(a_{i-1}, I, G)$, where $I$ is the set information about the objects, $G$ is the goal, and $f$ is a function that returns the next primitive action set based on the agent’s current action and senses.
The syntax of task definition is:

```
TASK task_name (type_1, arg_1, ..., type_n, arg_n)
#Variables
<variable declaration>
#Body
<block of commands>
#end
```
TDL (commands)

Possible task commands are:

- `<action>`
- task or function call
- `PAR(<block b1>, <block b2>)`: Blocks b1 and b2 are executed in parallel
- `DO(<block b>) UNTIL c`: Block b is executed until condition c is true
- `IF <bool c> THEN (<block b1>) ELSE (<block b2>)`: If condition c is true, block b1 is executed, else block b2 is executed
**TDL (example)**

**TASK** sit_on_free_table()

**Variables**

entity Table "", Chair ""

**Body**

DO ( task walk_around() )

UNTIL exists( Table, and( eq( [Table] class, 'table' ), and( exists( Chair, and( eq( [Chair] class, 'chair' ), and( near( Chair, Table ), ) ) ) ) ) ) ;

task go_and_sit(Table)

**end**
TDL (conclusion)

Advantages:
- It is really simple.

Drawbacks:
- It is strictly oriented to SimHuman tool. There is no possibility to introduce new primitive actions.
- Goal-oriented actions were introduced, but implementation and usage were not discussed.
- There is no information about the TDL state in last works of its authors, so probably TDL became forgotten.
Reactive Plan Language

Reactive Plan Language (RPL) was developed for high-level robot planning.

A RPL plan looks like a Lisp program.

The key concept of the RPL is a procedure that can be defined using

\[
(\text{DEF-INTERP-PROC name (formals) body})
\]

The execution is based on the task network construction.

\[
(:\text{TAG} \text{ LL} (\text{LOOP} (\text{A}) (:\text{TAG} \text{ SS} (\text{B}))))
\]

In order to access the second step B in the third iteration of the loop we can use a construction

\[
(\text{TAGGED SS (SUB ITER 3 LL)})
\]
During its life-cycle a task passes through following states:

1. *created*, when a task is first created
2. *enabled*, when a task is ready to be executed
3. *active*, when execution begins
4. *finished*, *failed* or *evaporated*.

The status of each task is stored in a fluent, which can be used to trigger events.
RPL (Key concepts)

RPL provides possibility to deal with following aspects:

- **failure**, a task failure can be managed using FAIL, TRY-ALL and TRY-IN-ORDER constructions. It is similar to the exception handling in high-level programming languages.

- **concurrency**, RPL controls the concurrency using a type of semaphore called a *valve*. Whenever two tasks should no execute simultaneously, we arrange that they compete for a valve. The entity that requests and waits for a valve is called a *process* (PROCESS name -body-). Valves are created using (CREATE-VALVE name b). It can be requested and released using (VALVE-REQUEST p v) and (VALVE-RELEASE p v).
RPL (Key concepts cont.)

- **policies and priorities.** Sometimes, a plan is executed to constrain the execution of another plan that is called a policy. For this reason RPL has a construction (WITH-POLICY s a), where ’s’ is a "secondary" plan and ’a’ is a "primary" plan. There is another construction (WITH-TASK-BLOCKED T -body-) which prevent running any subtask of the task T until the body is finished.

- **protections and violations.** Protected state is a state that has to be kept true during some interval. There is a PROTECTION construct in RPL for protected states management. (PROTECTION [:RIGID|:HARD|:SOFT] prop f repair), where ’f’ is a fluent being protected.
RPL (conclusion)

- RPL is a very flexible and powerful plan language.
- RPL is very similar to the programming languages such as Java or C.
- LISP syntax makes it harder to read and understand.
- There is a big set of reactive languages which are similar to RPL in spirit and notation.
Golog

Golog is a logic programming language.

Golog was developed for applications, whose design is based on a sophisticated logic of actions.

- High-level control of robots and mechanical devices.
- Programming intelligent software agents.
- Modeling and simulation of discrete event systems.

Golog uses the situation calculus for modeling dynamically changing application domains.
Situation calculus is a first order language with following key concepts:

- Situation = possible world history, which is a sequence of actions
- $S_0$ is used to denote the initial situation in which no actions have yet occurred.
- The world is conceived as being in a state $s$; this state can change only when some agent performs some action.
Golog (situation calculus)

- Binary function symbol $do(\alpha, s)$ denotes the successor situation to $s$ resulting from performing action $\alpha$.

- *Relational fluent* is a predicate which takes a situation term as the last argument. For example, $is\_carring(robot, p, s)$ means that a robot is carrying package $p$ in situation $s$.

- Knowledge fluent $K$, where $K(s, s')$ holds if system is in situation $s$ but believes it might be in situation $s'$. 
Golog (domain definition)

To define application domain we have to define following sets of axioms:

- Action precondition axioms.
- Successor state axioms.
- Unique names axioms for the primitive actions.
- Accessibility axioms for K.
- Domain initial state $S_0$ (including $K_0$).
- Complex actions and procedures.
Golog (example)

\textbf{if} car\textunderscore in\textunderscore driveway
\textbf{then} drive
\textbf{else} walk
\textbf{endIf}

\textbf{while} (\exists \text{block}) \text{ontable(block)} \textbf{do}
\text{remove\_block}
\textbf{endWhile}

\textbf{proc} remove\_block (\pi x)[ \text{pickup}(x); \text{putaway}(x)]
\textbf{endProc}

Where $\pi$ is a nondeterministic operator.
Golog (commands)

The full list of Golog constructs is following:

- **a** - primitive action
- **δ₁; δ₂** - sequence
- **φ?** - test
- **δ₁|δ₂** - nondeterministic choice of actions
- **(πx)δ(x)** - nondeterministic choice of arguments
- **δ*** - nondeterministic iteration
- **if φ then δ₁ else δ₂ endif** – conditionals
- **while φ do δ endwhile** – loops
Golog (implementation)

Golog interprets complex actions and procedures as macros which have to be expand into a situation calculus formula (sometimes second order), so

\[
\text{while } \phi \ \text{do } \delta \ \text{endWhile} \equiv [\phi?; \delta]^*; \neg \phi?\]
proc maketower(n)
  (πx,m)[ tower(x,m) ? ;
    if m≤n then stack(x,n−m)
    else unstack(x,m−n)
  endIf]
endProc
proc stack(x,n)
  n=0? | (πy)[ put(y,x); stack(y,n−1)]
endProc
proc unstack(x,n) ... endProc
maketower(7); ¬(∃x) on(x,A)?
Golog (execution)

In order to execute such kind of program Golog interpreter involves a finding a proof in **second-order logic**.
Golog is ancestor of the huge family of logic programming languages for dynamically changing application domain. Extensions exist for dealing with

- continuous change
- concurrency
- decision-theoretic planning
- others...

Golog interpreter works in two steps:
1. Transform Golog program into situation calculus formulas
2. Find a proof in second-order logic.
WS composition is considered as a planning and execution task. The new extension of Golog was achieved by introducing new language constructions:

- **Desirable** \((a, s)\) is a new fluent in situation calculus which means that action \(a\) is desirable in situation \(s\). It is different from precondition of action \(a\) \(Poss(a, s)\) where action \(a\) is physically possible in situation \(s\). This fluent constraints the search space for actions.

- **Legal** \((a, s) ≡ Poss(a, s) ∧ Desiable(a, s)\)
Golog for WS composition

WS composition is considered as a planning and execution task. The new extension of Golog was achieved by introducing new language constructions:

- **Order connective ":"**. Informally, $a_1 : a_2$ will perform the sequence of action $a_1; a_2$ whenever $\text{Poss}(a_2, \text{do}(a_1, s))$ is true. However, when it is false, the ":" construct dictates that Golog search for a sequence of actions $\bar{a}$ that achieves $\text{Poss}(a_2, \text{do}(\bar{a}, \text{do}(a_1, s)))$.

- All actions is distinguished in two groups: usual actions and sensing actions. Only sensing actions can be used to update knowledge data $K$. 
ReadyLog

Readylog is an extension of Golog which offers following additional control constructs:

- `solve(p; h)` - (offline) dt-planning
- `waitFor(c)` - event-interrupt
- `withCtrl(c; a1)` - condition-bounded execution
- `pconc(a1; a2)` - concurrent actions
- `prob(valprob; a1; a2)` - probabilistic actions
- `ppproj (c; a1)` - probabilistic (offline) projection
To illustrate the use of these constructs we give some examples from the soccer domain. To control different phases of the game special signals like beforeKickOff are sent from an external computer by the referee. The $with\text{Ctrl}$ statement means that as long as the condition holds the particular procedure is executed:

```
proc mainloop
    while gameRunning do
        pconc(withCtrl(beforeKickOff; positionOnField);
            withCtrl(gameOn; playSoccer),... 
    endwhile
endProc
```