Logic, Ontology and Planning: the Robot’s Knowledge
Lecture 5

Stefano Borgo

Laboratory for Applied Ontology (LOA), ISTC-CNR, Trento (IT)

ESSLLI course 2018
Sofia, Bulgaria
Today’s Lecture

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
Course Overview

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
Robotics

- History
- Agency
- Robot definition
- Robot classification
- Trends in robotics
Ontology

- Ontological analysis
- Ontologies
- The role of ontology in information systems
- The ontology toolkit
- The DOLCE foundational ontology
Cognitive Science

- Concept blending
- DOL, Hets, OntoHub
- Formalization of blending
- Image schemas
- Formalization of image schemas
Modeling

- Hierarchies (warning)
- Environment
- Context
- Resource
- Behaviors and Functions
Course Overview

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
The plant is composed of different automatic and manual machines devoted to perform loading/unloading, testing, repairing and shredding of PCBs and a reconfigurable transportation system connecting them. The transportation system is composed by 15 reconfigurable mechatronic components, called transportation modules (TM).
Printed Circuit Board (PCB)
Considering the internal structure of this TM, it is possible to define three different types of component:

1. the conveyor component;
2. the port component; and
3. the cross-transfer component.
Course Overview

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
In the treatment of engineering functions, the function is seen as an entity detached from the agent performing it:

- Functional representation: desired behavior of the device
- Functional basis: transformation of the input flows into output flows
- Functional concept: interpretation of a behavior of the device

In all these cases, the user of the device is *irrelevant*.

This is not so when the robot has to reason about what to do.
Engineering functions from the robot’s perspective

S. Borgo et al. “A planning-based architecture for a reconfigurable manufacturing system.” ICAPS 2016
Engineering functions talk about homogeneous changes (or states) and are subdivided by types of change.

- To join is a change that reduces the number of (topological) objects by connecting two or more of them.
- To channel is a change in which an object changes location.
- To reclassify is a change in which an object changes status.
- To store is a state in which an object or a quantity is maintained in a certain position.

These functions only in some cases correspond to the full change an agent has to realize.
Tasks vs functions

Tasks are descriptions of changes in the world that are relevant for the agent. Here relevant means that the task’s completion marks an important step toward the realization of the goal.

- To cut with a shear is a task that integrates a function (to branch) and a way of execution (cut with the shear).
- To squeeze a lemon is a task that integrates a function (to change magnitude) and the object (the lemon).
- To connect a plug is a task that integrates several functions (to channel, to join, to stabilize) and two objects (the plug and the outlet).

... 

Most relevant tasks are combinations of engineering functions and (instances of) objects.
Manipulation tasks

From “Classifying Compliant Manipulation Tasks for Automated Planning in Robotics”, D. Leidner et al., IROS 2015
Aligning functions and manipulation tasks

- **FUNCTION** (as effect)
- **ACTION**
  - CONVERT
  - BRANCH
  - JOIN
  - CHANGE 
  - OVER
  - RECLASSIFY
  - CHANNEL
  - CHANGE MAGNITUDE
  - STORE
  - COLLECT
  - RELEASE
  - STABILIZE
  - INCREASE
  - DECREASE
Tasks can be very specific: wiping asks
Aligning functions and wiping tasks

- FUNCTION (as effect)
  - ACTION
    - CONVERT
    - BRANCH
    - JOIN
    - CHANGE OVER
    - RECLASSIFY
    - CHANNEL
    - CHANGE MAGNITUDE
    - STORE
    - COLLECT
    - RELEASE
    - STABILIZE
    - INCREASE
    - DECREASE
Course Overview

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
Physical robot, Knowledge module, Execution module
Knowledge module (KBCL)

The overall cognitive architecture resulting from the integration of a Knowledge Manager (top left), which contains the built-in know-how of the agent on its structure and the environment, and a Deliberative Controller, which constitutes a “classical” a plan-based control architecture.
A plan-based control architecture (deliberative controller) has three layers:

1. a deliberative layer which provides the agent with the capability of synthesizing the actions needed to achieve a goal (i.e., the Planning Framework);

2. an executive layer which executes actions of a plan and continuously monitor their actual outcome with respect to the expected status of the system and the environment (i.e., the Monitor and the Executor); and

3. the mechatronic system (and its functional processes) which represents the system and the environment to be controlled (i.e., the Mechatronic Module and the related transportation system).

The deliberative controller realizes a sense-plan-act cycle by means of a Planning Framework (top right) and an Executor system (bottom right). In our implementation both modules use timeline-based technology.
The Monitor and the Executor are responsible of “physically” interacting with the TM by sending commands and receiving signals about their execution (i.e., feedbacks).

The Executor sends commands to the TM according to the actions that compose the plan to be executed.

The Monitor receives signals about the (either positive or negative) outcome of the execution of such commands from the TM and checks whether the actual status of the mechatronic system complies with the plan.
The Deliberative Controller relies on a static planning model which completely characterizes the capabilities of a TM and the associated working environment.

However, such a model is not capable of dynamically capturing changes in the configuration of the transportation system such as, e.g., changes concerning the local topology of a TM or changes concerning the internal configuration of a TM. These changes affect the agent’s capabilities.

The Knowledge Manager enhances the flexibility of the Deliberative Controller by dynamically generating planning models.
Knowledge module (KBCL): the initial steps

When the TM is activated, the Monitor collects the raw data from the Mechatronic Module with which a knowledge processing mechanism (1) initializes the KB (it adds the instances that represent the actual TM’s state) (Point 1) and (2) dynamically generates the control model providing a first planning specification (Point 2). Then the planning system generates a production plan (Point 3) and the plan execution is performed through the executive system (Point 4).

When the Monitor detects a change in the structure of the agent and/or its collaborators (e.g. a total or partial failure of a sensor/actuator or of a neighbor), the KBCL process starts a reconfiguration phase (Point 5) entailing the update of the KB, and starting a new iteration of the overall loop.

The KB is updated only when changes prevent or possibly deteriorate execution.
Course Overview

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
Each KB is specific to the agent. The management of such a KB relies on a knowledge processing mechanism implemented by means of a Rule-based Inference Engine which leverages a set of inference rules to generated and updated a KB of an agent.
Knowledge mechanism

This mechanism involves two reasoning steps:

1. the low-level reasoning step (→ local working environment)
   it is about the components that actually compose the agent’s structure (e.g., ports, conveyors, etc.), and the associated collaborators. It relies on the internal and local contexts of the ontology and a set of classification rules.

2. the high-level reasoning step (→ working environment and functional capabilities)
   it relies on the taxonomy of functions and the capability inference rules to complete the knowledge processing mechanism. It works on the KB developed by the previous step (internal and local context of the agent).
Course Overview

1. Previous lectures
2. Our guiding scenario
3. Engineering function vs tasks
4. The architecture and the information flow
5. Knowledge processing mechanism
6. Knowledge generation for the planner
Leveraging on the ontology and contexts

The **local context** provides:
- list of ports/interfaces
- list of internal states
- list of engines/actuators

The **ontology** and the **global context** provide general information, e.g.:
- ports are locations
- conveyors are connectors of locations
- connection is transitive for the Channel (transportation) function

This suffices to generate all the possible ways for a transportation agent to execute a Channel function.
Contextual knowledge: internal, local, general

Elaboration of data received from the Diagnosis Module
Inferring collaborators of a TM (low-level reasoning)

\[
ROBOT(r) \land PORT(p) \land hasLoc(p, l_p, t) \land ROBOTPART(p, r, t) \land hasOpStat(p, active, t) \land ROBOT(c) \land hasLoc(c, l_c, t) \land connection(l_p, l_c, t) \rightarrow hasCollab(r, c, t)
\]
Inferring collaborators of a TM (low-level reasoning)

\[
\text{ROBOT}(r) \land \text{CONVEYOR}(c_1) \land \text{hasOpStat}(c_1, \text{active}, t) \land \text{COMPONENT}(c_2) \land \text{COMPONENT}(c_3) \land \text{hasLoc}(c_1, l_1, t) \land \text{hasLoc}(c_2, l_2, t) \land \text{hasLoc}(c_3, l_3, t) \land \\
\text{connection}(l_2, l_1, t) \land \text{connection}(l_1, l_3, t) \land \rightarrow \text{hasCapacity}(r, f) \land \text{CHANNEL}(f) \land \text{cStart}(f, l_2) \land \text{cEnd}(f, l_3) \land \text{cConnect}(l_2, l_3)
\]
The rational of the rule

$\text{ROBOT}(r) \land \text{CONVEYOR}(c_1) \land \text{hasOpStat}(c_1, \text{active}, t) \land \text{COMPONENT}(c_2) \land \text{COMPONENT}(c_3) \land \text{hasLoc}(c_1, l_1, t) \land \text{hasLoc}(c_2, l_2, t) \land \text{hasLoc}(c_3, l_3, t) \land \text{connection}(l_2, l_1, t) \land \text{connection}(l_1, l_3, t) \land \rightarrow \text{hasCapacity}(r, f) \land \text{CHANNEL}(f) \land \text{cStart}(f, l_2) \land \text{cEnd}(f, l_3) \land \text{cConnect}(l_2, l_3)$

This rule takes the functional interpretation of the \textit{CONVEYOR} category as the set of components that can perform channel functions.

If a conveyor component connects two components of the TM through its spatial location (clause $\text{connection}(l_2, l_1, t) \land \text{connection}(l_1, l_3, t)$), then the conveyor can perform a primitive channel function between the components’ locations.

Moreover, the $\text{cConnect}(l_2, l_3)$ (complex channel function) is a transitive predicate which allows to connect different channel functions. If two spatial locations are connected through the $\text{cConnect}$ predicate then there exists a composition of primitive channel functions that “connect” them.

\textbf{Note:} a primitive channel function involves components of one TM only.
\begin{align*}
RBO\text{BOT}(r) & \land RBO\text{BOT}(rc_1) \land RBO\text{BOT}(rc_2) \land hasCollab(r, rc_1, t) \land \\
hasLoc(rc_1, rl_1, t) & \land hasCollab(r, rc_2, t) \land hasLoc(rc_2, rl_2, t) \land PORT(c_1) \land \\
hasOpState(c_1, active, t) & \land hasLoc(c_1, l_1, t) \land PORT(c_2) \land hasOpState(c_2, active, t) \land \\
hasLoc(c_2, l_2, t) & \land connection(l_1, rl_1, t) \land connection(l_2, rl_2, t) \land cConnect(l_1, l_2) \rightarrow \\
hasCapacity(r, f) & \land CHANNEL(f) \land cStart(f, rl_1) \land cEnd(f, rl_2)
\end{align*}
Variables for the timeline-based planner

From the control perspective, it is possible to identify three different classes of state variables:

1. functional state variables;
2. primitive state variables; and
3. external state variables.

Functional state variables model a physical system as a whole in terms of the high-level functions it can perform (notwithstanding its internal structure).

Primitive state variables model the physical and/or logical elements that compose a physical system. In particular, these state variables model the elements we must actually control to execute high-level functions.

External state variables model elements of the domain whose behavior is not directly under the control of the system. For example, these variables model conditions that must hold to successfully perform operations.
Generating the model

The model generation procedure

1: `function buildControlModel(KB)`
2:     // extract agent’s information and initialize the P&S model
3:     agent ← getAgentInformation(KB)
4:     model ← initialize(KB, agent)
5:     // define components of the model
6:     svs ← buildFunctionalComponents(KB, agent)
7:     svs ← buildPrimitiveComponents(KB, agent)
8:     svs ← buildExternalComponents(KB, agent)
9:     // build the set of task decomposition rules
10:    S ← buildSynchronizationRules(KB, agent)
11:    // update the P&S model
12:    model ← update(model, svs, S)
13:    `return model`
14: `end function`
function BUILD_FUNCTIONAL_COMPONENTS(KB, agent)
    // initialize the list of functional variables
    sv$s ← \emptyset$
    // get types of functions according to the Taxonomy in the KB
    taxonomy ← getTaxonomyOfFunctions(KB)
    for all function ∈ taxonomy do
        // check if the KB contains individuals of function
        capabilities ← getCapabilities(KB, agent, function)
        if ¬isEmpty(capabilities) then
            // create functional variable
            sv$s ← createFunctionalVariable(function)$
            // add a value for each "inferred" capability
            for all capability ∈ capabilities do
                sv$s ← addValue(sv, capability)$
            end for
            // add created state variable
            sv$s ← addVariable(sv$s, sv)$
        end if
    end for
    return sv$s$
Generating primitive variables

```
1: function BUILDPRIMITIVECOMPONENTS(KB, agent)
2:   svrs ← Ø
3: // get agent’s operative components
4:   components ← getActiveComponents(KB, agent)
5:   for all component ∈ components do
6:     // check if component can perform some functions
7:       capabilities ← getCapabilities(KB, component)
8:       if ¬IsEmpty(capabilities) then
9:         // create primitive variable for component
10:        sv ← createPrimitiveVariable(component)
11:       // check component’s functional capabilities
12:       for all capability ∈ capabilities do
13:         sv ← addValue(sv, function)
14:       end for
15:       svrs ← addVariable(svrs, sv)
16:     end if
17:   end for
18:   return svrs
19: end function
```
Generating external variables

The external variable generation procedure

```python
1: function buildExternalComponents(KB, agent)
2:     svs ← ∅
3: // get agent’s collaborators
4:     collaborators ← getCollaborators(KB, agent)
5: for all collaborator ∈ collaborators do
6:     // create an external variable to model the collaborator
7:     sv ← createExternalVariable(collaborator)
8: // model the possible behaviors of collaborators
9:     states ← getOperativeStates(collaborator)
10: for all state ∈ states do
11:     sv ← addValue(sv, state)
12: end for
13: svs ← addVariable(svs, sv)
14: end for
15: return svs
16: end function
```
Generating synchronization rules

The synchronization rule generation procedure (inter-component causal and temporal requirements for the plan to be successful, they describe dependencies between the variables of a planning domain and may determine a hierarchy among them.)
The generated timeline-based model

A partial timeline-based model (TM equipped with one cross-transfer unit)
Implementation details

The ontology is provided in FOL but FOL is used only for preprocessing (primarily to ensure conceptual consistency).

Most of the inferences at runtime are done in the OWL version of the KB (we exploit primarily the contextual classification and relationships).

The ontology editor Protégé has been used for KB design and testing.

For runtime reasoning in the Knowledge Manager, we have used the Ontology and RDF APIs and Inference API provided by the Apache Jena Software Library.

Finally, the Deliberative Controller has been realized by means of the GOAC architecture whose deliberative features are implemented by means of APSI-TRF.
Modeling an adaptive autonomous agent aware of its capabilities
Main references

Most of the material we have seen today comes from:


If you want to use the timeline planner PLATINUm, download it here:
https://github.com/pstlab/PLATINUm

A basic introduction can be found here:

and a formal presentation is here:
End of Lecture 5 and of this course!