Autonomous Robot Controllers Capable of Acquiring Repertoires of Complex Skills

Michael Beetz, Freek Stulp, Alexandra Kirsch, Armin Müller, and Sebastian Buck

Munich University of Technology, Department of Computer Science IX
{beetzm,stulp}@in.tum.de
http://www9.in.tum.de/agilo

Abstract. Due to the complexity and sophistication of the skills needed in real world tasks, the development of autonomous robot controllers requires an ever increasing application of learning techniques. To date, however, learning steps are mainly executed in isolation and only the learned code pieces become part of the controller. This approach has several drawbacks: the learning steps themselves are undocumented and not executable.
In this paper, we extend an existing control language with constructs for specifying control tasks, process models, learning problems, exploration strategies, etc. Using these constructs, the learning problems can be represented explicitly and transparently and, as they are part of the overall program implementation, become executable. With the extended language we rationally reconstruct large parts of the action selection module of the AGILO2001 autonomous soccer robots.

1 Introduction

Programming sophisticated low-level control skills as well as action selection strategies for autonomous robots acting in dynamic and partially observable environments is both tedious and error prone. Autonomous robot soccer, which has become a standard "real-world" test-bed for multi robot control, provides a good case in point. In robot soccer (mid-size league) two teams of four autonomous robots – one goal keeper and three field players – play soccer against each other. The key characteristics of mid-size robot soccer is that all sensing and action selection is done on-board.

Competent soccer play entails, besides other capabilities, the skillful execution of various navigation tasks such as defending, dribbling, moving past opponents, and intercepting the ball. To realize them, robots could use computational means to infer which control signal should be issued to arrive at a desired state, how long it will take to get there, and how promising it is in the current situation to try to arrive at this state.

Because of the dynamical and adversarial nature of a soccer play and physical subtleties such as friction on different surfaces and the weight of the robot, programming procedures for these reasoning tasks is very difficult. An attractive alternative is the development of control systems that can acquire and adapt such procedures automatically. Obviously, such a control system must learn at various levels of abstraction. It must learn process models from sensory data such as the effects of control signals on the dynamical state, and optimize control laws, e.g. for approaching the ball. It must acquire models of control routines including their success rates and time requirements to decide whether or not to go for a ball or defend the goal. Finally, it must learn the situation-specific selection of appropriate actions.

D. Polani et al. (Eds.): RoboCup 2003, LNAI 3020, pp. 592–599, 2004.

In current practice, such learning tasks are typically considered and solved in isolation. A programmer gathers a set of examples for learning or provides an exploration strategy. Then a learning system, a feature language in which the acquired data are to be encoded, and a suitable parameterization of the learning system are selected. Subsequently, the learning step is executed, the learned routine transformed into an executable piece of code, and provided as part of a software library. The controller can then use the learned routine by calling it as a subroutine. This approach has several disadvantages: the learning steps themselves are undocumented and not automatically executable, and therefore difficult to reproduce.

In this paper, we extend an existing control language to automate this process. Using the constructs of this language, the system model can be specified, and the learning problems can be represented explicitly and transparently, both becoming an integrated part of the overall program. Therefore, the learning problems become executable, documentation of models is integrated in the program, modularity is higher, and analysis of the system is simplified.

Several programming languages have been proposed and extended to provide learning capabilities. Thrun [8] has proposed CES, a C++ software library that provides probabilistic inference mechanisms and function approximators. Unlike our approach a main objective of CES is the compact implementation of robot controllers. Programmable Reinforcement Learning Agents [2] is a language that combines reinforcement learning with constructs from programming language such as loops, parameterization, aborts, interrupts, and memory variables. A difference with our method is that learning tasks can also be specified in our language, and that the learning methods are not confined to reinforcement learning only.

In the remainder of this paper we proceed as follows. The next section describes how process models can be specified. In section 3 it is shown how learning problems can be specified, and in section 4 we give a case example: the AGILO2001 controller. We conclude with a detailed outlook on our future research investigations.

2 Modeling the System

In this section we specify models of different components of the robot and its environment, and show how they can be made explicit in our language. The first class of models we present are process models, such as the robot's dynamics and sensing processes. The second class models the robot's control routines.

2.1 Specifying the Environment Process

define controlled process robot field process
  static environment field model
  environment process ball proc., robot proc., team-mate proc. $_{1,2,3}$, opponent proc. $_{1,2,3,4}$
  sensing process $state \rightarrow$ camera-image $\times$ odometry-reading $\times$ $message$

The construct above specifies that the "robot field process" is a controlled process (we use the dynamic system model, described in [6]), consisting of a static field model and environment and sensing process. These two processes can themselves be decomposed into subprocesses. In the model of the AGILO RoboCup control system the environment process consists of the dynamics of the robot, which specifies how the control

inputs change the physical state of the robot. The other subprocesses of the environment process are those that change the physical states of the team mates, the ball, and the opponent robots. All these processes are largely independent of each other. The only interactions between them are collisions and shooting the ball.

By using this construct the model of the environment is an explicit part of the overall program. This ensures that documentation about the model is integrated, and that modularity of different processes is enforced. Not only does this construct model the real environment, it can also be used as an environment simulator specification. To realize this simulator, the process models need to be implemented. A straight-forward method of realizing this is by manually implementing procedures for each process. If these process models suffice for predicting the effects of control signals on the game situation then they constitute an ideal simulator for the system. In section 3 it will be shown that by using learning task constructs, the manual implementation can be replaced by a learning procedure, which can be integrated into the overall program as well.

2.2 Specifying the Control Process

In this section we provide the control system with models of its control routines. These models enable the robot to do more sophisticated action selection and thereby improve its behavior with respect to the given performance measure. We provide this model in two pieces. First, the control task that specifies *what* has to be done and second, the control routine that specifies *how* it has to be done. The rationale behind separating these two aspects is that a task can be accomplished by different routines that have different performance characteristics.

Control Tasks specify *what* the robot should be capable of doing. For example, the robot should be capable of going to the ball, intercepting a ball, and defending. A control task can be specified in two ways: first, we can specify it using a start state and a description of goal states. An example of such a control task is the following one: reach the position of the ball facing the opponent goal, which we represent as $\langle x = x_{ball}, y = y_{ball}, \theta = \theta_{toball} \rangle$. The set of possible control tasks can be constrained as $\{\langle is, gs \rangle | is \in \mathcal{IS} \wedge gs \in \mathcal{GS}\}$. For example, we might constrain this control task to situations where the ball is not in the own penalty area. In addition, we can specify a probability distribution over the possible set of tasks. This probability distribution affects the expected performance of the control routines.

An alternative way of stating a control task is to specify it using the start state and an objective function, a performance measure that the robot should try to maximize. For example, for attacking, an appropriate performance measure might be the expected time needed to shoot a goal, which should be minimized.

control task achieve-dynamic-state
  task specification $\langle x = x_{ball}, y = y_{ball}, \theta = \theta_{toball} \rangle$
  process signature $\langle x, y, \theta \rangle \times \langle x_g, y_g, \theta_g \rangle \rightarrow \langle v_{tra}, v_{rot} \rangle$
  control process specification
    goal $(\langle x_g, y_g, \theta_g \rangle) \wedge$ achievable$(\langle x_g, y_g, \theta_g \rangle)$ $\Rightarrow$ *active* (navigate $(\langle x_g, y_g, \theta_g \rangle)$)
    goal $(\langle x_g, y_g, \theta_g \rangle) \wedge \neg$achievable$(\langle x_g, y_g, \theta_g \rangle)$ $\Rightarrow$ *fails* (navigate $(\langle x_g, y_g, \theta_g \rangle)$)
    active (navigate $(\langle x_g, y_g, \theta_g \rangle)) \wedge$ timed out $\Rightarrow$ *fails* (navigate $(\langle x_g, y_g, \theta_g \rangle)$)
    active (navigate $(\langle x_g, y_g, \theta_g \rangle)) \wedge \langle x, y, \theta \rangle = \langle x_g, y_g, \theta_g \rangle$
      $\Rightarrow$ *succeeds* (navigate $(\langle x_g, y_g, \theta_g \rangle)$)

Let us consider, as an example, the control task specification shown above. The task specification states the desired values for the state variables x, y, and θ. The process signature indicates that the current dynamic state and the desired dynamic state are mapped into control inputs for the translational and rotational velocity of the robot. The control process specification then states when the control task should be activated and the different possible outcomes of performing the control task.

Control Routines specify *how* the robot is to respond to sensory input or changes in the estimated state in order to accomplish a given control task. We might have different routines for a given control task, which have different performance characteristics. Control routines first of all consist of an implementation, a procedure that maps the estimated state into the appropriate control signals. Besides the implementation we have the possibility to specify models for the control routine, encapsulating the procedural routine within a declarative construct which can be used for reasoning and manipulating. Suppose we had a model consisting of decision rules that can identify situations in which the control routine is very likely to succeed and very likely to fail. We could then apply these rules in order to decide whether to activate the routine in the current situation or terminate it if the routine is likely to fail. This is not possible with only a procedural representation of the control routine.

3 Adding Learning Capabilities

In the last section we have specified process models and implemented them as procedures. However, it is often difficult and tedious to program them manually. In this section we will describe how the corresponding learning problems and the mechanisms for solving them can be specified. There are a number of program pieces that can be learned rather than being specified by programmers. In our robot soccer application, the robot learns, for example, the process model for its dynamics. Also, there are many opportunities for learning the implementation as well as the models of control routines.

We will restrict ourselves to learning problems that can be solved by function approximation. In particular, we will look at problems that can be learned by artificial neural networks and by decision tree learning algorithms. Our primary example will be learning how to achieve a given dynamic state, which is representative for learning in the context of control routines.

To state learning problems we must provide several pieces of information. The type of function that is to be learned, whether it is a process model or a control task. We must also specify the robot to be controlled and the environment it is to act in. In addition, the learning problem specification contains a specification of a problem specific learning system, and the data gathering process. This information is provided using the macro *define-learning-problem*.

define learning problem achieve-dynamic-state
  control-routine *achieve-dynamic-state-3*
  control-task achieve ($\langle x = x_{ball}, y = y_{ball}, \theta = \theta_{toball} \rangle$)
  dynamic system agilo-robot-controller
  data-collector achieve-state-data-collector
  learning-system achieve-state-learner

The learning problem specification shown above specifies these information pieces for the task *achieve dynamic state*. In the remainder of this section, we introduce the representational structures for specifying the different components of learning problems and then discuss issues in the implementation of these constructs.

3.1 Specifying Data Collection

The first step in learning is the data collection. To specify data collection we must state a control task or sample distribution and we have to provide monitoring and exception handling methods for controlling data collection. CLIP [1] is a macro extension of LISP, which supports the collection of experimental data. The user can specify where data should be collected, but not how.

In order to acquire samples we have to define a sample distribution from which samples are generated. During data collection the perception, control signals, and control tasks of each cycle are written to a log file. The data records in the log files can then be used for the learning steps.

define task distribution achieve-dynamic-state
  with global vars
    translation-tolerance ← 0.03
    max-time-steps ← 20
  for initial state $\langle x, y, \theta, v_{tra}, v_{rot} \rangle \leftarrow$ ⟨ -4.5, 1.5, 0.3, [0-0.6], [0-0.7] ⟩
    for goal state $\langle v_{tra:g}, v_{rot:g} \rangle \leftarrow$ ⟨ [0-0.6], [0-0.7] ⟩
      with setup *setstate* ⟨ x, y, θ, v_{tra} , v_{rot} ⟩, *setgoal* ⟨ $v_{tra:g}$, $v_{rot:g}$ ⟩

The task distribution specification above defines a distribution for the initial state consisting of the robot's pose and translational and rotational velocity, as well as the goal state of an episode, which consists of the desired translational and rotational velocity. The body of the episode consists of setting the robot in its initial state and then setting the desired dynamic state as a goal for the controlled process.

The data collection cycle starts with an initialization phase in which a task is sampled from the task distribution. In the distribution above this is the *setstate* command in the *with setup* statement. The data collector monitors the initialization to detect whether or not it has been successful. Unsuccessful initialization is particularly likely when the learning task is performed using the real robots. Upon successful initialization the data collection phase is activated. In the data collection phase the perception and control signals are collected. The data collector then detects the end of the episode and decides whether or not the episode is informative and should be recorded.

Thus, in order to collect data reliably and efficiently it is necessary to specify special purpose code for the initialization phase, for failure handling, and resetting the robot. These code pieces can be provided using the *define data collection functions* .

3.2 Specifying Learning Systems

Learning problems can be solved with very different learning systems such as artificial neural networks or decision tree learners. In order to specify a task-specific learning systems three pieces of information must be provided. First, we have to specify how the collected data are to be transformed into patterns that are used as input for the learning system. An example is the transformation of global coordinates to an robot-centric

coordinate system. Second, a function for parameterizing the learning system. For an artificial neural network for instance we have to define the topological structure of the network. Finally, we must provide a function that transforms the output of the learning system into an executable function that can be employed within the robot control program. Thus, the input patterns for the *achieve-dynamic-state* control task can be defined as follows:

define state space achieve-dynamic-state
  input-features (($v_{tra:0}$ (percept:v_{tra} :time t))
    ($v_{tra:g}$ goal-state:v_{tra}))
  output-values (($v_{tra:1}$ (percept:v_{tra} :time $(t+1)$)))
  output-function (v_1 float)

The macro declares the training pattern to consist of the feature $v_{tra:0}$, which is taken to be the v_{tra} of the percept component of a log file entry and $v_{tra:g}$ that is the v_{tra} of the goal state of the log file entry. The output value of the pattern, called $v_{tra:1}$, is the translation velocity of the subsequent entry in the log file. The output of the learned function will be $v_{tra:1}$.

3.3 From Partial Specification to Executable Program

We have implemented our representational structures using LISP's macro facilities and provided them in the context of *Structured Reactive Controllers (SRCs)* [3]. The LISP-based parts are equipped with an abstract robot interface consisting of C libraries loaded as shared objects. The system uses shared memory communication as a coupling to the low-level control processes. To date, we have provided the functionality of two learning systems: the decision tree learning algorithm C4.5 and the Stuttgart Neural Network Simulator (SNNS).

An important implementational aspect is how we get from declarative problem specifications to executable and effective learning processes. For this and other purposes control routines, process models, and control routine models are represented as first class objects that computer programs can reason about and manipulate. One property of these objects is that they can have an implementational status, such as *to-be-learned*. The distinct characteristic of this program is that in the beginning it only partially specifies the behaviour, leaving unspecified behaviour to be learned at a later stage.

Transforming this partial specification to an executable control program proceeds in three steps. In the first phase a problem collector traverses the top-level control routine and collects all references to procedures with the status *to-be-learned* and stores them in the set of learning problems. In the second phase, the routines are learned. We consider a routine as learnable if it is *to-be-learned* and does not call any code *to-be-learned*. Therefore, the procedures are learned in an order that respects the dependencies between them, until all routines are learned. Finally, the complete program can be run to produce the behaviour, for instance playing robot soccer.

4 A Rational Reconstruction of the AGILO2001 Controller

With the extended language we have rationally reconstructed large parts of the action selection module of the AGILO autonomous soccer robots, with processes, data collec-

tion, learning system generation specified in constructs. The control program resulting from this reconstruction can automatically learn a repertoire of routines for playing robot soccer competently. In this section the learning process is demonstrated.

In the bootstrap learning phase the first learning task that is tackled is the learning of the robot dynamics. This learning task acquires a process model for the environment and does not depend on the solution of any other learning problems. To learn the dynamics we use the log files obtained from earlier RoboCup games and transform them into a pattern file for the learning system. For learning, we train an artificial neural network using the SNNS system in order to approximate the mapping: $state_i \times action_i \mapsto state_{i+1}$. In [4] this procedure is described in more detail.

The next learning task that the robot can address is how navigation tasks are to be achieved. We have already seen in the specification of this task in section 2.2. This is learned in a simulated environment, which only requires the learned robot dynamics. To solve the learning task as a function approximation problem we provide the robot with streams of control signals that generate good navigation trajectories. From this data the robot learns the mapping $state_{current} \times state_{goal} \mapsto action_{current}$, using a multi layer neural network. For a more detailed description see [5].

Once the controller can execute the navigation task, the robot learns a model of the task. Given the simulated dynamics and the controller learned in the two previous steps, the robot is given many automatically generated navigation tasks to execute. The duration of the task is recorded, and this training data is used to learn the mapping $state_{current} \times state_{goal} \mapsto time\_to\_complete$, again using a multi layer neural network. This mapping is necessary to coordinate the behaviour of our robots, the next step in creating the complete AGILO controller. How this is done is discussed in [5].

Since all these learning tasks are specified with constructs in the program, adaptation of the program is simplified. Learning tasks can be exchanged, without requiring a redesign of the program structure. The program just needs to check if any constructs are *to-be-learned*, and relearn them and any depending constructs. Another case example that shows the strength of making learning tasks explicit is when hardware is upgraded. Recently our robots have received new control-boards. Instead of having to repeat all learning steps to model these new boards manually (involving undocumented scripts, manual data copying and transformation, and so forth), we only have to provide the system with a log-file, and set all relevant learning tasks to *to-be-learned*. The dependencies between the learning tasks ensure that first the robot dynamics will be learned. Given these dynamics, which can be used in the simulator, the navigation tasks can be relearned. Given the navigation routines, models of the navigation routines can be learned, and so on, until the complete executable AGILO2001 controller is acquired. We are currently testing these procedures. Furthermore, the scope of learning tasks is being extended beyond the AGILO2001 controller, to include additional control tasks such as dribbling with fake movements, defending, shooting, and others. In addition, we are starting to tackle reinforcement learning problems.

5 Conclusions

In this paper, we have extended a control language with constructs for explicitly representing (1) the physical system that is to be controlled and (2) the learning problems to be solved. In the extended language entities such as control tasks, process models,

learning problems, and data collection strategies can be represented explicitly and transparently, and become executable. In the learning and execution phase, the entities are first class objects that control programs cannot only execute but also reason about and manipulate. These capabilities enable robot learning systems to dynamically redesign learning problems. We have also sketched the implementation of the learning phase and showed how a bootstrap learning method can automatically complete a partially defined control program by solving the stated learning problems. The extensions that we have presented are expressive enough to rationally reconstruct most of the AGILO2001 action selector. Other complex control systems need to be implemented using our approach and the conciseness and expressivity of our constructs need to be assessed and analyzed. We are just starting to incorporate optimizing learning techniques such as reinforcement learning into our approach.

We see the main impact of our framework along two important dimensions. From a software engineering perspective, the language extensions allow for transparent implementation of learning steps and abstract representation of complex physical systems. These aspects are typically not adequately addressed in current control systems, which makes them hard to understand and adapt to new requirements and conditions. The second dimension, which we find much more exciting, is the use of the framework as a tool for investigating more general and powerful computational models of autonomous robot learning. The programmability of learning systems, the modifiability of state representations, the possibility of reparameterizing learning systems, and the executability of learning specifications within the framework enables us to solve complex robot learning tasks without human interaction. The framework thereby enables us to investigate adaptive robot control systems that can autonomously acquire sophisticated skills and competent task control mechanisms for a variety of performance tasks.

References

1. S. Anderson, D. Hart, J. Westbrook, and P. Cohen. A toolbox for analyzing programs. *International Journal of Artificial Intelligence Tools*, 4(1):257–279, 1995.
2. D. Andre and S. Russell. Programmable reinforcement learning agents. In *Proceedings of the 13th Conference on Neural Information Processing Systems*, pages 1019–1025, Cambridge, MA, 2001. MIT Press.
3. M. Beetz. Structured Reactive Controllers – a computational model of everyday activity. In O. Etzioni, J. Müller, and J. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents*, pages 228–235, 1999.
4. S. Buck, M. Beetz, and T. Schmitt. M-ROSE: A Multi Robot Simulation Environment for Learning Cooperative Behavior. In *Distributed Autonomous Robotic Systems 5, LNAI*. Springer-Verlag, 2002.
5. S. Buck, M. Beetz, and T. Schmitt. Planning and Executing Joint Navigation Tasks in Autonomous Robot Soccer. In *5th International Workshop on RoboCup, Lecture Notes in Artificial Intelligence (LNAI)*. Springer-Verlag, 2001.
6. T. Dean and M. Wellmann. *Planning and Control*. Morgan Kaufmann Publishers, San Mateo, CA, 1991.
7. D. McDermott. A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University, 1991.
8. S. Thrun. Towards programming tools for robots that integrate probabilistic computation and learning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, San Francisco, CA, 2000. IEEE.