Control Task-Hierarchies with Concurrent Abstract Actions

Bernhard Hengst,
School of Computer Science and Engineering
University of New South Wales, Sydney, Australia

***Abstract*— This paper describes a modular cognitive robotic architecture for the real-time control of teams of robots in hostile environments. The architecture extends approaches to hierarchical reinforcement learning, to include concurrent abstract actions in task-hierarchies, decision-tree function approximation, and polling execution to generate robust control actions. The architecture was successfully demonstrated at an international level in a real robotic setting – playing soccer at RoboCup with a team of autonomous bipedal robots.**

I. Introduction

This paper describes a robust control architecture for real-time autonomous robotic systems operating in partially observable, stochastic, and hostile environments. Examples of such systems include urban search and rescue robots negotiating unstructured terrain, and teams of robots playing soccer.

The paper makes the following contributions to the field of intelligent robots and systems:

- A formalised modular cognitive robotic architecture that operates at multiple levels of abstraction. It supports both hard-coding and learning of the control policy.
- A task-hierarchy that includes concurrent abstract actions. We extend previous concepts that use unitary and serial abstract actions. Concurrent actions will be shown to generalise the execution stack graph over the hierarchy from a path to a tree.
- Function approximation and polling methods that generate robust behaviour despite unpredictable environmental transitions. The extended architecture is successfully deployed on real robots and environments.

Brachman [1] and others expressed concern that AI's shared vision of creating intelligent machines may be threatened by the fragmentation of the field into a myriad of speciality areas. In response he called for the critical need to raise awareness of the value and challenge of architecture.

Early *sense-plan-act* robot control procedures exemplified by *Shakey The Robot* [2] have evolved to hierarchical multi-module architectures such as the *Real-time Control Systems (RCS)* developed at the US National Institute of Standards and Technology (NIST) [3].

In the rest of this paper we start with the description of our control architecture that is formalised in terms of a task-hierarchy of semi-Markov decision problems. We describe the composition of modules in the hierarchy and how they are interlinked.

We extend this architecture with concurrent actions because control engineers need systems that can launch multiple simultaneous temporally extended actions at several levels in the task-hierarchy. This need arises, for example, when controlling multiple robots, or separately controlling articulated parts of the same robot.

We show how concurrency and termination issues are resolved in unpredictable environments using a polling procedure. Polling also solves the termination issue when executing concurrent subtasks. We describe the use of decision-tree function approximation to partition the high-dimensional state space of the world-model to determine which modules and module states should be active in the task-hierarchy.

We report on an actual implementation of this architecture for a team of small humanoid robots tasked to play soccer. In this implementation the policies are hard-coded and not learnt. There are up to 9 levels of control in the hierarchy, from the game controller managing the two 10 minute half games, right down to the controllers for individual motors in each of the 21 degree-of-freedom robots. We conclude with related and future work.

II. Formalising the Architecture

An architecture specifies those aspects of a cognitive agent that are the same over time and over different applications. Our architecture consists of control-modules. We first describe the general makeup of our modules and then how they interconnect. The exact number of modules, their function and interconnection will vary depending on the environment, the goals, the specific robots, their sensors, and effectors. Later we will see how the architecture implements specific control of robots for the RoboCup Standard Platform League soccer competition.

A. Modules

The architecture is described by a type of task-graph [4] that consists of a set of modules linked in a hierarchy - a *task-hierarchy*. Task-hierarchies are directed acyclic graphs where parent control modules invoke child modules as temporally extended actions.

A module in the task-hierarchy is a mini-agent that accepts input and takes actions based on its policy. It also produces output that abstracts its state for input to higher-level modules. Generally higher-level modules describe more abstract concepts and act on longer time-scales.

A generic module is depicted in Figure 1. It is a tuple $M\langle S, A, T, I, O, R, Q, \pi, B, P, E\rangle$ where:

S is a set of states. A module can only be in one state at a time. A state may describe a parameterised probability distribution, admitting the modelling of

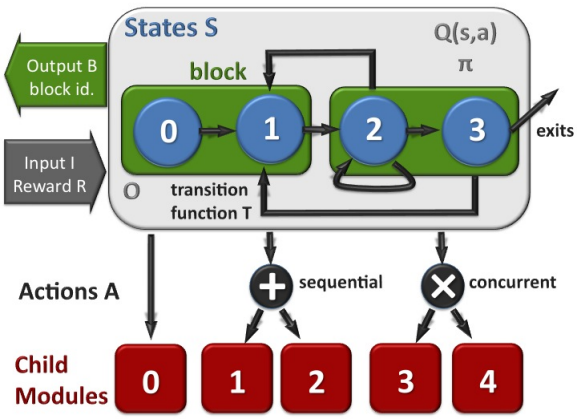

Fig. 1. A Generic Module.

uncertainity. States are usually multi-dimensional or structured eg $s = (s_0, s_1, \ldots, s_n) \in S$

A is a set of actions that the module can take. Concurrent actions are multi-dimensional, $a = (a_0, a_1, \ldots, a_m) \in A$

T is a stochastic transition function. $T : S \times A \times S \rightarrow [0, 1]$. T maps a state given an action to a probability distribution over S

I is a set of input states. They represent the sensed information for the module

O is an observation function. $O : distributionS \times I \rightarrow S$. The observation reduces the uncertainty over S to a point distribution.

R is an expected reward or utility function giving the expected real reward having taken the last action in the last state. $R : S \times A \rightarrow \mathcal{R}$.

Q is an action-value function over states and actions. $Q^\pi : S \times A \rightarrow \mathcal{R}$ is the expected reward for taking an action plus the expected sum of rewards following policy π thereafter.

π is a policy, mapping states to a distribution over actions. $\pi : S \times A \rightarrow [0, 1]$. When the action is deterministic, we overload π, i.e.. $\pi(s) = a$.

B is a set of output states.

P partitions S into blocks B. $P : S \rightarrow B$. Blocks represent more abstract concepts (e.g. rooms within a house). They provide inputs to parent-level modules allowing them to compose abstract states.

E is a set of module exit conditions. They terminate execution of the module and return control to the parent module.

Not all the elements of the tuple need to be specified in all cases. For example, if the policy π is provided as background knowledge, the reward function R and action-value function Q may not be required. *Perception* modules, such as object detectors or localisers, have input and output variables but no actions.

In general the actions A may be state dependent and not all available in every state. Actions are *primitive* when they act directly on the environment or *abstract* when they invoke (start executing) child modules. Abstract actions are usually *temporally extended*, meaning that they take multiple time-steps to terminate.

Actions are of three types: *unitary*, *sequential*, and *concurrent*. A unitary action is one that invokes the execution of one child module. A sequential action invokes multiple modules, but only executes one at a time in an arbitrary order (e.g. putting on a left and right sock). Sequential actions generate Partial-Order task-hierarchies [5]. Concurrent actions are similar to sequential actions except that the child modules are executed simultaneously instead of sequentially. We use the symbol $\oplus$ to designate sequential actions in the task-hierarchy and $\otimes$ for concurrent actions. A concurrent action invokes and executes two or more child modules at the same time (e.g. moving two fingers to grasp an object).

For unitary actions, exit conditions E are specified as state-action (s, a) pairs [6]; for sequential actions as tuples $(s_1, \ldots, s_k, a)$ [5]; and for concurrent actions as tuples $(s_1, a_1, \ldots, s_k, a_k)$. Subscripts for states s and actions a designate one of the k sequentially or concurrently activated child modules.

Without an explicit input I and observation function O, modules are similar to Markov *options* [7]. A module is a semi-Markov decision process (SMDP) [8]. Defining modules in this way allows us to inherit all available reinforcement learning methods (see [9] for an introduction). The effect of input I and observation function O is to make the combined operation of transition function T and function O a fully observable deterministic update.

This does not preclude modelling stochastic processes. We can model a Kalman filter by modelling Gaussian probability distributions with mean and variance state variables and state transitions T as the usual closed-form updates. Particle filters, representing general distributions, are defined by the state of a finite set of particles avoiding the need to make states in S probability distributions. In general the states I and S can represent probability distributions over histories of inputs and actions and the update function can be a general Bayesian filter.

In summary, a module most generally implements a Markov decision process, that together with the sensor model, realises an underlying deterministic state machine able to represent stochastic events and beliefs.

By way of an example, consider the top-level module used in RoboCup soccer by the Standard Platform League. This module is called the *game controller* and is shown at a coarse level of detail in Figure 2. The states represent the state-of-play. *Ready* means the robots on both teams are in the process of positioning themselves ready for kick-off, during *Set* the robots are not allowed to move (other then their heads) and may be repositioned manually by the referees, *Playing* is when they are engaged in 10 minute halves of soccer, etc. The primary input comes from a human operator changing the game-state manually. For example when the referee determines that a goal has been kicked, the state reverts to *Ready*. The actions of the game-controller control all the robots wirelessly. Figure 2 shows an example of a

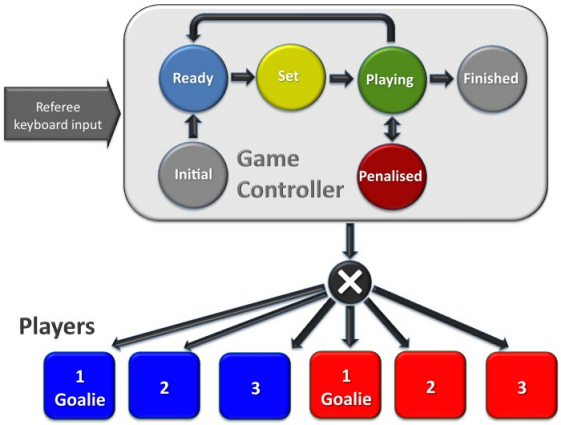

Fig. 2. Game-controller module.

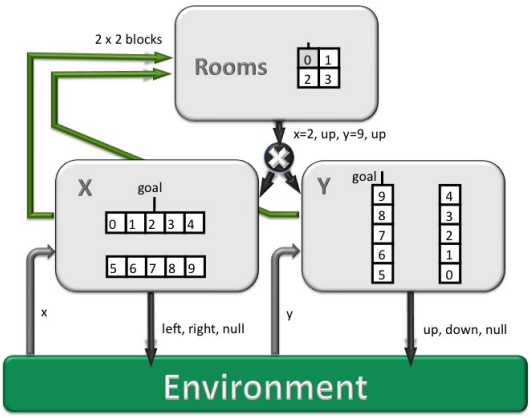

Fig. 3. Module Interconnections.

concurrent action invoking six robots simultaneously – three per side. The effect of this concurrent action is to launch separate processes in each of the six autonomous robots.

B. Modules Interconnect to form Task-Hierarchies

Modules are linked in an acyclic task-hierarchy in two ways:

1) **Actions** Parent modules invoke child modules as abstract actions. Abstract actions are temporally extended, meaning that the duration of the action can persist over several time-steps. Child modules implement actions from their parent module as goals that they aim to achieve. Goals are specified in terms of child-module exit termination conditions, or simply *exits* described earlier. Typically the child-module executes a policy with the objective of transitioning to the exit-state, executing the exit action, and terminating.
2) **Output-Input** Child modules communicate their state partition block identifiers to their parents. This has the effect of abstracting the state-space at the parent-level to allow higher level control. When multiple child modules are invoked by a parent, the parent receives the Cartesian product of all the child module block identifiers as input ($I_{parent} = B_{child-0} \times B_{child-1} \times \ldots \times B_{child_n}$)

As an example, Figure 3 shows one parent module linked by concurrent-actions to two child modules. This controller could represent a robot moving around a home with 4 rooms (Figure 4) using a GPS-like (x, y) position sensor. The x and y dimensions are partitioned into two blocks each, to allow reliable abstract actions to be learnt to move the agent in both x and y directions inside rooms. The Cartesian product of the two block identifiers generate four abstract states in the parent module and represent the four rooms in the house.

In operation, the child modules perceive their respective x and y location values from the sensor. They communicate the block values that contain the x and y states to the parent module that in turn determines the room occupied by the robot (say the upper-lefthand room - state 0 - shown shaded in Figure 3). The parent module now follows its policy and

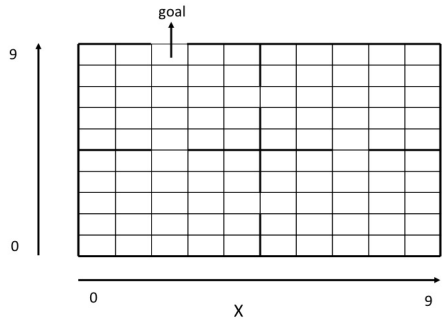

Fig. 4. Four room maze.

takes concurrent action $(x = 2, null, y = 9, up)$ to guide the robot out of the house. The effect of this abstract action is to invoke both child modules. The X module's policy moves the robot left or right until state $x = 2$. The Y module moves the robot up or down to position $y = 9$. Finally the action $(null, up)$ is executed terminating the concurrent action with the robot having achieved its goal.

C. Concurrent Action Termination and Rewards

The above example illustrates the *all* termination scheme that returns control to the parent module ensuring that all termination conditions are executed at the same time. In this scheme some child modules need to execute *null* actions while waiting for the others to complete. Alternative termination schemes are *continue* - letting child modules run while terminating ones are re-invoked, and *any* - terminating every child modules when any one of one of them terminates [10]. In the context of a "polling" execution procedure, to be described later, execution is interrupted and the overall situation re-evaluated at every time-step.

The treatment of rewards for concurrent abstract actions are of two types depending on whether each concurrently executing module receives its own reward or a common reward. When rewards are common (e.g. related to elapsed time) we learn each concurrently executing module using the same common reward. In this case the value function for the

concurrent action is the maximum value to termination of all the concurrently executing modules. When rewards are independently supplied to each concurrently executing child module (e.g. energy usage) we add the value-functions from the multiple child modules. The Q value-function can now be decomposed over a task-hierarchy with concurrent actions and state abstraction opportunities exploited following a similar approach to MAXQ [4].

D. Task-Hierarchy Execution

The execution of a task-hierarchy starts with modules that sense the environment. Information cascades up the hierarchy via the output(B)-input(I) links. Once the root-level module is reached, it is invoked and recursively activates modules down the hierarchy based on the policy of the parent modules. At any time-step unitary and sequential actions lead to a single execution path through the task-hierarchy. With the introduction of concurrent modules, a tree of modules is invoked and executed as shown in Figure 5. The action types can be mixed with the invocation and execution trace down the task-hierarchy forked by concurrent actions. This *execution-tree* can change rapidly from time to time if the the state of the modules changes.

While the task-hierarchy is acyclic, enabling a compact representation of reusable modules, the invocation of modules generates distinct instantiations. For example, even though walking modules may be the same between robots, each robot runs its own version in the combined execution-tree.

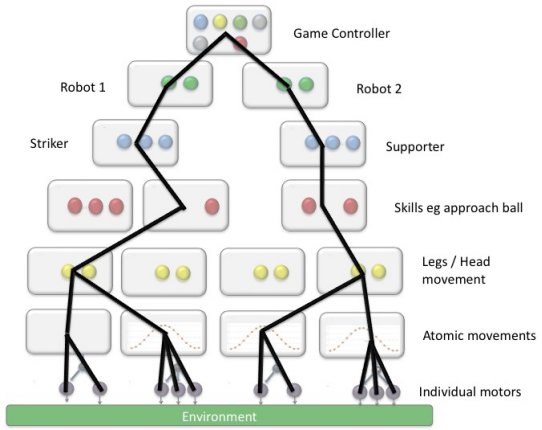

Fig. 5. The execution stack of active modules form a tree in a task-hierarchy. Concurrent actions cause branching indicating parallel execution. Perception modules that identify objects such as the ball and goals, and ones the localise the robot, are not shown.

E. Module Function Approximation

For environments with high-dimensional state descriptions and possibly continuous state dimensions, we use function approximation to determine the states and policy. An example function approximator is a decision-tree that provides the state of each invoked module. The state of a module may depend on a combination of the input I, the observation function O, the previous state $s \in S$ and the termination of the current (abstract) action $a \in A$. These variables are partitioned by the decision-tree into module states. The states are associated with actions. The decision-tree leaves are the module actions. The decision-tree acts as a policy function approximator, aggregating many variable values that have the same action.

Using robot soccer as an example, once the ball has been located the two non-goalie players on each team decide whether their role is to be a *striker* - approaching and kicking the ball, or a *supporter* - covering another part of the field. The module that makes this role decision uses the following variables: the last role of the robot ($last\_role$), the distance to the ball ($ball\_dist$), and the other player's distance to the ball ($other\_dist$). The decision-tree for the role module is show as Algorithm 1. T_1 and T_2 are two positive threshold values that create a hysteresis effect to avoid role vacillation. The resultant policy favours the *striker* role to ensure aggressive play.

Algorithm 1 Role Module Policy Function Approximation

```
1:  if role = none then
2:      action = striker
3:  else
4:      if last_role = striker then
5:          if dist_ball > other_dist + T1 then
6:              action = supporter
7:          else
8:              action = striker
9:          end if
10:     else
11:         if dist_ball < other_dist + T2 then
12:             action = striker
13:         else
14:             action = supporter
15:         end if
16:     end if
17: end if
```

F. Polling Execution

The performance of a controller is usually measured by how well it maximises a function of future reward, for example the sum of rewards to termination, or the infinite series of future discounted rewards. When the controller is modularised and executes a task-hierarchy, optimal control is called *hierarchically optimal* when it maximises the function of future reward within the constraints of the hierarchy. It is *recursively optimal* when modules maximise rewards in a context free way, without taking into account the state of their parents [4]. This is in contrast to a *flat* controller consisting of one large module that is (theoretically) able to produce a globally optimal solution.

For the above hierarchical controllers, modules execute until they reach their termination conditions whereupon they return control to their parent modules. When the environment is stochastic, it is possible, due to stochastic drift to another exit, that a different child module would be more optimal,

but the current one is locked-in and stubbornly persists until its termination [11]. This effect is even more pronounced when the environment is perturbed outside the anticipated stochastic effects modelled by the transition function T.

Dietterich showed that the use of a *polling* execution procedure may improve the hierarchical or recursively optimal policy [4]. Polling interrupts the execution of the hierarchy at each time-step and relevant modules are reinvoked based on the latest state information. We extend polling execution to task-hierarchies using concurrent actions and decision-tree function approximation. In the event of even catastrophic unintended consequences, such as two robots colliding and falling over, the controller will switch policy to recover the situation.

III. Results

The architecture was implemented on Nao robots for the rUNSWift team in the 2010 RoboCup Standard Platform league. The overall task-graph is summarised in Figure 6.

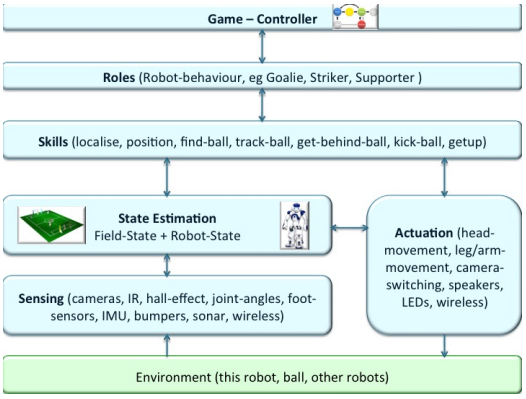

Fig. 6. Summary of the 2010 rUNSWift task-graph.

In Figure 7 we illustrate the approach by describing one control branch of the execution tree from the root module down to primitive actions.

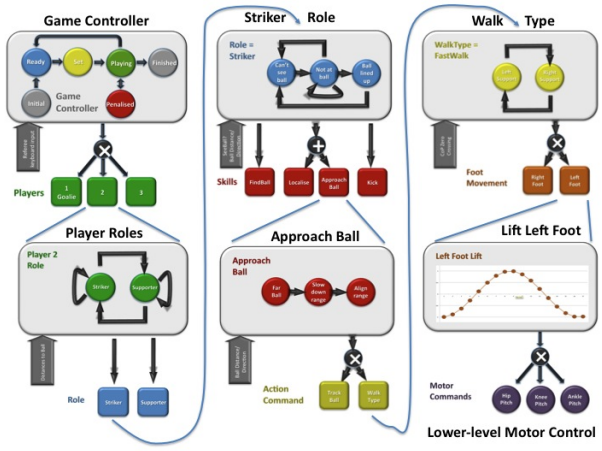

Fig. 7. A branch of the tree through the task-hierarchy.

At the root level, the *game-controller* module uses concurrent actions to simultaneously control all player robots as described previously. Two of our team players have *role* modules that switch the role of each player between *striker* and *supporter* as described by Algorithm 1. Assuming the striker role module is selected, the states of the striker transition from searching for the ball, to approaching it from the right direction, to finally kicking the ball towards the goal. A sequential action is available that requires the robot to both localise and approach the ball. The robot cannot perform both actions concurrently because active localisation requires it to take its "eye" off the ball to look for landmarks. The order of these actions is not important, only that the repetitive application of this sequential action ensures that the robot remains localised as it approaches the ball. The approach-ball module has concurrent actions available to it that move the head and body simultaneously. The head contains the camera and is tasked to track the ball. The body, including the arms and legs, control walk-types generating omni-directional bipedal locomotion. The walking action is parameterised with *forward*, *left*, *turn*, and *power* variables that are set depending on state, allowing the robot to adjusts its direction to walk directly to the ball, slow down without falling when closer to the ball, and to align the ball ready for kicking without accidentally bumping into it. The bipedal walk module invoked by the approach-ball walk-type action, alternates the support between the left and right foot. The input to the walk module is feedback from foot-sensors providing a centre-of-pressure measurement that is used to stabilise the walk. Concurrent actions that simultaneously activate both legs are used to generate the detailed movements. An example module for one of the legs moving the foot up and down is the bottom module shown in Figure 7. The states of the module represent the height that the foot is from the ground. They follow a sinusoidal path to smooth out leg component accelerations and soften the foot replacement on the ground. The concurrent actions available for foot-lifting are the primitive actions that move the hip, knee and ankle motors to achieve the desired lifting action.

IV. Related Work

Modular control architectures date back at least to Ashby's meta-stable controllers [12]. Ashby showed how control, or *regulation* as he called it, can be scaled (amplified) using a hierarchic arrangement.

Our work builds on approaches to hierarchical reinforcement learning, that include temporally extended actions, or options [7], value-function decomposition over MAXQ task-hierarchies [4], automatically constructing task hierarchies with HEXQ, including serial (multi-tasking) actions [6], [5]. The hierarchy of abstract machine (HAMQ) approach to specifying task-hierarchies [13], has been generalised to specifying programs (ALISP) [14] and concurrency [15]. To our knowledge learning ALISP program structure is still an open problem. An issue with HRL approaches is scaling to higher-dimension applications involving real robots.

The real-time control architecture (RCS) [3] is intended for real robots, but is not designed to learn the controller or the policy. RCS modules include perception, world-model

(state), behaviour (action policy), and judgement (value-function) components and are linked in a type of task-hierarchy. RCS is a reference level architecture, providing broad functions, but leaving implementation details to each specific application. Another related controller approach for higher-level behaviour is the Extensible Agent Behaviour Specification Language (XABSL) [16] developed in the context of RoboCup. XABSL is not formalised using MDP and HRL theory but does employ decision trees and polling to control higher level behaviour.

Our architecture combines many of the features of other proven approaches and distinguishes itself by generating behaviour at all levels, includes serial and concurrent actions, is suited for real and hostile robotic control environments and has demonstrated structure learning, albeit only in simple environments to date.

V. Discussion

In this paper we have introduced concurrent actions in a MAXQ-like task-hierarchic architecture. It is clear from the foregoing description that there are several areas where concurrent actions are necessary to complete an adequate task-graph description. Concurrent actions are required to simultaneous activate multiple robots, for head and body movements, the control of the legs, and driving several motors.

Application of precursors of this architecture [5] have been restricted to grid-world like applications. We have described its application to a complex real-world robotic application with the complexity of the control of a team of autonomous robots broken down into an interconnected set of standard modules.

The SPL league in RoboCup (Figure 8) differs from other leagues in that the robotic hardware is identical for all competitors and may not be modified, with results reflecting only the suitability of robotic architectures and the performance of software components. One way to evaluate effectiveness is to test the final system in the environment for which it is designed, to see whether it achieves its objectives, and to compare it to competitors.

Fig. 8. rUNSWift team of Nao robots playing soccer in the 2010 RoboCup SPL league in Singapore.

rUNSWift was placed first in the Challenge competition and was a finalist in the SPL soccer competition in RoboCup 2010, Singapore. The top four teams were the University of Bremen (B-Human), the University of New South Wales (rUNSWift), the University of Texas at Austin (Austin Villa), and Carnegie Mellon University (CMurfs) [17].

The concurrent action architecture was applied to rUNSWift 2010 by hand-coding both the hierarchical module structure and the module policies. As the architecture is formalised in terms of hierarchical reinforcement learning, it opens the way to machine learn module policies and module structure on real robots as the main thrust of our future work.

Acknowledgements

The support of the ARC Centre of Excellence for Autonomous Systems is gratefully acknowledged, as is the contribution of the rUNSWift 2010 team members: Jayen Ashar, David Claridge, Brad Hall, Hung Nguyen, Adrian Ratter, Stuart Robinson, Benjamin Vance, Brock White, Yanjin Zhu, Maurice Pagnucco, and Claude Sammut.

References

[1] R. J. Brachman, "More than the sum of its parts," *AI Magzine*, vol. 27, no. 4, pp. 19–34, Winter 2006.

[2] N. J. Nilsson, "Shakey the robot," AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, Tech. Rep. 323, Apr 1984.

[3] J. S. Albus, *Engineering of Mind: An Introduction to the Science of Intelligent Systems*, 2001. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0471438545.html

[4] T. G. Dietterich, "Hierarchical reinforcement learning with the MAXQ value function decomposition," *Journal of Artificial Intelligence Research*, vol. 13, pp. 227–303, 2000.

[5] B. Hengst, "Partial order hierarchical reinforcement learning," in *Australasian Conference on Artificial Intelligence*, 2008, pp. 138–149.

[6] ——, "Discovering hierarchy in reinforcement learning with HEXQ," in *Proceedings of the Nineteenth International Conference on Machine Learning*, C. Sammut and A. Hoffmann, Eds. Morgan-Kaufman, 2002, pp. 243–250.

[7] R. S. Sutton, D. Precup, and S. P. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999. [Online]. Available: citeseer.nj.nec.com/sutton99between.html

[8] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York, NY: John Whiley & Sons, Inc, 1994.

[9] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press, 1998.

[10] K. Rohanimanesh and S. Mahadevan, "Learning to take concurrent actions." in *NIPS*, 2002, pp. 1619–1626.

[11] M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier, "Hierarchical solution of Markov decision processes using macro-actions," in *Fourteenth Annual Conference on Uncertainty in Artificial Intelligence*, 1998, pp. 220–229.

[12] R. Ashby, *Design for a Brain: The Origin of Adaptive Behaviour*. London: Chapman & Hall, 1952.

[13] R. Parr and S. J. Russell, "Reinforcement learning with hierarchies of machines," in *NIPS*, 1997.

[14] D. Andre and S. J. Russell, "Programmable reinforcement learning agents," in *NIPS*, T. K. Leen, T. G. Dietterich, and V. Tresp, Eds. MIT Press, 2000, pp. 1019–1025.

[15] B. Marthi, S. Russell, D. Latham, and C. Guestrin, "Concurrent hierarchical reinforcement learning," in *Proceedings of the 19th international joint conference on Artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005, pp. 779–785. [Online]. Available: http://portal.acm.org/citation.cfm?id=1642293.1642418

[16] M. Lötzsch, J. Bach, H.-D. Burkhard, and M. Jüngel, "Designing agent behavior with the extensible agent behavior specification language XABSL," in *RoboCup 2003: Robot Soccer World Cup VII*, ser. Lecture Notes in Artificial Intelligence, D. Polani, B. Browning, and A. Bonarini, Eds., vol. 3020. Padova, Italy: Springer, 2004, pp. 114–124.

[17] RoboCup SPL Technical Committee, "Robocup standard platform league competition," Website: http://www.tzi.de/spl/bin/view/Website/WebHome, 2010.