CSE 591 - FALL 04. HANDOUT 6. VERSION 0.2

Chitta Baral

Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406 USA
chitta@asu.edu

http://www.public.asu.edu/∼cbaral/

November 30, 2004

COMPLEX PLANS AND AGENT ARCHITECTURES

Recap

- We learned about how to **describe actions and their impact** in the environment; expressing observations; expressing relationship between objects in the world.
- We learnt about planning from the initial state;
- We will learn about observation assimilation; planning from the current state; and *a deliberative agent architecture.*
- We learnt about **expressing goals and directives** using temporal logic constructs (LTC, and CTL*) and also the notion of maintainability.
- Next Step: Agent architectures and complex plans.

Agent architectures

- Purely reactive: Sense-match-act
- Reactive with a domain-dependent reasoning component: Instead of match, domain-dependent information is used to decide what action to do next.
- Purely deliberative:
  - Observe (Sense) and assimilate.
  - Make a plan. (or revise the current plan)
  - Execute part of the plan.
- Hybrid mixture of reactive and deliberative:
  - Reactivity for most important, most critical, most frequently encounterd, most recent states and deliberativity for rest.
  - The reactive states are frequently updated, especially the "most recent" part; based on recent deliberations.
  - Various level of reactivity: some with match and others with various degree of reasoning using domain-dependent knowledge.

Various kinds of policies and various kinds of sensing

- $states \rightarrow actions$ $\qquad states \rightarrow 2^{actions}$
- $propositional\_formulas \rightarrow actions$ $\qquad propositional\_formulas \rightarrow 2^{actions}$
- $trajectories \rightarrow actions$ $\qquad trajectories \rightarrow 2^{actions}$
- $past\_temporal\_formulas \rightarrow actions$ $\qquad past\_temporal\_formulas \rightarrow 2^{actions}$
- $CTL^*\_with\_PTL\_formulas \rightarrow actions$ $\qquad CTL^*\_with\_PTL\_formulas \rightarrow 2^{actions}$
- $Uniform\_belief\_states \rightarrow actions$ $\qquad Uniform\_belief\_states \rightarrow 2^{actions}$
- $Prob\_belief\_states \rightarrow actions$ $\qquad Prob\_belief\_states \rightarrow 2^{actions}$
- Various kinds of sensing:
  - complete sensing every cycle
  - limited sensing every cycle
  - error-prone sensing every cycle

Approaches for "reactive with domain dependent reasoning component"

The match-act part of a purely reactive architecture is replaced by:

- Condition, Reasoning program pair:

  The amount of time the reasoning program takes is usually more than table lookup and less than knowledge-less planning (i.e., planning without any domain knowledge to help guide the planning process)

- Different kinds of reasoning programs

  – Prolog or logic programming based. (Kowalski, Sadri, Pereira, etc.)

  – Specific agent programming languages (See the book by Subrahmanian et al.)

  – Planning using domain dependent knowledge

    Possible kinds of domain dependent knowledge

    * Temporal
    * Hierarchical (HTN)
    * Procedural (GOLOG and ConGOLOG)
    * A combination of the above

The number of (Condition, Reasoning program) pairs may not be too big, if a reasoning program (or the domain dependent knowledge) is applicable for a large number of states.

– BDI agent programs (Belief, Desire, Intention)

Procedural knowledge: GOLOG and ConGOLOG [Reiter01]

- GOLOG: Al**GO**l in **LOG**ic.
- An Algol or Pascal like language with a logical foundation behind it.
- An example GOLOG program:

  $a_1; a_2; (a_3|a_4|a_5); f?$

  It encodes procedural knowledge about a plan that says: do action a_1, followed by a_2, followed by one of a_3, a_4 and a_5 such that f is true after that.
- An off-line GOLOG interpreter takes a GOLOG program, has information about effect of actions and the environment, reasons about the actions using that information, and generates a sequence of actions (i.e., a plan).

  It is expected to be faster than knowledge-less planners as it has to search less.

Syntax of GOLOG

1. an action a is a program,
2. a formula ϕ? is a program,
3. if p_i's are programs then $p_1; \ldots; p_n$ is a program,
4. if p_i's are programs then $p_1 \mid \ldots \mid p_n$ is a program,
5. if p_1 and p_2 are programs and ϕ is a formula then " **if** ϕ **then** p_1 **else** p_2" is a program (also denoted by $if(\phi, p_1, p_2)$),
6. if p is a program and ϕ is a formula then " **while** ϕ **do** p" is a program (also denoted by $while(\phi, p)$), and
7. if X is a variable of sort s, $p(X)$ is a program, and $f(X)$ is a formula, then $\mathbf{pick}(X, f(X), p(X))$ is a program.

Besides we may have procedures of the form: $proc(name(X), body(X))$.

Operational semantics of GOLOG

A trajectory $s_0 a_0 s_1 \ldots a_{n-1} s_n$, denoted by α, is *a trace of a GOLOG program* p if the following holds:

- for $p = a$ and a is an action, $n = 1$ and $a_0 = a$,
- for $p = \phi?$, $n = 0$ and ϕ holds in s_0,
- for $p = p_1; p_2$, there exists an i such that $s_0 a_0 \ldots s_i$ is a trace of p_1 and $s_i a_i \ldots s_n$ is a trace of p_2,
- for $p = p_1 \mid \ldots \mid p_n$, α is a trace of p_i for some $i \in \{1, \ldots, n\}$,
- for $p =$ **if** ϕ **then** p_1 **else** p_2, α is a trace of p_1 if ϕ holds in s_0 or α is a trace of p_2 if $\neg\phi$ holds in s_0,
- for $p =$ **while** ϕ **do** p_1, $n = 0$ and $\neg\phi$ holds in s_0 or ϕ holds in s_0 and there exists some i such that $s_0 a_0 \ldots s_i$ is a trace of p_1 and $s_i a_i \ldots s_n$ is a trace of p, and
- for $p =$ **pick**$(\vec{X}, f(\vec{X}), q(\vec{X}))$, then there exists a constant $\vec{x}$ of the sort of $\vec{X}$ such that $f(\vec{x})$ holds in s_0 and α is a trace of $q(\vec{x})$.

- if $p = name(v)$ for a procedure $proc(name(X), body(X))$ then $s_0 a_0 s_1 \ldots a_{n-1} s_n$ is a trace of $body(v)$.

Elevator example

- Actions:
  - $up(N)$: go up to the floor N.
  - $down(N)$: go down to the floor N.
  - $turnoff(N)$: turn off the request to go to N.
  - $open$: open the elevator door.
  - $close$: close the elevator door.
- Fluents:
  - $currentFloor(N)$: the elevator is at floor N.
  - $on(N)$: There is a request to go to floor N.
  - $opened$: the elevator door is opened.
  - $above(N)$ and $below(N)$ are two defined fluents; meaning that the floor N is above (or below resp.) the current floor.

- Effect of actions:

$up(N)$ **causes** $currentFloor(N)$
$up(N)$ **causes** $\neg currentFloor(N')$ **if** $currentFloor(N'), N \neq N'$
$down(N)$ **causes** $currentFloor(N)$
$down(N)$ **causes** $\neg currentFloor(N')$ **if** $currentFloor(N'), N \neq N'$
$turnoff(N)$ **causes** $\neg on(N)$
$open$ **causes** $opened$
$close$ **causes** $\neg opened$
executable $up(N)$ **if** $above(N)$
executable $down(N)$ **if** $below(N)$
executable $turnoff(N)$ **if** $on(N)$

- Initial state:

initially $on(3), on(5)$ $(\neg on(X), X \neq 3, \neq 5)$
initially $currentFloor(2)$ $(\neg currentFloor(X), X \neq 2)$
initially $\neg opened$

- Some GOLOG procedures:

  $proc(serve(N), [go(N), turnoff(N), open, close]).$
  $proc(go(N), up(N)|down(N)|currentFloor(N)).$
  $proc(serve\_a\_floor, pick(N, on(N), serve(N))).$
  $proc(control, [while(on(N), serve\_a\_floor),$
  $\quad if(currentFloor(0), open, [down(0), open]).$

- Off-line traces: The program $serve\_a\_floor$ has two traces:
  - *up(5), turnoff(5), open, close*
  - *up(3), turnoff(3), open, close.*

- Exercise: What is the goal achieved by the GOLOG call $control$? Express it in Temporal logic.

Partial ordered knowledge: hierarchical task networks (HTNs)

- Motivation:
  - Certain domain knowledge is better expressed as partial ordered information.
  - Example: Classes need to be taken to graduate; a partial order of pre-requisites.
    - ACM and similar bodies give a description (goal) of what is needed for someone to have a Bachelor's degree.
    - Several courses are identified. Taking a course is an *action*.
    - The *effect* of taking a course results is that the student acquires certain knowledge.
    - Certain courses can only be taken if a student already has certain knowledge. (*executability condition*).
    - An university or department usually does not let students use the above knowledge to make a plan of which courses to take when.
    - It rather gives a flow chart (or a partial order) about which courses can be taken when and how many courses (of what kind) a student needs to take to get a certain degree. Such information can be expressed as an HTN.

  * The HTN serves as domain knowledge that the student can use (in this case is required to use) to make a plan.
  * The advantages of using HTN are: easier to plan and easier to check a plan.
  * While checking the plan, we don't need to use the goal but rather check if the HTN is followed or not.
  * But we should a-priori check that a plan obtained using the HTN indeed achieves the goal.
  * The above is rarely done. It is a good exercise (with some research issues) to take an HTN from the literature, and figure out what goal the HTN is meant to achieve and prove that it indeed achieves that.

- HTN stands for Hierarchical Task Networks, is originally developed more than 25 years ago(Sacerdotti 1977; Tate 1977).
- It is said that most planners that are use in the industry are based on HTN. (But they ignore expressing the goal explicitly and verifying the correctness of their HTN.)
- As in case of GOLOG, HTN helps in speeding up plan construction as it reduces search.

HTN syntax and relationship with GOLOG

- Two kind of tasks:
  - primitive tasks: tasks that can be done directly. (e.g., a simple action.)
  - non-primitive tasks: a task defined using a method; similar to procedures in GOLOG; how to perform them need to be figured out.
- Primitive tasks: Have an *operator* of the form $[operator\ f(v_1 \ldots v_k)(pre : l_1, \ldots, l_m)(post : l'_1, \ldots, l'_n)]$, which describes the executable condition $(pre : l_1, \ldots, l_m)$ and effect $(post : l'_1, \ldots, l'_n)$ of the primitive task $f(v_1 \ldots v_k)$.
- Non-primitive tasks: have a method, similar to a GOLOG procedure.
  - a method is of the form (α, d), where α is a non-primitive task name, and d is a task network.
  - α corresponds to a procedure name in GOLOG, and d corresponds procedure body.

- Tasks and task networks: correspond to GOLOG programs.

  Is of the form $[(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \varphi]$, where

  n_i is a label for task α_i

  φ is a constraint formula constructed from variable binding constraint such as $(v = v')$ and $(v = c)$, ordering constraint such as $(n < n')$, state constraint such as (n, l), (l, n) and (n, l, n'), where v, v' are variables, c is a constant, n, n' are tasks, and l is a literal.

- A *plan* is a sequence of ground primitive tasks.

- Note: No mention of a goal of a task network beyond achieve tasks. (non-primitive tasks are sometimes categorized as achieve tasks and perform tasks.)

HTN Examples

1. Moving blocks.

$$[(n_1 : achieve[clear(v_1)])(n_2 : achieve[clear(v_2)])(n_3 : do[move(v_1, v_3, v_2)])$$
$$(n_1 < n_3) \wedge (n_2 < n_3) \wedge (n_1, clear(v_1), n_3) \wedge (n_2, clear(v_2), n_3) \wedge (on(v_1, v_3), n_3)$$
$$\neg(v_1 = v_2) \wedge \neg(v_1 = v_3) \wedge \neg(v_2 = v_3)]$$

There are 3 tasks: clearing v_1, clearing v_2 and moving v_1 to v_2. The constraints are that moving v_1 must be done last, that v_1 and v_2 must remain clear until we move v_1, that v_1, v_2, v_3 are different blocks, and that $on(v_1, v_3)$ must be true immediately before the move.

2. Traveling

- There can be more than one methods for the same task.
- $(travel(x, y), [(n_1 : pay\_driver)(n_2 : get\_taxi)(n_3 : ride\_taxi(x, y)),$
  $n_2 < n_1 \wedge n_3 < n_2])$.
  The above corresponds to traveling by taxi.
- $(travel(x, y), [(n_1 : buy\_ticket(airport(x), airport(y))$
  $(n_2 : drive(x, airport(x))(n_3 : fly(airport(x), airport(y)))$
  $(n_4 : drive(airport(y), y)), n_1 < n_2 \wedge n_2 < n_3 \wedge n_3 < n_4])$
  The above corresponds to traveling by air.

Exercise:

- Do the exercise in Page 13.
- Complete the blocks world example in Page 18. I.e., give an HTN that can achieve a goal configuration of blocks from an initial state. (Hint: Read papers on SHOP by Nau et al. from the University of Maryland.)

Operational semantics of HTN planning

- Let p is a ground primitive task, and s be a state, $apply(s, p)$ is defined by $\Phi(p, s)$($\mathcal{A}$ semantics).
- Let $d = [(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \varphi]$ be a ground primitive task network. Let s_0 be the initial state. A plan σ completes d at s_0 if it is an ordering of $\alpha_1 \ldots \alpha_m$ such that the constraint φ is satisfied.

  More precisely, let $\pi(i)$ denote the position of α_i in the new ordering given by $\sigma = \alpha_{f(1)} \ldots \alpha_{f(m)}$, where $f(1) \ldots f(m)$ is a permutation of $1 \ldots m$. (I.e., $\pi(i) = j$ iff $f(j) = i$.) Also let $s_i = apply(s_{i-1}, \alpha_{f(i)})$. Then σ completes d if

  - $first\{n_i, n_j, \ldots, \ldots\}$ evaluates to $min\{\pi(i), \pi(j), \ldots\}$,
  - $last\{n_i, n_j, \ldots, \ldots\}$ evaluates to $max\{\pi(i), \pi(j), \ldots\}$,
  - $(n_1 < n_j)$ if $\pi(i) < \pi(j)$,
  - (l, n_i) is true if l holds in $s_{\pi(i)-1}$,
  - (n_i, l) is true if l holds in $s_{\pi(i)}$,
  - (n_i, l, n_j) is true if l holds for all s_t, $\pi(i) \leq t < \pi(j)$

- Let $d = [(n : \alpha)(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \varphi]$ be a task network, where α is a compound task. Let $m = (\alpha, [(n'_1 : \alpha'_1) \ldots (n'_k : \alpha'_k), \varphi'])$ be a method for α. Then d can be decomposed into a "more primitive" task
  $d' = [(n'_1 : \alpha'_1) \ldots (n'_k : \alpha'_k)(n_1 : \alpha_1) \ldots (n_m : \alpha_m), \varphi' \wedge \psi]$, where ψ is obtained from φ with the following modifications:
  - replace $(n < n_j)$ with $(last\{n'_1, \ldots, n'_k\} < n_j)$;
  - replace $(n_j < n)$ with $(n_j < first\{n'_1, \ldots, n'_k\})$;
  - replace (l, n) with $(l, first\{n'_1, \ldots, n'_k\})$;
  - replace (n, l) with $(last\{n'_1, \ldots, n'_k\}, l)$;
  - replace (n, l, n_j) with $(last\{n'_1, \ldots, n'_k\}, l, n_j)$;
  - replace (n_j, l, n) with $(n_j, l, first\{n'_1, \ldots, n'_k\})$;
  - replace n in a $first$ or $last$ expression by $n'_1, \ldots, n'_k$.
- A plan P solves a task d, if d can be decomposed into d_1, and d_1 can be decomposed into d_2 and so on until d_n, such that d_n is a primitive task network and P completes d_n.

Temporal domain knowledge

- Earlier we discussed temporal goals.
- Temporal logic can also be used to specify domain knowledge that guides us in finding a goal.
- Given a goal g we say a temporal formula F is sound with respect to g, and an initial state s, if for any trajectory $s_0 = s, a_1, s_1, \ldots, a_n, s_n$ that satisfy F, there is a minimal plan α with $a_1, \ldots, a_n$ as a prefix.
- Given a goal g we say a temporal formula F is complete with respect to g, and an initial state s, if for all minimal plans $a_1, \ldots, a_n$ that achieve s from f, the trajectories $s_0 = s, a_1, s_1, \ldots, a_n, s_n$ satisfy F.
- A plan is said to be minimal, if it is no longer a plan after we remove any action from it.

Logistics Domain

- Actions:
  - $load(O, V, L)$: vehicle V loads object O at location L.
  - $unload(O, V, L)$: vehicle V unloads object O at location L.
  - $move(V, L1, L2)$: vehicle V moves from location $L1$ to $L2$.
- Fluents:
  - $in(O, V)$: object O is in vehicle V.
  - $at(O, L)$: object O is at location L.
- Effects of actions:

  $load(O, V, L)$ **causes** $\{in(O, V), \neg at(O, L)\}$.

  $unload(O, V, L)$ **causes** $\{\neg in(O, V), at(O, L)\}$.

  $move(V, L1, L2)$ **causes** $\{\neg at(V, L1), at(V, L2)\}$.

  executable $load(O, V, L)$ **if** $\{at(O, L), at(V, L)\}$.

  executable $unload(O, V, L)$ **if** $\{in(O, V), at(V, L)\}$.

  executable $move(V, L1, L2)$ **if** $\{at(V, L1), \neg at(V, L2)\}$.

Temporal Control Knowledge for the logistics domain

1. In the next step, either do not move a vehicle, or move a vehicle only to a place where there is an object to be picked up.

   $\forall V : at(V, L) \rightarrow \bigcirc(at(V, L) \vee (at(V, L') \wedge L' \neq L \wedge \exists O : at(O, L) \wedge \neg goal(at(O, L)))$

2. Do not unload an object to a location that is not the goal place.

   $$\Box((in(O, V) \wedge \neg goal(O, L) \wedge \bigcirc(at(V, L))) \rightarrow \bigcirc(in(O, V)))$$

3. If an object O is being at L_1, which has L_2 as its goal place, then moving an vehicle to L_1 must precede moving to L_2.

   $\Box(at(O, L1) \wedge at(V, L) \wedge L \neq L_1 \wedge goal(at(O, L_2)) \wedge L_2 \neq L_1 \wedge L_2 \neq L \wedge \bigcirc(\neg at(V, L2)))$

   Note: it is problematic when there is another object at L_2 that wants to be moved to L_1

Blocks World

- Actions: $pickup(X)$, $putdown(X)$, $stack(X, Y)$, $unstack(X, Y)$.
- Fluents: $ontable(X), clear(X), handempty, holding(X), on(X, Y)$.
- Effect of actions:
  - $pickup(X)$ **causes** $\{holding(X), \neg ontable(X), \neg clear(X), \neg handempty\}$.
  - $putdown(X)$ **causes** $\{\neg holding(X), ontable(X), clear(X), handempty\}$.
  - $stack(X, Y)$ **causes** $\{on(X, Y), clear(X), handempty, \neg holding(X), \neg clear(Y)\}$
  - $unstack(X, Y)$ **causes** $\{\neg on(X, Y), \neg clear(X), \neg handempty, holding(X), clear(Y)\}$.
- Executability conditions:
  - **executable** $pickup(X)$ **if** $ontable(X), clear(X), handempty$.
  - **executable** $putdown(X)$ **if** $holding(X)$.
  - **executable** $stack(X, Y)$ **if** $holding(X), clear(Y)$.
  - **executable** $unstack(X, Y)$ **if** $on(X, Y), clear(X), handempty$.

Temporal domain constraints for the Blocks world

- $goodtower(X) \equiv clear(X) \wedge \neg Goal(holding(X)) \wedge goodtowerbelow(X)$
- $goodtowerbelow(X) \equiv (ontable(X) \wedge \neg\exists[Y : Goal(on(X,Y))])$
  $\vee\exists[Y : on(X,Y)]\neg Goal(ontable(X)) \wedge \neg Goal(holding(Y)) \wedge \neg Goal(clear(Y))$
  $\wedge\forall[Z : Goal(on(X,Z))]z = y \wedge \forall[Z : Goal(on(Z,Y))]Z = X$
  $\wedge goodtowerbelow(Y)$
- $badtower(X) \equiv clear(X) \wedge \neg goodtower(X)$
- $\Box(\forall[X : clear(X)]goodtower(X) \Rightarrow \bigcirc(clear(X) \vee \exists[Y : on(Y,X)]goodtower(Y))$
  $\wedge badtower(X) \Rightarrow \bigcirc(\neg\exists[Y : on(Y,X)])$
  $\wedge(ontable(X) \wedge \exists[Y : Goal(on(X,Y))]\neg goodtower(Y)) \Rightarrow \bigcirc(\neg holding(X)))$.

  Do not destroy good towers; do not place additional blocks on top of bad towers; and do not pick up singleton bad tower blocks unless their final position is ready.
- Open question: Is the above formula sound? complete? with respect to arbitrary initial states and goals.