Model-based Hierarchical Control of a Mobile Manipulation Platform
Outline

● Motivation
● Related Work
● Key TREX Concepts
● Field Trials with PR2
● Next steps
The Robot Programmer's Dilemma

- Many components to integrate & manage
- Often, there are interactions among components
- Often many alternatives to consider to achieve good behavior
- Choices made in-situ
- Murphy's Law is true
  - Everything that can go wrong will
Case Study – The PR2 Triathlon

**DRIVE**
- Topological/Metric Map
- Regions and Connectors
- Doorways, Doors and Outlets

**OPEN DOOR**
- Clear, Open, Locked, Latched
- Door/Handle Detection & Manipulation
- Whole-body Control

**RECHARGE**
- Plug & Outlet Detection
- Plug Manipulation
- Visual Servoing

10 Recharge Goals distributed around the building, to be achieved in under 2 hours
Case Study – The PR2 Triathlon

- Long term planning for top-level goals
- Large number of components
  - 25 robot action primitives + other nodes
  - 5 mechanisms - arm, base, head, tilt_laser, gripper
  - 20 low-level, real-time controllers
- Concurrent actions
  - while moving through the door, push it
- Precedence constraints
  - push_door must start before move_base_door
Case Study – The PR2 Triathlon

• Safety Constraints
  – The arm must be stowed while driving
  – The plug must be on the base while driving

• Configuration Interactions
  – The tilt laser must be running while driving
  – *unlatch_handle* requires
    • r_arm_cartesian_wrench_controller
    • r_arm_cartesian_twist_controller
    • r_arm_cartesian_pose_controller
    • r_arm_cartesian_trajectory_controller
Case Study – The PR2 Triathlon

- **Discrete States**
  - Controller is up or down
  - Plug stowed or not

- **Continuous States**
  - Battery level (M1)
  - Base pose

- **Uncertainty**
  - Action duration
  - Action SUCCESS or FAILURE
  - Action Feedback Parameters
Related Work

Procedures
Scripts
Plans

Model

Deliberative Planner

Goals

Reactive Planner

Reactive Executive

Deliver Actions

Skills

Behaviors

Commands

Structured Reactive Controllers (Beetz00)
IDEA (Muscettola02)

Remote Agent (Muscettola98)
Saphira (Konolige96)

3T (Bonasso97)
Kirk (Kim01), Titan (Williams03)

TREX (McGann08*)

Evolved from IDEA
Planning is ubiquitous
Divide-and-conquer by design
Shared models at all levels
Implementation based directly on EUROPA (Frank03)

Add 'Deliberative' Planner
Limited Scope for planning
Lose integration

Expand scope of Planning
Shared models
Tighter Integration

PRS (Georgeff89)
PRS-Lite (Meyers96)
Colbert (Konolige97)
RAP (Firby87)
TDL (Simmons98)

Reactive Executive
No Planning

Implementation based directly on EUROPA (Frank03)
Key TREX Concepts

• Timelines, Tokens & Timeline-based Execution
• Sense-Plan-Act and the Wonder Widget
  – The reusable *DeliberativeReactor*
  – *Planning can be trivial*
• Divide-and-conquer
  – Partitioning and Composition
Timeline-based Execution

- A **Timeline** describes the evolution of a state variable in the past, present and future.
- A **Plan** is thus a desired trajectory for a state variable, captured as a timeline.
- A **Timeline** is naturally mapped to executing code.

---

A temporally flexible, partial plan (move_base)

Client-side

Server-side

A grounded execution history (move_base)
The Wonder Widget
A Teleo-Reactor

- A general purpose component that integrates planning and execution
- Continuous Sense-Plan-Act loop
- Can span a spectrum of reactive and deliberative behavior
- Leverages automated planning to augment programmer
- Renders programs model-compliant by construction
Divide-and-Conquer

• A *software* agent can be viewed as a collection of concurrent *control loops* (a.k.a. [Teleo-]Reactors).

• Reactors are differentiated by
  – functional scope – which state variables to consider
  – temporal scope - how far to look ahead

• Reactors have internal and external state
  – they share as little state as possible
  – they share as much state as necessary
  – they can task each other, allowing composition

• Composition of Wonder-Widgets & other variants as needed

Application of good engineering principles to achieve scalability and manage complexity.
Divide-and-Conquer (AUV)

Mission Manager

Science Goals = \{\text{Idle}(\ldots), \text{Transect}(\ldots), \ldots\}

Path = \{\text{Go}(\ldots), \text{At}(\ldots)\}

Navigator

Path = \{\text{Go}(\ldots), \text{At}(\ldots)\}

\begin{align*}
\text{Attitude}(\text{pitch, roll, yaw}) \\
\text{Position}(\text{lat, long, depth})
\end{align*}

Commands = \{\text{Idle, ascend}(\ldots), \text{descend}(\ldots), \ldots\}

Exec

Commands = \{\text{Idle, ascend}(\ldots), \text{descend}(\ldots), \ldots\}

\begin{align*}
\text{Position}(\text{lat, long, depth}) \\
\text{Attitude}(\text{pitch, roll, yaw})
\end{align*}

\(\pi = 22 \text{ hrs}\)
\(\lambda = 60 \text{ s}\)

\(\pi = 5 \text{ s}\)
\(\lambda = 1 \text{ s}\)

\(\pi = 1 \text{ s}\)
\(\lambda = 0 \text{ s}\)

Courtesy: Frederic Py, MBARI
A Running Example (AUV)

Courtesy: Frederic Py, MBARI
Case Study – The PR2 Triathlon

**DRIVE**
- Topological/Metric Map
- Regions and Connectors
- Doorways, Doors and Outlets

**OPEN DOOR**
- Clear, Open, Locked, Latched
- Door/Handle Detection & Manipulation
- Whole-body Control

**RECHARGE**
- Plug & Outlet Detection
- Plug Manipulation
- Visual Servoing

10 Recharge Goals distributed around the building, to be achieved in under 2 hours
Divide-and-Conquer (PR2)  
(Partitioning and Composition)

- Can close control loops locally
- Can recover locally
- Can simplify by separating out separable parts
- Can and did develop & test incrementally

Actually, each external action is an instance of a *TeleoReactor* which is an adapter to a ROS *robot action*

The Reactor Topology for the PR2 Triathlon
A State Machine for Door Opening
class DoorController extends AgentTimeline {
  // Idle
  predicate Inactive{}
  // Checking the doorway to see if we can drive thru without having to contact the door
  predicate CheckingDoorway{}
  // For positioning the robot at the approach point to the door from which it can detect the door
  predicate Positioning{}
  // For detecting the door.
  predicate DetectingDoor{}
  // For detecting the handle.
  predicate DetectingHandle{}
  // For moving towards the door. Used prior to grasping, and/or opening the door
  predicate ApproachingDoor{}
  // Just to grab the handle. Relevant if the door is latched
  predicate GraspingHandle{}
  // Just to unlatch the handle. Relevant if the door is latched
  predicate UnlatchingHandle{}
  // To simply drive thru the doorway with no door contact. Requires a clear path
  predicate DriveThruDoorway{}
  // To actually physically open the door which requires base and arm motion. The base must drive
  // through while contact the door either with or without the handle
  predicate OpenDoor{
    bool use_handle;
  }
  // Constructor
  DoorController(){ super(Internal, “Inactive”); }
};
Simple Decomposition Rule for \textit{DetectingDoor}

\begin{verbatim}
DoorController::DetectingDoor{
    // Detect door action
    contains(detect_door.Active cmd);
    ends cmd;
    // if we succeed, then depending on the latch state we can go straight for the door
    // or we need to detect the handle first. If it fails, maybe the door has been opened!
    // We might as well check the doorway for a clear path again
    if(isSucceeded(cmd.object) == true){
        cmd meets(detect_door.Inactive cmd_feedback);
        if(cmd_feedback.latch_state == LATCH_STATE_UNLATCHED){
            meets(ApproachingDoor);
        } else {
            meets(DetectingHandle);
        }
    } else {
        meets(CheckingDoorway);
    }
}
\end{verbatim}
Decomposition for *OpenDoor*

```cpp
DoorController::OpenDoor{
    contained_by(doorman.Active);
    contains(move_base_door.Active cmd_move_base_door);
    cmd_move_base_door.max_duration <= MOVE_BASE_DOOR_DURATION_BOUND;
    cmd_move_base_door before(stop_action.Active cmd_stop_action);

    // Conditionally generate steps with or without the handle
    if(use_handle == true){
        contains(open_door.Active cmd_open_door);
        cmd_open_door starts_before cmd_move_base_door;
        cmd_stop_action.action_name == "open_door";
        ends cmd_open_door;
    } else {
        // Need to touch the door and then push it. Both must succeed
        contains(touch_door.Active cmd_touch_door);
        cmd_touch_door before(push_door.Active cmd_push_door);
        cmd_push_door starts_before cmd_move_base_door;
        isSucceeded(cmd_touch_door.object) == true;
        cmd_stop_action.action_name == "push_door";
        cmd_push_door.end == end;
    }

    // Disjunction for successor state
    if(isSucceeded(cmd_move_base_door.object)){
        meets(Inactive s);
        s.status == SUCCESS;
    } else{
        meets(OpenDoor s);
        s.use_handle == false;
    }
}
```
Generative Planning to Augment Programmer

RechargeController::Positioning{
    contained_by(recharger.Active rc);
    if(isTimedOut(rc.object)){
        duration == 1;
        meets(Inactive s);
        s.status == PREEMPTED;
    }
    else {
        contains(move_base.Active cmd);
        ends cmd;
        isSucceded(cmd.object) == true;
        cmd.max_duration <= MOVE_BASE_DURATION_BOUND;
        // Bind pose based on approach pose for the outlet
        map_get_outlet_approach_pose(cmd.x, cmd.y, cmd.z,
                                      cmd.qx, cmd.qy, cmd.qz, cmd.qw, rc.outlet_id);
        cmd.frame_id == rc.frame_id;
        meets(FindingOutlet);
    }
}

- Programmer specifies what should happen in the local context of the recharge controller state machine.
- The model and the planner **fill in details** based on model constraints.
### Quantitative & Qualitative Results

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive control rate</td>
<td>10 Hz</td>
</tr>
<tr>
<td>Total number of internal timelines</td>
<td>47</td>
</tr>
<tr>
<td>Total number of external timelines</td>
<td>66</td>
</tr>
<tr>
<td>Total number of EUROPA timelines</td>
<td>87</td>
</tr>
<tr>
<td>Mission duration</td>
<td>3799 secs</td>
</tr>
<tr>
<td>Total number of actions executed</td>
<td>494</td>
</tr>
<tr>
<td>Total number of action failures</td>
<td>29</td>
</tr>
<tr>
<td>Total number of planning cycles</td>
<td>907</td>
</tr>
<tr>
<td>Memory consumption of the executive</td>
<td>10 MB</td>
</tr>
<tr>
<td>Estimated model line count</td>
<td>1207</td>
</tr>
<tr>
<td>TREX CPU utilization (mean)</td>
<td>9.80%</td>
</tr>
<tr>
<td>TREX CPU utilization (std)</td>
<td>5.00%</td>
</tr>
</tbody>
</table>

**Explanation of timeline counts**
- 26 RCS internal state variables
- 21 TREX internal state variables
- Replicated timelines in EUROPA = 87 - 47 = 40
  \[ 87 = 66 + 47 - 26 \]

**Explanation of action statistics**
- Many actions for tilt_laser configuration and controller switching
- Failure implies action aborted or timed out
- Planning cycle is initiated whenever reactor receives a goal, or flaw entailed by the model

**Qualitative Assessments**
- The model size was very modest for a demonstration of this scale
- Computational resources were also quite modest
- Control rate of 10 Hz is quite 'snappy' for action transitions
- The model provides a very high-level way to program a robot
- The use of the model and plan database for augmenting programming is very powerful (simple planning – no search typically)
Thank you!

- NASA for ongoing collaboration on EUROPA, and making it available.
- MBARI
  - seeding open-source TREX code base with version developed for the AUV.
  - Kanna Rajan & Frederic Py for ongoing collaboration.
- Willow Garage for supporting the work.
References

- [Muscettola98] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, Brian C. Williams. Remote Agent: To Boldly Go Where No AI System Has Gone Before. AI 103(1-2):5-47
- [Simmons98] Reid Simmons, David Apfelbaum: A Task Description Language for Robot Control. IROS, 1998