A multi-threaded architecture for cognitive robotics

Keith Clark
Imperial College London
Overview of talk

- Nilsson’s Teleo-Reactive (TR) Procedures
- Simulation demo of TR controlled can collecting robot
- Nested TR procedure calls and multi-threading
- TR procedures accessing inferred beliefs
  - a cognitive robot control language
- Simulation demo of TR controlled block manipulating robot
  - command and goal events
  - multiple tasks
- Classic BDI event based multi-tasking agent architecture
  - partial plans expanded during execution
  - but plans lack robustness of TR procedures
- From TR procedures to TR plans
- AgentMT(TR)
  - a multi-threaded BDI architecture using TR plans
Programming reactive robots - Teleo-Reactive (TR) procedures

- Production rule notation for controlling reactive behaviour
- Inspired by control theory ideas of continuous monitoring and persistent responses
- But has parameterized procedures and nested procedure calls - so quite rich program structuring tools
- Due to Nils Nilsson -
Motivating example: Can collecting robot

- Enclosed flat space with empty red soft drinks cans scattered about and a corner painted blue

- There are no other objects in the space
- Robot is to locate, grab and deliver a red can to the blue corner
- It then waits until can is removed by a human
- Then repeats the task
- New red cans can be placed in space at any time
Robot’s percepts and actions

• Robot has a camera pointing forwards, obstacle detecting sensors, and a gripper with pressure sensors.

• It has a percept routines that are constantly analysing the camera image and the other sensor readings to provide updates of its perceptual beliefs:

  - see(D,C)  C coloured blob at position D in field of vision
  - D is left, center or right
  - near(C)  close to a C coloured thing
  - touch  touching something
  - at(blue)  at the blue corner
  - holding  something is between grippers

• It has action routines that enable it to:

  - forward(F)  move forward at speed F
  - turn(D,R)  turn to D (left or right) at rotation speed R
  - close  close grippers
  - open  open grippers
Teleo-reactive can collector

collect_cans .. { 
  at(blue), see(_,red), not holding ⇒ {}.
  holding ⇒ deliver_can.
  true ⇒ open, get_can.
}.

get_can .. { 
  touching ⇒ close.
  see(_,red), near(red) ⇒ approach(red,0.5,0.1).
  see(_,red) ⇒ approach(red,1.5,0.3).
  true ⇒ turn(right, 0.5).
}.

approach(C,F,R) .. { 
  see(center,C) ⇒ forward(F).
  see(Side,C) ⇒ forward(F) \| turn(Side,R).
  true ⇒ nil
}.

concurrent actions
General form of TR procedure
(generalising Nilsson’s syntax)

\[ p(\ldots) \ldots \{ \]
\[ K_1 \Rightarrow A_1 \sim U_1. \]
\[ K_2 \Rightarrow A_2 \sim U_2. \]
\[ \ldots \]
\[ K_m \Rightarrow A_m \sim U_m. \]
\} 

Each \( K_i \) is a boolean test on beliefs (perceptual level or inferred)
Each \( U_i \) is an optional belief store update
Each \( A_i \) is a (possibly) parallel composition of sequences of
primitive actions one of which may end with a call to
a TR procedure, even a recursive call to \( p \)
or the empty action \{\}
Informal rules of evaluation

• **First rule** with true condition is **fired**
  • its action is started and any memory update performed
• Actions are by default **durative** (eg `forward(F)`)
  • Actions of fired rule persists until another rule becomes first rule with a true condition and is fired
• **Rule conditions are continuously evaluated**
  • In practice re-evaluated only when a belief that might effect choice of rule to fire is changed
• When an action of fired rule calls a T-R procedure
  • Rules of *calling* procedure **still evaluated**
  • Called proc. **exited** as soon as fired rule changes in calling procedure
  • In a multi-threaded implementation each called TR procedure has its own **thread** of control
Exit from called procedure

\[
p(\ldots)\{ \\
K_1 \Rightarrow A_1 \\
K_2 \Rightarrow q(\ldots) \\
\ldots \\
K_m \Rightarrow A_m \\
\}
\]

q exited as soon as:
K_1 becomes true
or K_2 becomes false and
some K_i, i>2 becomes true
Desirable features of TR programs

**regression**
for each rule $K_i \Rightarrow A_i$, $i>1$
the execution of $A_i$ normally results in a state of the world in which the test $K_j$ of an earlier rule (i.e. $j<i$) will be true

**completeness**
In each procedure there is always one rule that can be fired
- can be guaranteed by having a last default rule with condition **true**

**universal**
A procedure that is complete and satisfies the regression property is a *universal* program for the goal $K_1$ - the test of its top-most rule
TR rules using inferred beliefs

• Agent belief store:
  – perceptual beliefs *computed* from sensor readings
  – fixed beliefs about the environment (environment model)
  – definitions of higher level concepts defined in terms of perceptual and fixed beliefs beliefs

• Inferable beliefs give us higher level view of the current state of the environment

• Cognitive level TR procs. mostly test inferred beliefs

• Reactive level TR procs. tests mostly test perceptual beliefs

• The cognitive procs. *cascade down* to the reactive procs.
Layered TR programming

\[ p(\ldots) \{ \]
\[ T_1 \Rightarrow A_1 \]
\[ T_2 \Rightarrow q(\ldots) \quad q(\ldots) \{ \]
\[ \ldots \]
\[ T_i \Rightarrow q'(\ldots) \quad I_1 \Rightarrow A'_1 \]
\[ \ldots \]
\[ T_i \Rightarrow q'(\ldots) \quad I_2 \Rightarrow r(\ldots) \quad r(\ldots) \{ \]

switches between diff. inter.
level procs. as high level
inferred beliefs change

\}

invokes a diff. low level
proc. as inter. level
inferred beliefs change

\}
Re-implementation of Nilsson’s Block Moving Robot Arm
Robot arm belief store

• We assume camera image can be analyzed to compute percepts:
  \[\text{on}(1,\text{tbl}) \text{ on}(2,3) \text{ holding}(5)\]

• Higher level beliefs can be inferred from these using fixed beliefs:
  \[\text{clear} (\text{tbl})\] - table always has a clear space somewhere
  \[\text{block}(1). \ \ldots \ \text{block}(5).\] number 1 to 5 identify blocks

and the rules:

\[\text{clear}(B) \leftarrow \text{block}(B), \neg \text{on}(\_,B).\]

\[\text{ordered}([B]) \leftarrow \text{on}(B,\text{tbl}).\]

\[\text{ordered}([B1, B2 | Bs]) \leftarrow \text{on}(B1, B2), \text{ordered}([B2 | Bs]).\]

\[\text{tower}([B | Bs]) \leftarrow \text{ordered}([B | Bs]), \text{clear}(B).\]
makeTower procedure - tests inferable concepts about perceived state of world

makeTower([B])..{
    on(B,tbl), clear(B) ⇒ {}.
    true ⇒ moveTo(B, tbl).
}. makeTower([B1, B2 | Bs])..{
    tower([B1,B2|Bs]) ⇒ {}.
    ordered([B1,B2|Bs]) ⇒ unpile(B1).
    tower([B2|Bs]) ⇒ moveTo(B1, B2).
    true ⇒ makeTower([B2|Bs]).
}. 
moveTo procedure -
tests percepts and inferred belief

moveTo(B, Place) . { 
  on(B, Place) ⇒ nil.
  clear(Place), holding(B) ⇒ putdown(B, Place).
  clear(B), clear(Place), not holding(_) ⇒ pickup(B).
  holding(B') ⇒ putdown(B', tbl).
  clear(Place) ⇒ unpile(B).
  true ⇒ unpile(Place).
}

putdown, pickup primitive arm actions
makeClear procedure

unpile(B) ..{

clear(B) ⇒ nil.

on(B’,B) ⇒ moveTo(B’,tbl).
}

Mutual recursion between `unpile` and `moveTo` with destination `tbl` (the table) will move to the table all blocks piled on top of B.
Elasticity of such TR procedure stacks

- As percepts are updated inferable beliefs change
- Changes can cause higher level TR procedure to abandon a call to lower level procedure
- Each level of TR program is elastic
- If another agent undoes the effect of earlier actions, TR procedure call stack will auto-reconfigure to try to re-achieve that effect
- Similarly, if another agent helps, the TR call stack will re-configure to skip actions
- TR programming thus well suited to human-robot, or multi-robot co-operative applications, or dynamic environments
Multi-tasking robots

• TR procedure suite is typically programmed for a single task
• An outer control layer, that allows selection of an approp. top level TR procedure to respond to a key event, such as a task execution request or goal to achieve, allows us to program multi-tasking robots
• TR block stacker has several capabilities. It can build towers, or just unpile, or move a specific block to a specified location.
• Asking it to build a tower comprising [1,3,2] can be viewed as a request to
  \[\text{achieve(tower\([1,3,2]\))}\]
  I.e. to invoke an appropriate procedure so that eventually
  \[\text{tower\([1,3,2]\)}\]
  can be inferred from its percepts
• BDI agent architectures are multi-tasking and event driven, where events lead to the invocation of plans and timesharing between plans is the norm.
A conventional BDI agent architecture

Event Reaction Rule: $e::\text{context} \rightarrow \text{plan}$
Key advantages of BDI architecture

• Plans have sub-goals, which leaves selection of plan for the sub-goal to runtime
  – more flexible than calling a named TR procedure

• Has plan failure recovery mechanism
  – typically leads to selection of alternative plan taking into account the belief context after the failure

• Allows a plan $P'$ invoked for a prior event $e'$ to be suspended
  – when we want to invoke a plan $P$ for a higher priority event $e$ where plans $P, P'$ are incompatible
  – $P$ normally resumed when $P'$ terminates
Drawbacks of conventional BDI

• Plans are sequential and inelastic
• Robustness can only be obtained using the plan failure mechanism
  – if effect of an action is undone fail plan
  – select an alternative plan for new context
• Similarly, opportunity seizing behavior needs to be handled by dropping existing plan and adopting shorter plan for the new belief context
• Can overcome drawbacks by adopting a multi-threaded architecture using TR plans
New event reaction rules

• Event reaction rules have the form:

\[ e:: \text{context} \rightarrow \text{TRPlan} \]

• context, can query percept beliefs, inferable beliefs and the record of current intentions
  - context is a belief and intention store query which is feasibility condition for use of the TRPlan

• Events are data values of the form:

\[ \text{add}(B), \text{del}(B), \text{start}(\text{TRPlan}), \text{stop}(\text{TRPlan}), \text{achieve}(B), \text{drop}(B) \]

  – add(B), del(B)  significant updates of the belief store
  – start(TRPlan), stop(TRPlan)  invoke and terminate named plans
  – achieve(B), drop(B)  goal related events
Example reaction rules

\texttt{achieve(tower(Blocks)) :: diffBlocks(Blocks) \rightarrow makeTower(Blocks).}

Needs extra def in KB:

\texttt{diffBlocks([]).}
\texttt{diffBlocks([E,,L]) \leftarrow block(E), \neg E \in L, \text{diffBlocks}(L).}

\texttt{achieve(clear(B)) :: block(B), on(_,B) \rightarrow unpile(B).}
TR plan actions

• TR plans are TR procs. with additional rule actions:
  
  **achieve(B)**  find and invoke an approp. plan for **achieve(B)**  
  using event reaction rules (indirect call of a TR plan)

  **post e**  post e to event store
  any invoked plan executed independently
  enables a TR plan to spawn an independent thread

  **exit**  exit this plan
  **fail**  fail this plan
  will result in attempt to find an alternative plan if failed
  plan invoked in response to an **achieve** event
Example TR plan

\[ \text{moveTo}(B, \text{Place}) \Rightarrow \{
\]

- \( \text{on}(B, \text{Place}) \Rightarrow \text{nil} \).

- \( \text{clear}(\text{Place}), \text{holding}(B) \Rightarrow \text{putdown}(B, \text{Place}) \).

- \( \text{clear}(B), \text{clear}(\text{Place}), \text{not\_holding} \Rightarrow \text{pickup}(B) \).

- \( \text{holding}(B') \Rightarrow \text{putdown}(B', 0) \).

- \( \text{clear}(\text{Place}) \Rightarrow \text{achieve}(\text{clear}(B)) \).

- \( \text{true} \Rightarrow \text{achieve}(\text{clear}(\text{Place})) \).

\}[.]

achieve subgoal allows flexibility if more than one plan for the goal
Multi-threaded cognitive agent architecture - AgentMT(TR)

- Current percepts, + KB rules & facts
- Library of event reaction rules & TR plans
- Record of current intentions
- Event handler
- Message handler
- Percept handler
- TR plan
- TR sub-plan
- Action arbiter

Events flow through the system, interacting with the current percepts, knowledge base rules and facts, a library of event reaction rules and TR plans, and a record of current intentions. The system includes components for handling messages, percepts, events, TR plans, TR sub-plans, and managing actions through an action arbiter.
Event handling thread

1: Respond to all drop(B) or stop(TRPlan) events

2: Find event e with an applicable plan instance P such that:
   - P has not failed for e
   - P is not incompatible with plan P’ of an active higher priority intention (e’, P’)

3: Remove e from event store
   - suspend active lower priority intentions (normally resumed when new intention for e terminates)
   - launch thread for new intention (e, P)
   - if P fails, return e to event store, record P as failed for e

4: Repeat
Example of need for intention suspension

- Periodically can collector needs to recharge its battery
- Event that triggers its recharge plan is belief update event: `add(battery_low)` generated as a result of a battery charge sensor reading

![Diagram showing can collector and recharging plate]

- `add(battery_low)` has higher priority than `do(collect_cans)` event
- Can collecting plan is suspended and resumed when charging plan terminates
Example of concurrent intentions

• A communicating robot can be executing a conversation plan for each robot that has initiated a conversation with it
• Trigger event is first communication received from the other robot
• Each such conversation accesses and atomically updates the shared belief store of the agent
• New beliefs added by these threads indirectly affect the behaviour of the non-conversation plans
Conclusions

• AgentMT(TR) agents are deliberative and reactive
• Plans are robust
  – can exploit opportunity
  – can recover from setbacks
• Agents are multi-threaded
• Percepts and messages continuously processed
• Being used at Imperial for cognitive robot control, and as a general agent architecture
• About to explore use for a new cognitive architecture modeling human concurrent problem solving