Situation Calculus and YAGI

Gerald Steinbauer
Institute for Software Technology
Progression

- another solution to the **projection** problem
  - does a sentence **hold** for a **future** situation
- used for automated **reasoning** and **planning** in the SC
- observation:
  - regression becomes **slow** for a **long** action sequence $\alpha$
- idea:
  - progress (roll forward) the actual initial data $D_{S_0}$ to a **new** initial database $D_{S_\alpha}$
- advantages
  - queries can be answered using a **shorter** action sequence
- drawbacks
  - one loses the action history
  - progression is **not** always possible
Progression - Definition

- given a basic action theory
  \[ D = \Sigma \cup D_{SS} \cup D_{AP} \cup D_{UNA} \cup D_{S_0} \]
  and a ground action term \( \alpha \)
- a set of sentences \( D_{S_\alpha} \) is a (correct) progression of \( D_{S_0} \) to \( D_{S_\alpha} \) wrt. \( \alpha \) iff
  1. the sentences in \( D_{S_\alpha} \) are uniform in \( S_\alpha \)
  2. for every first-order formula \( \phi \) about the future of \( S_\alpha 
     \[ D \models \phi \text{ iff } (D - D_{S_0}) \cup D_{S_\alpha} \models \phi \]
Progression - Existence

- progression is the **dual** problem of regression
- unfortunately it is conceptually much more **complex**
- for a **finite** $D_{S_0}$
  - the progression is always definable using **second-order logic**
  - the progression is **not** always definable using **first-order logic**
- but we can **restrict** the initial database $D_{S_0}$
  - in order to express the regression in **first-order logic**
Relatively Complete Databases

• $D_{S_0}$ is relatively complete iff it is the set formed by
  1. the situation-independent sentences
  2. one sentence for each fluent $F$ in the form $\forall \bar{x}. F(\bar{x}, S_0) \equiv \Pi_F(\bar{x})$
     where $\Pi_F(\bar{x})$ is a situation-independent formula whose free variables are among $\bar{x}$
• consider a basic action theory \( D \) with a relatively complete initial database \( D_{S_0} \)
• suppose \( \alpha \) to be a ground action
• if we want to progress \( D_{S_0} \) to \( D_{S_{\alpha}} \) we have
  \[
  F(\tilde{x}, S_{\alpha}) \equiv \Phi_F(\tilde{x}, \alpha, S_0) \]
  … successor state axiom
• if \( F(\tilde{x}, S_{\alpha}) \) mentions a fluent atom \( G(\tilde{t}, S_0) \) replace it by
  • \( \Pi_G(\tilde{t}) \) using \( \forall \tilde{y}. G(\tilde{y}, S_0) \equiv \Pi_G(\tilde{y}) \)
• this leads to \( F(\tilde{x}, S_{\alpha}) \equiv \Psi_F(\tilde{x}, \alpha) \) where \( \Psi_F(\tilde{x}, \alpha) \) is situation-independent
• \( D_{S_{\alpha}} \) is the union of \( D_{U\text{NA}} \), the situation-independent sentences in \( D_{S_0} \) and the above for each fluent
Progressing Relatively Complete Databases II

• the set of sentences obtained are a progression of $D_{S_0}$ wrt. $\alpha$
• the progressed set is again relatively complete
  • we can apply the procedure again for another action
Golog

• Situation Calculus is yet only a theoretical construct
• Golog (alGol for Logic) is based on the Situation Calculus
• it is a program language for dynamic systems
• it allows a balance between reasoning/planning and imperative programming (i.e., planning is expensive)
• it allows complex actions
• there exist Prolog interpreters for Golog
Golog

- a Golog program $\delta$ is based SC
- it uses the macro $Do(\delta, s, s')$
- $Do(\delta, s, s')$ will be macro-expand to a SC formula
- the formula $Do$ states that situation $s'$ is reachable from situation $s$ by executing the program $\delta$
- the syntax supports primitive and complex actions
Golog Syntax (1)

• **Primitive Action**: $a$
  - has a precondition $Poss$ and the effects are modeled in the successor state axioms
  - $walk(R,L)$

• **Test action**: $\varphi?$
  - tests if $\varphi$ holds in a situation, does not change the situation
  - $\neg(\exists x,y).nextTo(x,y)\ ?^1$

• **Sequence**: $\delta_1;\delta_2$
  - executes $\delta_1$ and $\delta_2$ one after each other
  - $walk(R,L);pickup(R,K)$

^1 situation suppressed form of fluents
Golog Syntax (2)

- Non-deterministic Choice of Actions: $\delta_1|\delta_2$
  - (randomly) action $\delta_1$ or $\delta_2$ will be executed
  - $\text{walk}(R,A)|\text{walk}(R,B)$

- Non-deterministic Choice of Arguments: $(\pi x)\delta(x)$
  - (randomly) choose an argument $x$ for the action $\delta$
  - $(\pi x)\text{pickup}(R,x)$

- Non-deterministic Iteration: $\delta^*$
  - executes $\delta$ for a not defined number of times ($n \geq 0$)
  - $(\text{pickup}(R,L);\text{drop}(R,L))^*$
On the Semantics of Golog

- Golog programs are **macro-expanded** to Situation Calculus formulas using the macro $Do(\delta,s,s')$

- what is the **meaning** of Golog and a program $\delta$
  - $D \models (\exists s).Do(\delta,S_0,s)$
  - the Basic Action Theory **entails** if a given program $\delta$ lead to the situation $s$ starting from $S_0$

- drawback: the macros are **less expressive**
- a **program trace** can be obtained by a constructive proof of the above sentence
- some properties are provable: e.g., **termination**
YAGI

• Yet Another Golog Interpreter
• Golog suffers from several drawback
  • unclear and uncommon syntax and semantics
  • Prolog-based interpreter – no separation of front/backend
  • offline execution
  • no feedback from the world
• YAGI is designed for
  • keeping the situation calculus/Golog foundations
  • a sound definition of the language
  • a separation of the front/backend
  • familiarity to other common program languages
  • online execution
  • providing language statements to build up a basic action theory
YAGI – Interpreter Architecture

Diagram showing the architecture of YAGI, including:
- Parser
- YAGI Line L
- YAGI Shell
- AST (L)
- Query Results
- Status Information
- Diagnostics

Back-End:
- YAGIBAT
- Program
- Signal (Action Exec.)
- Signal (Setting Action)
- Signal (Setting Action Response)
- Signal (Exogenous Event Data)

System Interface:
- Navigation
- Localization
- Sensors
- ...
Fluents as Databases

• YAGI fluents are similar to fluents in Situation Calculus
  • they change their truth value with the situation
  • but YAGI always does progression and refer to $D_{S_0}$, YAGI omits the situation term

• a n-ary fluent $F$ in the BAT in YAGI is defined as
  • fluent $F[\text{domain}_1,\text{domain}_2,\ldots,\text{domain}_n]$;
  • domain is a finite set of strings, e.g. {"r1","r2","r3"}

• the semantics of a fluent $F$ is defined
  • by a database table
  • the database columns represent the fluent’s parameters
  • a fluent $F[<x_1,x_2,\ldots,x_n>]$ holds iff the tuple is represented as a row in the database
YAGI – Domain Definition

- **Fluents**
  - defined by a fluent definition
  - can be immutable if the keyword `fact` is used

- **Actions**
  - defined by an action definition of the following form
    
    ```
    action goto($x,$y)
    precondition: ((<$x> in at) and not(<$y>));
    effect: at = {<$y>};
    signal: "move from " + $x + " to " + $y;
    end action
    ```
YAGI Procedures

• Procedures
  • defined by a procedure definition of the following form
    proc dummy($x,$y)
      action_a($x);
      action_b($y);
    end proc
  • the body of a procedure may contain any number of YAGI actions and YAGI statements
YAGI Program

• a YAGI program comprises of
  • a set of declarations for
    • fluents
    • facts
    • fluent assignments
    • actions
    • procedures
  • a sequence of YAGI statements
    • test action
    • procedure call
    • nondeterministic choice of actions
    • nondeterministic pick of arguments
    • conditional
    • for loop
    • while loop
    • search
YAGI – Fluent Assignments I

- can be used to assign truth values to fluents
- it can be done for a n-ary fluent
  - \( F = S \); or \( F = G \); where \( S \) is a set of tuples compatible with \( F \)'s arity and domains and \( G \) is a fluent with the same arity and domains, fluent \( F \) only holds for the members of \( S \) or for the same terms \( G \) holds
  - \( F += S \); or \( F += G \); where \( S \) is a set of tuples compatible with \( F \)'s arity and domains and \( G \) is a fluent with the same arity and domains, fluent \( F \) additionally holds for the members of \( S \) or for the same terms \( G \) holds
  - \( F -= S \); or \( F -= G \); where \( S \) is a set of tuples compatible with \( F \)'s arity and domains and \( G \) is a fluent with the same arity and domains, fluent \( F \) does not hold for the members of \( S \) or for the same terms \( G \) holds
- assignments can be used in the declaration to specify the initial database \( D_{S_0} \)
YAGI – Fluent Assignments II

• **YAGI** allows also **loop assignments** of the form
  
  ```plaintext
  foreach <$x> in objects do
    owner += {<$x,"Alice”>}
  end for
  ```

• **YAGI** allows also **conditional assignments** of the form
  
  ```plaintext
  if ("Bread","Alice”> in owner) do
    happy += {"Alice”} 
  else
    happy -= {"Alice”}
  end if
  ```
YAGI Formulas I

- are similar to first order logic formulas
- they are used in
  - action preconditions, test actions, conditionals, while loops
- they are online evaluated for their truth value
YAGI Formulas II

- formulas are **build up** the following way
  - formula := atom
    - not (formula)
    - atom connection atom
    - exists tuple\(^1\) in set (such formula)
    - all tuple in set (such formula)
    - tuple in set
  - atom := value compare value
    - set compare set
    - true
    - false
  - compare := ==, !=, |<=, >=, <, >
  - connection := and, or, implies

\(^1\) tuples may comprise variables \(<x, "r1">\)
YAGI Execution Semantics

- YAGI follows an **online** semantic (similar to IndiGolog)
- the next possible and necessary action is derived and **immediately** executed
- the execution semantics is defined by 2 predicates
  - \( YagiTrans(\alpha, b, \alpha', b') \) holds if the YAGI program \( \alpha \) can be **executed** using database \( b \) leading to the remaining program \( \alpha' \) and an updated database \( b' \)
  - \( YagiFinal(\alpha, b) \) holds if the YAGI program \( \alpha \) can legally **terminate** using database \( b \)
Semantics of YAGI Parts (1)

• **primitive Action:** $a$
  - $YagiTrans(a, b, \alpha', b') \equiv \phi_{AP,a} \land \alpha' = null \land b' = \text{exec}(a, b)$
  - $YagiFinal(a, b) \equiv false$
  - $\phi_{AP,a}$ is the **precondition** of $a$, $null$ is the **empty** program, $\text{exec}$ updates the database according to $a$’s effects

• **test action:** $\phi;$
  - $YagiTrans(\phi; , b, \alpha', b') \equiv \phi[b] \land \alpha' = null \land b' = b$
  - $YagiFinal(\phi; , b) \equiv false$

• **Sequence:** $\alpha_1;\alpha_2$
  - $YagiTrans(\alpha_1; \alpha_2, b, \alpha', b') \equiv$
    \[
    \exists \gamma. \alpha' = (\gamma; \alpha_2) \land YagiTrans(\alpha_1, b, \gamma, b') \lor
    YagiFinal(\alpha_1, b) \land YagiTrans(\alpha_2, b, \alpha', b')
    \]
  - $YagiFinal(\alpha_1; \alpha_2, b) \equiv YagiFinal(\alpha_1, b) \land YagiFinal(\alpha_2, b)$
YAGI Setting Actions

- setting actions are similar to primitive actions
- except they have additional variables after a keyword external
- these variables are set by the backend
- the enable feedback from the world
- defined by an action definition of the following form

```plaintext
action holding ($o) external ($x)
  effect:    if ($x=="true")
             holds += {<$o>};
             else
             holds -= {<$o>};
  signal:   "holding " + $o;
end action
```
Signals

• signals are strings
• strings and the content of variables can be concatenated
• signals are emitted by actions to trigger activities in the backend
• the signal is decoded by the backend
Questions ?