Paper Handy

by Handy Wicaksono

Submission date: 05-Jul-2022 08:18PM (UTC+0700)
Submission ID: 1866894303
File name: MI-HLC2020_paper_22.pdf (4.03M)
Word count: 9413
Character count: 48250

1
Logic-Based Robotics

1.1 Robot Software Architectures

Robot software architectures are often characterised as hierarchical systems where the lower layers handle motor control and feature extraction from sensors, and the higher layers deal with problem solving and planning. Because the lower layers interact directly with the environment, data are usually continuous and noisy, and control decision operate on short time scales; whereas the upper layers work on longer time scales and usually assume the world is more discrete and deterministic.

Figure 1.1 shows an adaptation of Nilsson's (2001) triple tower architecture. Sensors, such as cameras, LIDAR, ultrasonics, microphones and tactile sensors deliver their raw data into a working memory. The sensor data are interpreted by programs in the "perception tower", transforming the into successively more abstract representations. For example, a raw camera image may become a collection of edge points, that then become sets of lines, then corners, then while objects. Relations between these objects may then be added to the working memory, which also represents the robots world model. The "action tower" contains planners and plan libraries that produce actions to be sent as motor commands to the robots actuators. Like the perception tower, the action tower is also hierarchical. At it's highest level, a task planner gen-

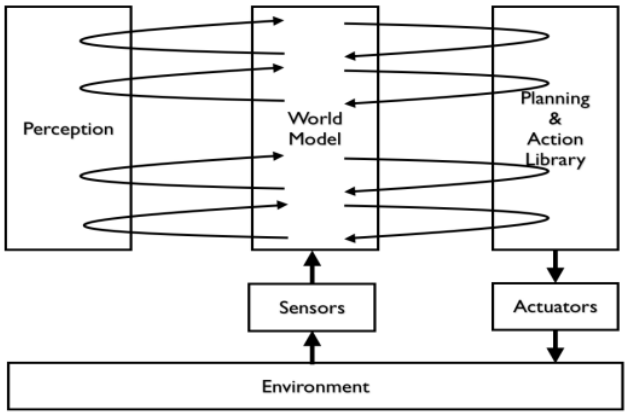


Fig. 1.1 Nilsson's triple tower as an example of a hierarchical robot software architecture

erates high-level actions, such as, "pick up the cup", which must be translated into trajectories for the mobile robot platform and arms. This requires motion planning in a continuous domain, and monitoring to perform error correction and, if necessary, instigate replanning if the original plan fails.

Early attempts at building integrated robot systems tended to focus on the higher levels of the architecture but often failed because they were unable to handle the uncertainty inherent in the physical world. Recent progress in robotics owes much to the development of probabilistic and behaviour-based methods that overcome some of the shortcomings of the early approaches. Deep learning systems (Bengio *et al.*, 2015), to some degree, automatically create hierarchical representations, at the cost of explainability. However, high-level symbolic reasoning and learning still have important roles to play. In addition to being more readable, they are capable of more powerful generalisations.

In the following work, we give several examples of how symbolic and sub-symbolic systems can be combined to take advantage of the strengths of each approach. In perception, Inductive Logic Programming (ILP) (Muggleton, 1991), can be used to learn descriptions of classes of objects and to find relations between objects. We also discuss examples of learning plans and behaviours for robots. Relational learning is used to acquire an abstract model of robot actions that is then used to constrain sub-symbolic learning for low-level control. Coincidentally, the first use of the term "deep learning" was by Dechter (1986) in a constraint logic programming setting. Models can be variously expressed in the classical STRIPS representation, as qualitative models or as teleo-reactive programs. We give examples of each in the context of the RoboCup Rescue, @Home and soccer Standard Platform League competitions.

1.2 Relational Learning in Robot Vision

Farid (2014) describes an ILP system that learns an object classifier for an autonomous robot in an urban search and rescue operation. The primitive input to the classifier is a RGB-D image[1], representing a partial view of the environment. A range image can be transformed into a set of 3D coordinates for each pixel, producing a point cloud. From that point cloud, we can extract features that will assist learning. To illustrate the process, Figure 1.2 shows a point cloud of a staircase with four steps. Using a plane detector, the point cloud is segmented into planes that are identified by unique colours. Planes are useful features in built environments, including indoor urban search and rescue for identifying floors, walls, staircases, ramps and other terrain that the robot is likely to encounter. An ILP system is used to discover the properties and relationships between the planes that form an object, representing them as Prolog clauses.

This method is most closely related to (Shanahan, 2002) who used a logic program as a relational representation for 3D objects in 2D line drawings, and abduction is used in object recognition. We have extended this representation, replacing the 2D lines with 3D planes. As mentioned, earlier, the first step in learning is to extract planes as the primitive features of the depth image. After extracting these features, sets of planes are labelled according to the class to which they belong. We have use to

[1] RGB refer to the colour channels per pixel and D is the depth

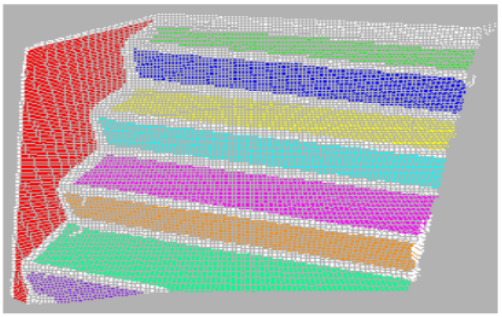


Fig. 1.2 Point cloud of a staircase

Fig. 1.3 Elements of a typical robot rescue arena

ILP systems, ALEPH ILP (Srinivasan, 2002) and MIL (Muggleton and Lin, 2013) to build a classifiers of objects such as staircases, ramps and other objects that occur in the rescue arena (Figure 1.3).

A plane is represented by the spherical coordinates of its normal vector (θ and ϕ) and other attributes are derived from the convex hull of the plane. These are the diameter and width of the convex hull and the ratio between these values. The plane's bounding cube is used to calculate the ratios between the three axes, two by two. The final plane feature is the axis along which the plane is most distributed.

After planes are found and their individual attributes are calculated, we then construct relations between each pair of planes. The first relation is the angle between the normal vectors of each pair of adjacent planes. The second is a directional relationship that describes how two planes are located with respect to each other in the point cloud view. For example, a plane may be located to the east of another, or one plane may cover another or be connected to another. Since planes exist in 3D space, we project the 3D view onto two 2D views and find spatial-directional relationships in each 2D view.

Figure 1.4 shows the point cloud segmentation with each region's convex hull and normal vector and the corresponding colour image. The red region, region 1, is the

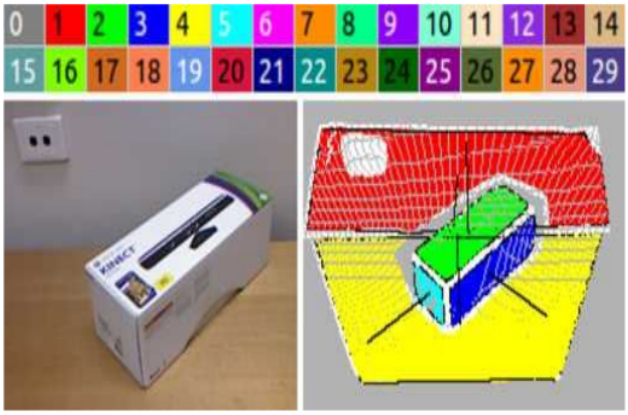


Fig. 1.4 Planes and plane features. The legend at the top shows the numerical label associated with each coloured plane.

wall, while the yellow region, region 4, is a desktop. For this example, we define a positive example for the class "box" that includes regions 2, 3 and 5, creating the predicate *box([pl00_2, pl00_3, pl00_5])*. That is, predicates of the form *box(+plane-set)* represent a set of planes in an image that form an object.

A set of predicates describes the individual attributes for each plane. All ratios and angles are discretised.

distributed_along(*plane*, *axis*) This is the long axis of the plane.
ratio_yz(*plane*, *ratio*) The aspect ratio of the planes bounding box in the yz plane.
ratio_xz(*plane*, *ratio*) The aspect ratio of the planes bounding box in the xz plane.
ratio_xy(*plane*, *ratio*) The aspect ratio of the planes bounding box in the xy plane.
normal_spherical_theta(*plane*, *angle*) Spherical coordinate of the planes normal vector.
normal_spherical_phi(*plane*, *angle*) Spherical coordinate of the planes normal vector.

Relations between pairs of planes are created for the angle between the normal vectors of two planes and the directional relationship for two adjacent planes from XY and XZ views. The relations can be one of: *north*, *south*, *east*, *west*, *connected*, *overs*.

angle(*plane1*, *plane2*, *angle*) The angle between the normal vectors of two planes.
dr_xy(*plane1*, *plane2*, *reln*) The directional relationship of two adjacent planes in the xy plane.
dr_xz(*plane1*, *plane2*, *reln*) The directional relationship of two adjacent planes in the xz plane.

The number of planes that form an object may differ. For example, staircases may differ in the number of steps:

```
staircase([pl02_06, pl02_08, pl02_10, pl02_11]).
staircase([pl02_01, pl02_03, pl02_04, pl02_05, pl02_06, pl02_08]).
staircase([pl02_04, pl02_05, pl02_06, pl02_08, pl02_10, pl02_11]).
```

An example of a learned classifier for the concept of "staircase", is shown below. It was constructed from 237 positive examples and 656 negative examples.

```
staircase(B):
    member(C, B), member(D, B), member(E, B),
    angle(E, C, '0~15'),
    dr_xz(D, C, east), dr_xy(E, D, south).
staircase(B) :-
    member(C, B), member(D, B), member(E, B),
    angle(D, C, '0~15'), angle(E, D, '90~15'), angle(E, C, '90~15'),
    distributed_along(E, axisX).
staircase(B) :-
    member(C, B), member(D, B), member(E, B),
    angle(E, D, '0~15'), angle(E, C, '0~15').
    dr_xy(D, C, south).
```

In these rules, the set of planes that constitute an object is denoted by variable B. Thus, *member(X, B)* means X is a plane from plane set B. The '~' represents $\pm$, e.g. 0 ± 15.

Rule 1 recognises plane set B as a *staircase* if it has two planes C and D that D is to the east of C in the XZ-view. It also contains plane E, which is approximately parallel to plane C. Also, the spatial-directional relationship between planes E and D in the XY-view is south. This rule covers 186 positive examples (above 78.48% of all positive examples) and no negative examples.

Rule 2 recognises B as a *staircase* if B has planes C, D parallel to each other and E is distributed mostly along X-axis and perpendicular to C and D. This rule covers 213 positive examples (above 89.87% of total positives) and no negative examples.

Rule 3 represents plane sets having at least three planes C, D and E where E is parallel to C and D, while D is to the south of C in the XY-view. This rule covers more than 53.58% of the positive examples.

A limitation of ALEPH is that it was not able to discover a recursive relation to represent a staircase. However, Metagol (Muggleton and Lin, 2013) and MIL, are capable of predicate invention and learning recursive definitions. The application of Metagol resulted in the following description of a staircase:

```
staircase(B) :-
    p_a(B).
staircase([X, Y, Z|B]) :-
    p_a([X, Y, Z]), staircase([Z|B]).

p_a(B) :-
    member(X, B), member(Y, B), member(Z, B),
    angle(X, Y, '90~15'), angle(X, Z, '0~15').
```

Here, the invented predicate can be interpreted as "step". That is, a staircase is a step or a step followed by another staircase. A step consists of three planes, two of which are roughly parallel and the third orthogonal to the parallel planes.

Here we see one of the most important reasons for using a symbolic description of visual concept. The recursive description of staircase is sufficiently general that it can recognise quite different kinds of staircase, without any further training examples, including circular stairs. This generality is difficult to achieve with any propositional or network based learner. It is true that we have had to do a substantial amount of feature engineering, however, once the primitive features are in place, predicate invention can introduce changes of representation, analogous to those created in a layered network, with the difference that the invented predicates can be interpreted by a human being.

In Nilsson's triple tower architecture, both the perception and action towers contain a hierarchy. In the perception tower, raw input data are presented at the bottom and programs in the tower progressively interpret the data in more abstract ways to obtain a world model that can be understood and reasoned about. For example, once the step and staircase concepts have been learned, a new scene may trigger the step rule, in a forward chaining manner. That, in turn, may trigger the staircase rules, and so on. Perception need not be entirely bottom-up as the higher level concepts may also create expectations of what can be seen, and so call lower-level routines to look for expected objects. In complex scenes, this is more efficient that trying to derive all possible properties and relations in the world. The lower-levels may consist of pre-built feature detectors, such as the plane detector, or the features may be learned a deep learning technique. A promising field of study is the combination of symbolic and sub-symbolic methods, taking advantage of the strengths of both approaches (Mao *et al.*, 2019).

1.3 Learning to Act

Like the perception tower, the action tower is also layered. The top-level generally plans how to act at an abstract level, while the lowest level deals with direct motor commands. Just as the perception tower is not strictly bottom-up, the action tower is not strictly top-down. The process may begin by a robot being given a goal to achieve. A planner, in the upper, deliberative layers, may generate a sequence of actions to perform. For example, if a person asks the robot to fetch a glass of water, it must first go to the kitchen, pick up a glass, place in under the tap, turn the tap, close the tap, return to the person, hand over the cup, etc. Each of these discrete actions are handed down to a set of motion planners to navigate the robot to the correct position, to control the arm and gripper, etc. Below that, each motor command must be monitored by a feedback control system to correct errors that inevitably occur. Errors may be so bad that the robot cannot complete its task and replanning is required.

This is the "classical" approach to robot action planning, but as Brooks (1986) pointed, often unexpected things happen at short time-scales and going through a complete planning sequence is too slow. For example, if a cat suddenly jumps out in front of the robot, by time the robot goes through its planning sequence, the cat, or the robot, may not be in very good shape. This the reason that control decision are made in layers. If a rapid response is required, like a reflex action, there may be a very short circuit between perception action, bypassing the upper layers of the architecture.

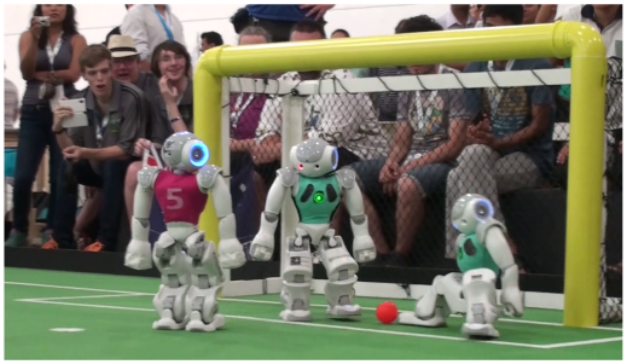


Fig. 1.5 RoboCup Standard Platform League

Long-term planning must assume that the world is reasonably predictable. In games like chess, the outcome of each move is completely predictable and the state of the world after a move can be fully observed. Therefore it is possible to look ahead and plan moves many steps in advance, subject to the opponent's response. In other domains, such as robot soccer (Figure 1.5), the world is highly unpredictable. The robot can easily slip or fall over. A kick may not go in the intended direction or the path of the ball may curve due to irregularities in the field. Uneven lighting may confuse the vision system. When robots are close to each other, it is difficult for the vision system to distinguish them. All of these things make planning almost impossible. The only recourse is to program behaviours as situation-action action rules, decision trees or state machines, that respond to the immediate state of the world without any, or very limited, lookahead. In such unpredictable domains, reinforcement learning or similar stochastic methods are most commonly used in an attempt to avoid laborious hand-craft of behaviours. Unfortunately, this kind of learning usually requires many trials before an adequate set of behaviours is produced. This means that we need a high-fidelity simulator because performing many trials on a real robot is very slow and the robot will almost certainly breakdown before the trials are complete. Alternatively, we can try to use, or learn, knowledge about the domain to significantly reduce the number of trials needed. This is the approach we take here.

Figure 1.6 the performance element of our architecture. As an example, consider a robot with bipedal locomotion. We can create a rough plan of how the robot should walk, such as: transfer weight to the left leg; left the right leg; swing the right leg forward. transfer weight to the right leg, and so on. This high-level sequence of actions must be translated into actual motor commands. We derive a set of constraints from the pre and post-conditions in the planning steps and pass them to a constraint solver. In this example, the constraints may limit the range of angles that the leg joints. Some kind of parameter refinement through trial-and-error learning may still be required, but the constraints reduce the parameter search space to a very large degree. By this method, Yik (2007, 2010), a bipedal robot was about the learn a stable walk in an

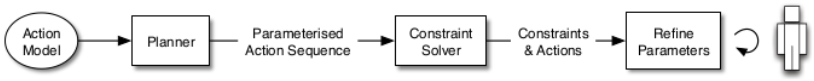


Fig. 1.6 A hybrid architecture combines different style of learning and constraint solving.

average of 40 trials. Considering that the robot had 23 degrees of freedom, this is a massive reduction in effort.

In this scheme, learning may occur in two different stages. The action models used by the planner may be programmed or learned, and parameter refinement may be achieved by reinforcement learning, genetic algorithms or hill climbing. In the following subsections we described several variants. At the planning stage, we can treat the world as being mostly discrete and deterministic and therefore use a classical planner and attempt to learn a STRIPS-like action model (Fikes and Nilsson, 1971). Alternatively, we can treat the world as continuous and either build a numerical or model, or in our case, a qualitative model (Kuipers, 1986).

1.3.1 Learning Action Models

Inductive Logic Programming can be used to learn STRPIPS-like action models. We will illustrate this with a system that learns to use and create tools for a robot (Brown and Sammut, 2013; Wicaksono and Sammut, 2018). An action model can be learned by observing another agent performing some task and then employing a form of explanation-based learning, using ILP. An "explanation" consists of matching an observed action to an existing action model. If an action cannot be explained, in this fashion, a new action model is created. A new tool can be created by a form of generalisation similar to that employed in Marvin (Sammut, 1981; Sammut and Banerji, 1986).

The state of a system is represented by a set of predicates, which may describe primitive features obtained from the vision system, including shape and pose of the objects. The state representation may also contain derived features, such as the alignment of a tool being along the same axis as the target object (e.g. if the tool is a hook used to pull an object).

As in STRIPS an action model includes the pre and post-conditions of an action. For example, a "position_tool" action may be described as.

```
position_tool(Tool, Box)
    PRE:          in_gripper(Tool)
    EFFECTS:  tool_pose(Tool,Box)

tool_pose(Handle,Hook,Box,State) :-
    attached_end(Handle,Hook,back),
    attached_side(Handle,Hook,Side),
    in_tube_side(Box,Tube,Side,State),
    in_tube_side(Hook,Tube,Side,State),
    touching(Hook,Box,back,State).
```

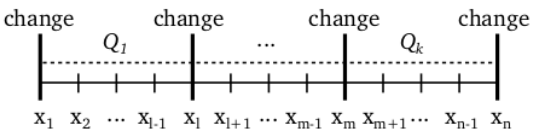


Fig. 1.7 A segment boundary is created when a qualitative variable, Q, changes, i.e. a condition that is currently true no longer holds, or a new condition becomes true. For example, when the underlying quantitative variable, x, undergoes a non-montonic change, e.g. it changes from increasing to steady or decreasing, the corresposning qualitative variable changes.

This describes a hook that is positioned inside a tube and behind an object that is to be pulled out of the tube.

An action model can be learned by observing the behaviour of another agent (Brown, 2009). The process is as follows:

1. Record a demonstration by a trainer.
2. Segment the continuous time series into discrete states and extract their features and relations.
3. Match the segments with existing action models.
4. If no match is found, construct a new action model using the unmatched segments.

Trace Recording. Camera images are captured and objects are recognised using methods described elsewhere (Wicaksono, 2020). Measurements consist of a stream of values of quantiative variables, such as position and orientation.

Segmentation of States. The stream continuous of continuous variable values must be divided into qualitatively meaningful segments. For example, we may be interested in the start and stop positions of a robot, but not the all the positions in between. If we regard position as a qualitative variable (Kuipers, 1986), then the qualitative value of that variable is the interval between the start and stop positions.

We define a *qualitative state*, denoted by Q_t, as the set of predicates that hold over some period. A qualitative state corresponds to a segment in a behavioural trace. Each time a state variable undergoes a qualitative change (e.g. change in motion or contact with another object), a segment boundary is created (Figure 1.7). The logical conditions that hold during a segment are conjoined to create a clause that describes the qualitative state.

Matching the Segments with Existing Action Models. To perform matching, we use a form of Explanation-Based Learning (EBL), a learning mechanism that finds a hypothesis that follows deductively from the background theory and also covers the training data (Mitchell *et al.*, 1986). In our case, an "explanation" consists of finding an action that could tranform one qualtiatve state into another, i.e. the acton explains the transition from segment to the next. A candidate action is one whose pre and post-conditions match the initial and qualitatve state. They match if the preconditions of an action A, denoted by $Pre(A)$, are theta-subsumed by Q_{i-1}, and effects of an action A, denoted by $Eff(A)$, are theta-subsumed by Q_i.

$$\exists A: \quad Pre(A) \preceq_\theta Q_{i-1} \wedge Eff(A) \preceq_\theta Q_i \tag{1.1}$$

Theta-subsumption is possible if and only if there exists a variable substitution θ such that $Pre(A)\theta \subseteq Q_{t-1}$.

If the system has a complete set of action models, this process should result in a sequence:

$$Q_1 \xrightarrow{A_1} Q_2 \xrightarrow{A_2} Q_3 \xrightarrow{A_3} \ldots \xrightarrow{A_n} Q_{n+1} \tag{1.2}$$

Building a New Action Model. If the system cannot find an action that connects a pair of qualitative states, the must system build a new action models that can complete the explanation. The precondition of the new action model is constructed so that it is satisfied by the first qualitative state in the pair. The postconditions, or effects, must be satisfied by the succeeding qualitative state.

Where the system is trying to learing how to use a tool, we assume that tool actions have two parts: an action that positions the tool, denoted by A_{pos}, and another action that applies the tool to solve the problem, denoted by A_{app}. Each part has its own action model, where its preconditions and effects are acquired from corresponding qualitative states.

$$\ldots \rightarrow Q_m \xrightarrow{A_{pos}} Q_{m+1} \xrightarrow{A_{app}} Q_{m+2} \rightarrow \ldots \tag{1.3}$$

We also create a predicate, *tool_pose*, to represent the tool's properties required for tool-use. This predicate appear in the effects of the positioning action and the pre-condition of the application action as follows. The constructed action model is almost certainly too specific as it is derived from a single example. The next step is to generalise the model and test it by the robot performing experiments.

Learning by Experimentation. The action model is generalised by generalising the *tool_pose* predicate. This is done by a variant of Mitchell's version space search (Mitchell, 1977). The version space, denoted by VS_{H,E^+,E^-} represents the subset of hypotheses from hypothesis space H that cover all positive examples E^+ (*hypothesis is complete*) but do not cover any negative examples E^- (*hypothesis is consistent*). Our definitions of coverage follow Muggleton (1991).

A generality-ordering on the hypotheses permits the version space to be represented by its boundaries: h_g (most general hypothesis) and h_s (most specific hypothesis). To learn the version space, Learning by experimentation can be carried out to generalise h_s and specialise h_g until a target hypothesis T, which is bounded by h_g^* and h_s^*, is found. Whenever there is a negative example, instead of specialisation, we perform negative-based reduction (NBR) (Muggleton and Feng, 1992) to find the properties that caused the failure. NBR originally eliminates redundant literals without reducing hypothesis coverage.

The system's initial most general hypothesis (h_g) covers all possible objects, while the most specific hypothesis (h_s) incorporates the tool's structure and pose acquired from the initial example. h_s is the bottom clause $\perp(e^+)$ that is constructed by saturation.

Experiment Generation. The robot tries to find the correct *tool_pose* hypothesis by experimenting with various objects and poses, enabling the system to generalise

tool_pose so that the action model can be applied under a broader range of conditions. An experiment consists of constructing an instance of the most specific hypothesis and using the resulting action model in a particular task.

In this domain, we distinguish between the predicates that describe a tool's structural properties and those that describe its pose, i.e. between its constant properties and the fluents associated with it. This construction requires the selection of the candaidate tool's structural properties. An object that matches the most structural properties in the *tool_pose* is chosen as the tool. More specifically, we test whether the structural properties of an object O satisfy all structural properties in h_g and most of the secondary ones in h_s. We require all structural predicates in h_g to be satisfied by the object If an object does not meet the requirement, its similarity score is assigned a value of zero. If it is passed, the similarity score is then computed as the number of a_o^j that match a_{hs}^i. We accumulate the score for each object and store its final score as the structural similarity score.

Having a correct tool is useless if it is not positioned correctly. To select the correct pose, firstly, we collect the pose, or spatial, literals in h_g then append to the hypothesis the spatial predicates in h_s. The predicates in h^{spa} are then used to construct constraints for a constraint solver. The intention here is to convert logical constraints, such as "behind" or "on the right" into numerical parameters that can be given to an inverse kinematic solver as a target position. Thus, "behind" can be interpreted as a range between an object and the back of the scene. In our case, we can formulate our problem to find the tool pose as a constraint satisfaction problem (CSP), which later on can be solved using a constraint solver such eCLiPSe (Apt *et al.*, 2007) or the CLP library in SWI Prolog (Triska, 2012). The constraint solver is then used to find the pose with respect to the constraints.

The experimental hypothesis must be tested in a tool-use experiment so we can decide whether the hypothesis should be generalised further or rejected. To be able to conduct a tool-use experiment, we start by running a planner to generate a sequence of actions that includes tool use. This plan is passed to an "executor", which is responsible for monitoring the states and deciding which action should be activated at a particular time. Abstract actions may be implemented using different techniques, e.g. inverse kinematics, constraint solving or visual servoing. the details of this process are described by Wicaksono (2020)

Generalisation. The results of an experiment, labelled as a positive example if the experiment succeeds or negative if it fails. These are used to generalise the *tool_pose* hypotheses. The learning algorithm finds a hypothesis (H), such that H is complete and consistent with respect to background knowledge (B) and examples (E).

As mentioned before, our hypotheses are bounded by h_g, the most general hypothesis, and h_s, the most specific hypothesis. The hypothesis space is developed by partially ordering the hypotheses using θ-subsumption (Plotkin, 1970). Plotkin (1970, 1971) uses θ-subsumption to construct the Relative Least General Generalisation (RLGG) of two clauses. The *Relative Least General Generalisation (RLGG)* of two clauses c_1 and c_2 is their least general generalisation $lgg(c_1, c_2)$ *relative* to background knowledge B.

Our learning algorithm is inspired by Golem (Muggleton and Feng, 1992), a bottom-up learning algorithm that uses the *RLGG* for generalisation. This method is suitable

for our purposes as we want to learn the characteristics of a useful tool. A top-down algorithm is better suited to finding a discriminatory classification rule. The original Golem algorithm generates the hypothesis by randomly picking pairs of positive examples, finding the *RLGG*s of those pairs, choosing the pair that gives the best coverage, storing the corresponding clause as a hypothesis, and eliminating positive examples that have been covered (sequential covering). Those steps are repeated and the generated clauses are added until the coverage stops increasing, or if all positive examples have been covered.

The clause generated by the $RLGG$ tends to be lengthy. Clause reduction eliminates redundant and irrelevant literals. Golem uses negative-based reduction (NBR), which tries to eliminate one literal at a time from the clause's body, checking if the result covers any negative examples. This process is repeated until every literal has been tested.

Unlike Golem, which learns from a batch of examples, our system learns incrementally, as only one positive example can be acquired in an experiment. This positive example is *saturated* with respect to the background knowledge, B. h_s is generalised by finding the least general generalisation (LGG) of itself and the saturated positive example. This generalisation is repeated whenever a positive example is acquired. We also use negative-based reduction to eliminate redundant literals.

Experimentation in Simulation and Real World. We conduct learning experiments in simulation and in the real world. A physics simulator is used to minimise learning in the real world, saving time and reducing the risk of damage to a physical robot. However, the experiments in the real world are still needed to validate the final result.

The algorithm for performing experiments in both worlds is simple. We conduct the learning experiments in a simulation, where each trial is repeated five times and the result (success or failure) that occurred more often is assumed to be the label for the example. After learning has finished in the simulator, the hypothesis and action models are used in real-world experiments. If the robot can perform tool use successfully, then the previous results are assumed to be valid and learning is complete.

It is not practical to perform exhaustive testing, therefore it is possible to learn an incorrect action model. If this happens, a theory repair mechanism is needed, either specialising an over general theory, or continuing to generalise a theory that is too specific. This can be implemented in much the same way as in Marvin [Sammut81].

1.3.2 Tool Creation

Tool use assumes that a suitable tool is available for a given task. What happens if no such tool exists? Fortunately, we now have 3D printers that give a robot the means to create a new tool. This is dones as follows:

1. An existing tool model is generalised.
2. The new qualitative model is converted to numerical model need to send to a 3D printer.
3. The manufactured tool is used in an experiment, providing another example for learning.

Tool Generaliser. To aid innovation in tool design, it is useful to have an ontology of tools. In our case, these are simple tools like hooks, levers and wedges. The main function of this ontology is to limit the search space when a generalisation is performed. The ontology can be used to suggest generalisations by climbing the hierarchy. For example, if the current model requires a hook to be on the right-hand side of the tool, a generalisation may be that the hook can be on either side. In this case, the new hypothesis is tested by generating a new instance of the hypothesis that does not match previous instances seen or constructed. The generalsation mechanism is based on Marvin (Sammut, 1981):

1. Find a tool that is closest to the requirement for the given task, using the scoring mechanism in Section 1.3.1.
2. Generate a new instance of the hypothesis which is different from any other positive example
3. Test if this new object belongs to the target concept by performing an experime.nt
   - If the new object belongs, the generalisation is accepted.
   - If it does not, perform specialisation by conjoining two generalisations of a trial concept, creating a new instance and carrying out another experiment until it is successful
4. Store the learned concept
5. Return to the first step until there are no more possible generalisations

Generalisation is done is done by a replacement operation, similar to absorption (Muggleton and Buntine, 1988). We use background knowledge to deduce additional relations from the example predicates. The deduced literals are added to the example predicates, while the example literals that implied the deduced literal are removed, this creating a more general concept.

In our case, this test example is manufactured as a new tool in a simulated or real world. The new tool is tested in a learning experiment. We use a similar method as in previous tool-use learning. Wicaksono (2020) describes the emprical evaluation using a real robot (the Rethink Robotics Baxter) on tasks requiring the simple tools metioned earlier: hooks, wedges and levers.

1.3.3 Learning to Plan with Qualitative Models

In the previous section, we described a learning system for a robot in which we assumes the world is reasonably deterministic and high-level actions are discrete. In scenarios such as tool use ro service robots in the home or office, these assumptions are justifiable. However, that is not the case in other domains. Consider a robot designed for urban search and rescue operations. These are generally tracked vehicles with reconfigureable subtracks or flippers. Driving such a vehicle over rough terrain requires considerable judgement, because the operator must make decisions about the configuration of the flippers, as well as steering and speed. Thus, subtracks give the robot greater terrain traversal capabilities at the expense of greater control complexity. Since the interactions of the robot with the terrain in a disaster site are extremely difficulty to predict, we use a learning system to build a model of how control actions,

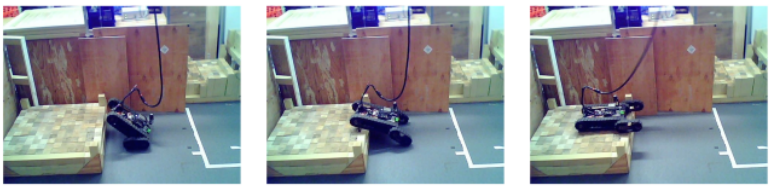
including changing flipper angles, affect the robot's state. Once we have the model, the driving system can plan a sequence of actions to achieve the desired goal state.

An important requirement is that the learning system must be able to acquire the model in a small number of trials. A naive application of reinforcement learning (Sutton and Barto, 1998), which is commonly used for such tasks, may need thousands of trials, which would be very slow and eventually break the robot. Therefore, a more economical approach is required. As in learning action models, we use a symbolic learning system to acquire a high-level model of actions and use this to constrain a reinforcement learning system.

To handle a continuous domain, we switch from a STRIPS-like representation of actions to a qualitative model, based on Kuipers (1986) QSIM. The qualitative model describes each action at an abstract level but does not specify exact numerical values for any parameters. This two stage learning process, acquiring an approximate abstract model, followed by reinforcement learning to refine parameters, greatly reduces the overall search space, and therefore reduces the number of trials required to learn a new skill. However, it is possible that the learned qualitative model is incorrect, in the sense that it does not provide the constraints needed to learn an operational behaviour. In this case, system acquires more training data to refine the qualitative model. Thus, it is a *closed-loop* learning system that can continually improve its behaviour.

The experimental platform we use is an iRobot Negotiator, shown in Figure 1.8 climbing a step. In this case, the step is too high for the robot to climb with the flippers forward, since they are not long enough lift to robot over the step. Instead, the robot reverses up to the step and uses the flippers to raise the body, which is long enough to reach over the edge of the step. The robot's planning system should be able to reason about the geometry of the vehicle and make appropriate decisions about what sequence of actions will achieve it's goal. To do so, the planner must have a model of how actions affect the robot and its environment. The model does not need to be highly accurate for the planner to get the right sequence. An approximate qualitative model will suffice.

The qualitative model extends QSIM so that it can be used for planning. QSIM represents the dynamics of a system by a set of *qualitative differential equations* (QDE). An example of a model for this domain is shown in Figure 1.9. The graph shows the relationship of the angles of the robot's body to the floor, θ_b, and the flipper angle, relative to the body, θ_f. The relation, $M^+(\theta_f, \theta_b)$, indicates that if one of the arguments



(a) (b) (c)

Fig. 1.8 The Negotiator robot reversing up a high step

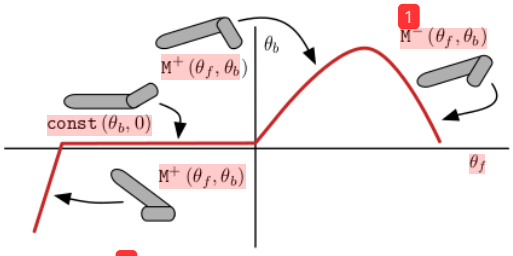


Fig. 1.9 Qualitative Models for the Negotiator Robot, describing the relationship between the angle of the flippers, θ_f, and base, θ_b, through four operating regions.

increases (θ_f), the other also increases (θ_b). The relation, $M^-(\theta_f, \theta_b)$, says the that when one variable increases, the other decreases, that is, they change in opposite directions. The $const(\theta_b, 0)$ relation states that θ_b remains steady at 0.

Each segment in the graph represents an *operating region*, that is, a region of the state space in which one model holds. As the flippers rotate clockwise through 360°, the body is raised and lowered, while the flippers are angled below the body, but have no effect when they are above the body. If the flippers are rotated anti-clockwise, they can raise the body. To accommodate different operating regions, we extend the QSIM notation so that each QDE has an associate *guard*, which is the condition under which that QDE holds.

$$Guard \rightarrow Constraint \tag{1.4}$$

Since a QDE does not specify a numerical relationship between variables, we regard a QDE as a *constraint* on the possible values of the variables. For example, in a particular region, if the flipper angle decreases, the body angle increases. QSIM was originally intended to perform qualitative simulation. That is, given an initial condition, QSIM estimates how the system's state evolves over time. A state in QSIM is represented by the qualitative values of the variables. However, a qualitative variable does not take on a numerical value. Instead it's value is a pair: *magnitude*/*direction*, where the magnitude may be a fixed landmark value or an interval, and the direction is one of increasing, decreasing or steady. For example, the robot's position may be given by $x = 0..x_{step}/inc$, which states that the x position of the robot is between its initial position and the position of the step, and its value is increasing. A set of transition rules specifies how one state evolves into the next. For example, when the robot reaches the step, the position becomes $x = x_{step}/std$. A detailed explanation is given by Wiley (2017).

Planning with Qualitative Models. Kuiper's QSIM has no concept of an action, which is needed for planning. We extend the QSIM representation by distinguishing certain variables as *control variables*, whose values can be changed by the planner. A change in a control variable corresponds to an action. For example, changing the flipper angle, θ_f, corresponds to the motor action that moves the flipper. Like classical planning,

given an initial state and a goal state, the qualitative planner searches for a sequence of actions that leads to the goal state. We briefly describe the planner below, but details of the implementation are given elsewhere (Wiley, Sammut and Bratko, 2014; Wiley, Sammut, Hengst and Bratko, 2016).

Qualitative planning differs from classical planning and numerical simulation because the variables can specify a range of values. Therefore, as in learning action models, a qualitative state can be thought of as defining constraints on regions of valid quantitative states, that is, where the variables take on specific values. For example, a qualitative state may be $x = 0..x_{step}/inc, \theta_b = 0/std, v = 0..max/std, \theta_f = 0..90/std$, which describes the robot driving up to the step. To find a sequence of actions the qualitative planner must propagate the constraints from the initial to the final state. Therefore, planning can be seen as a constraint satisfaction problem. We take advantage of this, translating the planning problem into an Answer Set Programming (ASP) problem (Gebser, Kaminski, Kaufmann and Schaub, 2013) and using the Clingo-4 solver (Gebser, Kaufmann, Kaminski, Ostrowski, Schaub and Schneider, 2011) to generate the plan, described in (Wiley, Sammut and Bratko, 2014).

The search space for the qualitative planner is considerably smaller than the search space for the corresponding continuous domain. This can be seen from the previous observation that one qualitative state covers a region of quantitative states. Thus, qualitative planning is reasonably efficient in finding a sequence of actions. However, these actions are only approximate, in the same sense as the driving instructor's plan for changing gears. In this case of the robot, as it approaches an obstacle, the plan may say that the flippers should be raised, but not by how much. We will see in Section 1.3.3, that "how much" can be found by reinforcement learning, but for this to be efficient, i.e. require only a small number of trials, the planner must pass on its constraints to the reinforcement learning system.

In addition to the sequence of actions, the planner generates the state transitions caused by those actions, giving us a sequence very similar to that shown in Equation 1.2. For each action, we have the preceding and succeeding states. As these are qualitative states, they effectively specify the preconditions and postconditions of the action. Thus, when the reinforcement learning system searches for the angle to set for the flippers, it must only search with the constraints of the pre and post conditions. In the following sections we explain how the qualitative model is learned and then how reinforcement learning is used to find operational parameters for the actions generated by the planner.

Learning a Qualitative Model. To learn a qualitative model of the robot, the system must acquire samples of the robot's interaction with its environment. In the experiments described in Section 1.3.3, a human operator drives the robot, performing random actions. This could equally be done by the robot "playing" by itself. Each time an action is performed, the before and after states are recorded so that the changed effected by the action can be determined. An example of flipper actions is shown in Figure 1.10. The figure also shows qualitative relations induced by Padé (Žabkar, Mozina, Bratko and Demsar, 2011). This systems uses a form of regression, called *tubed-regression*, to find regions of a graph where neighbouring points have the same qualitative relation. In Figure 1.10, Padé has identified regions where the body angle

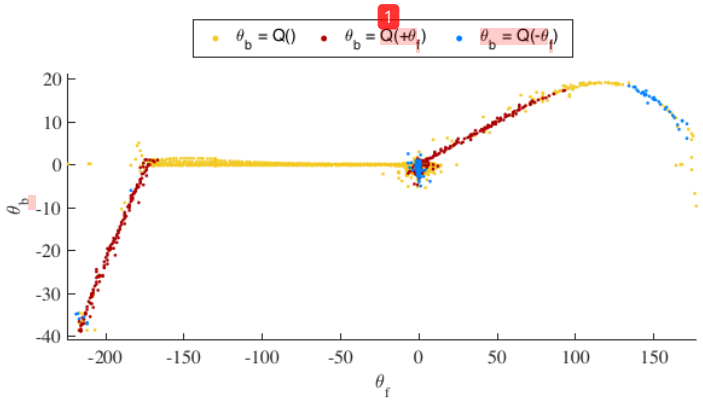


Fig. 1.10 Body angle versus flipper angle from actions

increases with the flipper angle, decreases, or remains steady. In this case, the binary relation between the angles has been rewritten in functional form so that the body angle is dependent on the flipper angle. Note that this plot corresponds to the graph in Figure 1.9.

The data show that there are several operating regions where these relations apply. Recall that we express the qualitative model as a set of rules, whose left hand side specifies the operating region and the right hand side give the qualitative constraints (Equation 1). These rules can be automatically generated from the graph by applying a symbolic rule learning system or decision tree learner. In this case, we use Quinlan's C4.5 (Quinlan, 1993). We make a kind of closed world assumption where the space outside the sampled area is assumed to contain only negative examples. Figure 1.11 shows the decision tree induced from the flipper data. Since the training data are noisy, the decision tree is not as clean as a model that a human might write.

With the qualitative model represented by the decision tree, the planner can determine the qualitative state of the system. QSIM's transition rules tell the planner what possible next states are reachable depending on which action is applied in the current state. With this information, the planner can search for a sequence of actions that will achieve its goal.

Refining Actions by Reinforcement Learning. Like learning action models in Section 1.3.1, the actions generated by the planner are qualitative. For example, to climb a low step, the plan may indicate that the velocity should be forward, non-zero and the flipper angle should be between 0 and 90°. To find values of velocity and flipper angle that actually work, the robot performs trial-and-error learning. Figure 1.12 illustrates this process. For each action generated by the plan, the robot has many options for executing that action (e.g. selecting an angle between 0 and 90°). Through trial-and-error learning, it must discover the parameter settings that will result in the

```
theta_b<=-5.072
|   theta_f<=-178.415: Q(+theta_f)
|   theta_f>-178.415: Q(null)
theta_b>-5.072
|   theta_b<=1.628
|   |   theta_f>17.279: Q(null)
|   |   theta_f<=17.279
|   |   |   theta_b<=-2.772: Q(+theta_f)
|   |   |   theta_b>-2.772: Q()
|   theta_b>1.628
|   |   theta_f<=133.524
|   |   |   theta_b>18.704: Q()
|   |   |   theta_b<=18.704
|   |   |   |   theta_f<=-10.273: Q(null)
|   |   |   |   theta_f>-10.273
|   |   |   |   |   theta_f<=98.500: Q(+theta_f)
|   |   |   |   |   theta_f>98.500: Q(null)
|   |   theta_f>133.524
|   |   |   theta_f<=154.558: Q(-theta_f)
|   |   |   theta_f>154.558: Q()
```

Fig. 1.11 The C4.5 decision tree

robot achieving its goal.

Selecting actions for the robot is a Semi-Markov Decision Process (SMDP) over Options (Sutton, Precup and Singh, 1999). In a SMDP, time is continuous and actions have a duration, which may be of variable length. The robot tries to select a set of options, that is, numerical control value settings, that will allow the robot to succeed in its task. The unique part of this process is how the SMDP is constructed from a plan. Since a qualitative variable specifies an interval, numerical values are sampled, constrained by the interval ranges. For each action, the qualitative states satisfying the pre-condition and post-condition are required. Options are formed from every combination of sampled numerical states that within the bounds of the qualitative

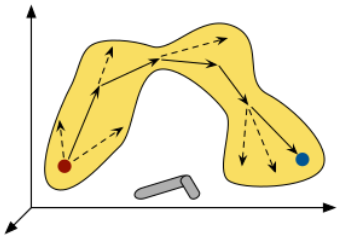


Fig. 1.12 The planner gives an approximation of actions, (yellow region) that must be refined by trial and error learning into precise motor actions (arrows).

states. QSIM constrains a variable's rate-of-change, as well as its magnitude. If the qualitative state satisfying the post-condition has an increasing rate-of-change, its quantitative value must increase during the option, likewise for decreasing or steady rates-of-change. Only options that satisfy the rates-of-change constraints are added to the SMDP.

Once the system has found the available options, it can perform its trial and error learning to turn the planning actions into operational motor commands. However, the quality of the plan depends on the quality of the model that was induced from the original training data, which was randomly sampled in the beginning. If the sampling does not yield sufficient training data in the regions of the state space relevant to the task, then the model may be poor, resulting in inefficient plans. This problem can be reduced by closed-loop learning.

Closed-Loop Learning and Experiments. The learning process begins with the collection of training examples for learning the qualitative model. In these experiments, the data were generated by a human operator instructing the robot to perform mostly random actions. Experiments were performed for several tasks using the Negotiator robot (Wiley, 2017; Wiley, Sammut, Hengst and Bratko, 2016). These included driving over different height steps and climbing a staircase. Of these, climbing a high step, which requires the reversing manoeuvre, was the most difficult.

In closed-loop learning, the system can repeat the entire learning process to improve its performance. Over three repetitions, the average number of trials needed to learn to climb the step was 104 and the average amount of time needed to complete the task is 32.7 seconds. The actions and their effects are recorded so that they can be added to the training examples for the model learner. After the data from the first pass of learning to climb a high step are added, over five repetitions, the average number of trials needed to learn to climb the step was reduced to 31 and the average amount of time needed to complete the task is 25.1 seconds, indicating that a better plan was produced. The systems performs better because the training data for closed-loop learning contains more training examples in areas that the human operator failed to properly sample.

1.4 Conclusion

The work described here demonstrates the usefulness of combining logical representations, reasoning and learning with numerical methods. While the latter have proved essential in enabling a robot to perform in the real world, symbolic methods are better suited to generalising over large classes of objects and behaviours. They provide more powerful planning and problem solving mechanisms and they are more easily explained than sub-symbolic methods.

Acknowledgements

We have benefited greatly from collaborations with our colleagues: Ivan Bratko, Keith Clark, Dianhuan Lin, Stephen Muggleton, Max Ostrowski, Torsten Schaub and Jure Žabkar. This work was supported byThe Australian Research Council.

References

Apt, Krzysztof R, Wallace, Mark et al. (2007). *Constraint logic programming using ECLiPSe*. Cambridge University Press New York.

Bengio, Yoshua, LeCun, Yann, and Hinton, Geoffrey (2015). Deep learning. *Nature*, **521**(7553), 436–444.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, **RA-2**(1), 14–23.

Brown, Solly (2009). *A relational approach to tool-use learning in robots*. Ph.D. thesis, School of Computer Science and Engineering, UNSW Australia.

Brown, Solly and Sammut, Claude (2013). A Relational Approach to Tool-use Learning in Robots. In *Inductive Logic Programming* (ed. F. Riguzzi and F. Železný), pp. 1–15. Springer Berlin Heidelberg.

Dechter, Rina (1986). Learning while searching in constraint-satisfaction problems. In *AAAI-86*, pp. 178–183.

Farid, R and Sammut, C (2014, Jan). Plane-based object categorisation using relational learning. *Machine Learning*, **94**(1), 3–23.

Fikes, R. and Nilsson, N. (1971). Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, **2**, 189–208.

Gebser, Martin, Kaminski, Roland, Kaufmann, Benjamin, and Schaub, Torsten (2013). *Answer set solving in practice*. Synthesis Lectures of Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, Martin, Kaufmann, Benjamin, Kaminski, Roland, Ostrowski, Max, Schaub, Torsten, and Schneider, Marius (2011). Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, **24**(2), 107–124.

Kuipers, Benjamin (1986). Qualitative simulation. *Artificial Intelligence*, **29**, 289–338.

Mao, Jiayuan, Gana, Chuang, Kohli, Pushmeet, Tenenbaum, Joshua B., and Wu, Jiajun (2019). The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *International Conference on Learning Representations*.

Mitchell, Tom M (1977). Version spaces: A candidate elimination approach to rule learning. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*, pp. 305–310. Morgan Kaufmann Publishers Inc.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. (1986). Explanation-based generalisation: A unifying view. *Machine Learning*, **1**.

Muggleton, S. (1991). Inductive logic programming. *New Generation Computing*, **8**, 295–318.

Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Machine. Learning*

Conference (ed. R. S. Michalski, T. M. Mitchell, and J. G. Carbonell), pp. 339–352. Morgan Kaufmann,, Ann Arbor, Michigan.

Muggleton, Stephen and Feng, Cao (1992). Efficient induction in logic programs. In *Inductive Logic Programming* (ed. S. Muggleton), pp. 281–298. Academic Press.

Muggleton, Stephen H. and Lin, Dianhuan (2013). Meta-interpretive learning of higher-order dyadic datalog: Predicate invention revisited. In *Proceedings of the 23rd International Joint Conference Artificial Intelligence (IJCAI 2013)*, pp. 1551–1557.

Nilsson, Nils J. (2001). Teleo-reactive programs and the triple-tower architecture. *Electronic Transactions on Artificial Intelligence*, **5**(B), 99–110.

Plotkin, GD (1971). A further note on inductive generalization. In *Machine Intelligence 6* (ed. B. Meltzer and D. Michie), pp. 101–124. Elsevier.

Plotkin, Gordon D (1970). A note on inductive generalization. In *Machine Intelligence 5* (ed. B. Meltzer and D. Michie), pp. 153–163. Edinburgh University Press.

Quinlan, J R (1993). *C4.5: Programs for machine learning*. Morgan Kaufmann, San Manteo, CA.

Sammut, Claude and Yik, Tak Fai (2010). Multistrategy learning for robot behaviours. In *Advances in Machine Learning I* (ed. J. Koronacki, Z. W. Ras, S. T. Wierzchon, and J. Kacprzyk), Volume 262, Studies in Computational Intelligence, pp. 457–476. Springer.

Sammut, C. A. (1981). *Learning Concepts by Performing Experiments*. Ph.d. thesis, Department of Computer Science, University of New South Wales.

Sammut, C. A. and Banerji, R. B. (1986). Learning concepts by asking questions. In *Machine Learning: An Artificial Intelligence Approach, Vol 2* (ed. R. S. Michalski, J. Carbonell, and T. Mitchell), pp. 167–192. Morgan Kaufmann, Los Altos, California.

Shanahan, Murray (2002). A logical account of perception incorporating feedback and expectation. In *Proceedings of 8th International Conference on Principles of Knowledge Representation and Reasoning*, Toulouse, France, pp. 3–13. Morgan Kaufmann.

Srinivasan, Ashwin (2002). The ALEPH Manual (Version 4 and above). Technical report, University of Oxford.

Sutton, Richard S and Barto, Andrew G (1998). *Reinforcement learning: An introduction* (1st edn). MIT Press, Cambridge, MA.

Sutton, Richard S, Precup, Doina, and Singh, Satinder P (1999, December). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, **112**(1-2), 181–211.

Triska, Markus (2012). The finite domain constraint solver of swi-prolog. In *International Symposium on Functional and Logic Programming*, pp. 307–316. Springer.

Wicaksono, Handy (2020). *A Relational Approach to Tool Creation by a Robot*. Phd thesis, School of Computer Science and Engineering, The University of New South Wales.

Wicaksono, H. and Sammut, C. (2018). Tool use learning for a real robot. *International Journal of Electrical and Computer Engineering (IJECE)*, **8**(2), 1230–1237.

Wiley, Timothy (2017). *A Planning and Learning Hierarchy for the Online Acquisition of Robot Behaviours*. Ph.D. thesis, School of Computer Science and Engineering,

University of New South Wales.

Wiley, Timothy, Sammut, Claude, and Bratko, Ivan (2014, August). Qualitative simulation with answer set programming. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence* (ed. T. Schaub, G. Friedrich, and B. O'Sullivan), Prague, Czech Republic, pp. 915–920. IOS Press.

Wiley, Timothy, Sammut, Claude, Hengst, Bernhard, and Bratko, Ivan (2016). A Planning and Learning Hierarchy using Qualitative Reasoning for the On-Line Acquisition of Robotic Behaviors. *Advances in Cognitive Systems*, **4**, 93–112.

Yik, Tak Fai (2007). *Locomotion of Bipedal Humanoid Robots: Planning and Learning to Walk*. Ph.D. thesis, School of Computer Science and Engineering, The University of New South Wales.

Žabkar, Jure, Mozina, Martin, Bratko, Ivan, and Demsar, Janez (2011). Learning qualitative models from numerical data. *Artificial Intelligence*, **175**(9-10), 1604–1619.

Paper Handy

ORIGINALITY REPORT

9%	8%	2%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	www.cogsys.org Internet Source	2%
2	www.araa.asn.au Internet Source	2%
3	mi20-hlc.doc.ic.ac.uk Internet Source	2%
4	Lecture Notes in Computer Science, 2014. Publication	1%
5	www.intelligentrobots.org Internet Source	1%
6	oxford.universitypressscholarship.com Internet Source	1%
7	"Region-Growing Planar Segmentation for Robot Action Planning", Lecture Notes in Computer Science, 2015. Publication	1%

Exclude quotes On

Exclude matches < 1%

Exclude bibliography On