1 Team Kouretes (Κουρήτες)

Team Kouretes was founded in February 2006 and became active in the Four-Legged league. The team had its first exposure to RoboCup at the RoboCup 2006 event in Bremen, Germany, where it participated in the Technical Challenges of the Four-Legged league. At that time, Aibo programming by the team was done exclusively in the interpreted Universal Real-Time Behavior Interface (URBI), without any use of existing code. Subsequent work led to the participation of the team in the Four-Legged league of the RoboCup German Open 2007 competition in Hannover, Germany. The software architecture of the team was developed on the basis of previously released code by GT2004 and SPQRL 2006. In Spring 2007, the team began working with the newly-released Microsoft Robotics Studio (MSRS). The team’s software was developed from scratch exclusively in C# and included all the required services, as well as the motion configuration files for the simulated RobuDog robot of RoboSoft. The team’s participation in the MSRS Simulation Challenge at RoboCup 2007 in Atlanta, USA led to the placement of the team at the 2nd place worldwide bringing the first trophy home.

In Spring 2008 the team switched to the new robotic platform, the Aldebaran Nao humanoid robot, working simultaneously on the real robots and on the Webots and MSRS simulators and developing new code from scratch. In the recent RoboCup 2008 competition in Suzhou, China the team participated in all divisions of the Standard Platform league (Aibo robots, Nao robots, Nao Webots simulation, Nao MSRS simulation). The team’s efforts were rewarded in the best possible way: 3rd place in Nao league, 1st place in the MSRS simulation, and among the top 8 teams in the Webots simulation.

1.1 Team Leadership

Michail G. Lagoudakis is an assistant professor with the Division of Computer Science of the Department of Electronic and Computer Engineering (ECE) at the Technical University of Crete since 2005. He received his Ph.D. degree from Duke University, USA in 2003 and was a postdoctoral researcher at the Georgia Institute of Technology, USA until 2005. His research experience in robotics includes path planning, motion control, reinforcement learning, coordination.

Nikos Vlassis is an assistant professor with the Division of Production Systems of the Department of Production Engineering and Management (DPEM) at the Technical University of Crete since 2007. He received his Ph.D. degree
from the Technical University of Athens, Greece in 1998 and was an assistant professor with the University of Amsterdam, Netherlands until 2006. His current research interests include stochastic optimal control, unsupervised learning, and reinforcement learning. Vlassis has extensive experience with the RoboCup Simulation league and various distinctions with the UvA Trilearn robot soccer team, including the 1st position at the RoboCup world championship (2003), three times 1st position at the German Open tournament (2003, 2004, 2005), and the 1st position at the American Open tournament (2003).

1.2 Team Members

Team Kouretes 2009 includes five student members from two academic departments. The brackets indicate the main area each member is working on.

1. Eleftherios Chatzilaris, undergraduate (ECE) [Localization, Vision]  
2. Alexandros Paraschos, undergraduate (ECE) [Software Architecture]  
3. Jason Pazis, graduate (ECE) [Walk and Skill Learning]  
4. Walid Soulakis, undergraduate (DPEM) [Webots Simulation]  
5. Evangelos Vazaios, undergraduate (ECE) [Robot Communication]

2 Team Research

A significant amount of team effort is spent on discussing and trying innovative ideas according to the current research trends in robotics and machine learning. This team culture has been strengthened by requiring that the core team members are senior undergraduate students working on their diploma thesis on a RoboCup-related topic; this is currently the case for all student members.

2.1 Software Architecture

A critical decision in any software project developed by a group is the organization of the code or the software architecture. In our design we follow a divide-and-conquer approach. So, in order to naturally decompose the problem of controlling the robot, we use a modular architecture. Not only is this approach beneficial for the concurrent development of the source code by many people, it also provides a convenient platform for testing and debugging (due to module independence).

Agents Our approach to software architecture views the entire robot as a collection of agents. Each agent is responsible for one (or more) tasks on the robot. Each agent collects data form its resources, that is from the robot’s sensors, from the network, or from other agents and then, after deliberation, chooses an action. In this software architecture, it is possible to develop any type of intelligent agent from simple reactive agents to complex deliberative agents.
**Modules** Each agent in the architecture consists of a set of modules which are executed sequentially. Each module is responsible for registering a set of dependencies that must be satisfied before its execution. Dependencies take the form of a list of modules that have to be executed before the execution of the module that registers them. Given a set of dependencies, a sequential execution order of all the agent's modules must be determined. In case of cyclic dependencies a notification message informs the developer, and the execution is aborted. The execution order that satisfies all the dependencies, and thus the solution of the serialization, is found by solving a Constraint Satisfaction Problem (CSP). The architecture framework is responsible for a variety of tasks at run time: loading the set of modules specified in the configuration file, determining the execution order by evaluating the constraints, and monitoring the execution. The modules can be changed by issuing a special request to the framework at any time; following a successful change request, the execution order is recalculated and the execution resumes. Each module consists of two separate components: the representation and the provider. The representation is the structure that stores the data manipulated by the module, for example an estimate of the ball location in the current field of view stored as a pair of image coordinates; the provider is the method that updates the representation, for example a simple blob detector that extracts the center of the largest orange blob. The provider can only update the representation of the module it belongs to, but can read other representations (of the same agent's modules) via a blackboard architecture. A module can have more than one providers (with only one being active at each time), but only one representation. By allowing more than one providers, different methods for manipulated the same data can be tested and compared.

**Debugging** In order to boost the development process, the framework introduces basic, but nevertheless important, debugging capabilities. The debugging features can be grouped into two sets. The first set contains **cpp** directives, such as `INFO`, `DEBUG`, and `FATAL`, which write text strings in the appropriate stream or file. These directives can be used by the developer at crucial points to form a basic trace log. Finally, these directives are enabled only when the `-DDebug` flag is active when compiling, thus there is no needless consumption of computational resources when the robot is deployed. In the second set, a configurable logger can take a snapshot of the blackboard at every frame, when the execution loop has finished executing the last module. The logger can be configured to save only specific representations.

**Back-End** The backend of the software architecture is designed to support a variety of robotic platforms. Each agent in the architecture can be implemented either as a process, or as an autonomous thread, or as part of the main thread of the executable, depending on the designer’s choice. The only platform requirements posed by the architecture are the availability of a **cpp** (cross-) compiler for the onboard computer system and the appropriate configuration of the concurrency framework used by the robot's operating system. The default configuration
of the architecture uses Linux system calls and pThread libraries for thread implementation and thus covers the majority of robotic platforms used in RoboCup competitions. To maximize even more platform independence, a good policy is to control all direct calls to the platform API. To this end, robot sensors and actuators are only accessed through special wrapper modules, which are the only ones with direct access to the robot API. Due to restrictions governing the Nao programming interface, our architecture communicates with the Nao robot through the Aldebaran’s NaoQi middleware.

The Kouretes Agent The four basic modules that currently consist the single Kouretes agent are the following:

**Image Processor** This module is responsible for processing the image frames from the color camera. It first performs color segmentation followed by ball and goal detection to update the visual perception of the robot.

**Localization** This module is responsible for estimating the current position and orientation of the robot in the field using its visual perception.

**Behavior** This module is responsible for high-level decision making given the current perception of the robot.

**Special Actions** This module is responsible for loading and executing special actions designed using the Kouretes Motion Editor tool (Section 2.5).

**Communication** This module is responsible for communicating information between robots when instructed to do so by the other modules.

**Robot Controller** This module is responsible for communicating with the game controller and setting the robot’s state (start, stop, penalized, goal).

Finally, the last components of the architecture are the **Kmake** building system, a simple graphical tool for copying binaries and configuring settings on each robot, and several bash scripts for configuring the architecture.

2.2 Robot Communication

Communication among the robots and communication with the game controller are vital components of each RoboCup team in order to develop complex behavior and cooperation within the team. Led by this idea, our team developed a communication system, called **narukom**, which enables both robot-to-robot and robot-to-computer communication. Message exchange in **narukom** is accomplished easily using several different communication channels. The system is divided into three tiers: the API tier, the Control tier, and the Physical tier. These tiers are organized hierarchically with the API tier being on top.

**The API Tier** This level provides a flexible set of functions in order to exchange messages between the nodes of the network, while enabling the Control tier to access the blackboard of the robot. It has two set of functions: direct and indirect. Direct functions call control functions directly, whereas indirect calls use sockets
for calling functions. That gives the flexibility to use different programming languages as long as the programmer adheres to the specified protocol. Moreover, there is a memory proxy on this level providing interaction with the blackboard of the robot.

**The Control Tier** The Control tier is responsible for both composing the messages requested from API and for selecting the right channel of communication for transmitting the messages. Control keeps track of requests; these are either internal, made by other modules of the robot, or external, made by other nodes in the network. Additionally, it is responsible for delivering the received messages to the correct module and for updating the blackboard of the robot if necessary. The structure of the messages is described via XML for each type. This choice enables the nodes to exchange XML files in order to decode messages of unknown structure. The actual data can be transmitted via different channels; it is up to the control to decide which channel is more suitable at the time for sending messages. The network channel uses udp connection and byte streams for data exchange, as it is fundamental to keep the traffic in the network as low as possible. Finally, control creates one thread per actuator and sensor it uses for interacting with its environment. In case of channels such as network there is only one thread which serves both roles.

**The Physical Tier** The terms sensor and actuator are used in order to model the variety of ways a robot interacts with its environment. In our system a sensor is not just a button or a tactile sensor; a network interface qualifies as a sensor, as it receives data from the environment. Respectively, actuator is anything that can influence the robot’s environment, such as speakers and/or robot leds. So, the lowest tier of the architecture realizes possible communication means through interaction with the environment. Every sensor receives a stimulant and interprets it as a message which is then delivered to the Control tier. Respectively, an actuator takes the messages from the Control tier and converts them to a form suitable for a certain communication medium (audio or light). Generally speaking, every sensor and actuator have their own threads, but in cases such as network interfaces which could serve both as sensor and as actuator there is only one thread created which serves both roles.

### 2.3 Vision

The ability of an autonomous robot to sense its environment is crucial for effective action selection. Our vision module tries to extract critical information from images taken by the CMOS VGA camera on the Nao’s head and consists of two consecutive steps: color segmentation and object recognition.

**Color Classification** The CMOS camera on the Nao delivers images in the YUV format. In order to meet real-time requirements, we are using a lower camera resolution (320×240) and a low frame rate (4 fps). Each point in the YUV
color space corresponds to a certain color. A color class is a subset of the color space, enclosing all variations of a certain color of interest in the real world. For RoboCup SPL games, the basic colors and objects of interest are orange (ball), skyblue (goal posts), yellow (goal posts), white (lines), green (field), red (team color), and blue (team color). For color classification we use modern classifiers with a current focus on decision trees (DT) trained using the C4.5 algorithm [1]. YUV values received by the camera are the inputs to the classifier and a color class label is the output. Regarding the number of input attributes, the simplest choice is to use the YUV values of the current pixel (N1 scheme). However, under difficult lighting situations, we can exploit color locality by using as input attributes not only the YUV values of the current pixel, but also the YUV values of pixels in its immediate neighborhood. These can be the 4 orthonormal, the 4 diagonal, or the 8 orthonormal/diagonal neighboring pixels. Through extensive experimentation we found that the neighborhood of the 4 orthonormal neighboring pixels works best for our purposes; under this scheme, the YUV values of 5 pixels in total are taken as the input attributes (N5 scheme).

The Kouretes Color Classifier The correct color segmentation of an image requires that a new classifier is learned for each location and lighting scheme used. In general, a large number of images must be taken and labeled, which often requires up to one hour of manual work. In order to make this procedure more precise and efficient we created the Kouretes Color Classifier (KC²) tool [2], shown in Figure 1. For training purposes, we need to take samples for each color class from several camera images and manually label them for training the classifier. Large areas of the raw image taken by the Nao can be selected

Fig. 1. The Kouretes Color Classifier (KC²) tool for generating training data.
using a graphical interface and all pixels within each area are associated with the desired color class label. The YUV locus (projected in each of the three dimensions) is displayed during the process. Once a sufficient number of data has been collected from several images, we can train the classifier. Figure 2 demonstrates our color segmentation method. A decision-tree classifier has been trained under the N1 scheme using over 50,000 training examples by labeling various regions in images taken by the robot camera. This moderate number of training data works well in avoiding under- and over-fitting. Similarly, another decision-tree classifier has been trained under the N5 scheme using over 100,000 training examples; the increase in the number of training data is necessary given the increase in the number of input attributes. The segmented image for each of these two classifiers is shown in Figure 2. In either case, the colors of interest have correctly been identified, however the N5 scheme yields much better results than the N1 scheme. Changing lighting conditions pose a severe challenge for color classification. This is particularly important this year, since the RoboCup 2009 competitions will be held using only the ceiling lights of the venue. To remedy the problem of varying illumination, we have provided three additional inputs to our classifier: the average values of the three color dimensions (Y, U, V) over the image (N5+3 scheme). These features provide information about the global illumination level. Figure 3 demonstrates the benefits of the N5+3 scheme over the N5 scheme on the dark raw image (2nd from left), when trained with data from both the bright and the dark images (60,000 examples from each image); N5 is trained only on the bright image.

Object Recognition The second step in our vision module is the recognition of objects of interest in the color-segmented images. The current object recognition procedure focuses only on three objects: the ball, the yellow goal, and the skyblue goal. Given that all three objects are uniformly colored by a single color, our method scans through the color-segmented image for blobs of the target color (orange, yellow, skyblue) and extracts a number of statistics if such a blob is found: the center of the blob, the elongation of the blob, the direction of the principal component of the blob, and the size of the blob as a ratio of blob pixels over total image pixels. The elongation and the ratio are used as filters for rejecting false positives. The threshold numbers for the elongation and the ratio
Fig. 3. Original (two illuminations) and segmented images (N5 and N5+3 schemes).

are determined empirically in the field. This simple method works surprisingly well, however for robustness it needs to be improved to include shape information and to take into account the current robot pose for determining the horizon line. Our current focus is on porting to Nao the histogram-based object recognition method developed by our team for the Aibo robots [3], given that the key objects in the Nao field are similar to the objects used in the Aibo field, except for some differences in size [4].

2.4 Localization

Self-localization of the robots in the field is accomplished using Monte Carlo Localization with Particle Filters implementation. The belief of the robot is a probability distribution over the 3-dimensional space of \((x, y, \theta)\), where \(x, y\) are the robot coordinates on the field from a bird’s eye view and \(\theta\) is the orientation of the robot with respect to the line that crosses the center of the goals. The belief is represented approximately using a population of particles. In order to perform belief update, it is necessary to obtain a motion model for the available high-level actions of the robot (walk, turn, etc.) and a sensor model for its landmark perception capabilities (goal recognition) over the \((x, y, \theta)\) space. We constructed a simple motion model \(P((x', y', \theta')|(x, y, \theta), a)\) for predicting the probability distribution of the pose of the moving robot after each action \(a\) and a simple sensor model \(P(z|(x, y, \theta))\) for correcting the belief towards the poses where the probability of obtaining the current observation is higher. The parameters of these models were learned through experimentation in Webots [5].

Having the motion and the sensor models, belief update is performed using an Auxiliary (AUX) particle filter:

- temporarily propagate the particles through the motion model
- temporarily weigh each particle using the sensor model and normalize
- resample the original particle population using the temporary weights
- propagate the particles through the motion model
- weigh each particle using the sensor model and normalize
- resample the weighted particle population to eliminate the weights

For the resampling process, Selection with Replacement and Linear Time Re-sampling have been implemented. Given any population of particles, the player’s pose is estimated as the robust mean of the weighted particles. Figure 4 shows
Fig. 4. Localization: initial and updated belief, particles, true and estimated path.

a global localization example where the robot begins at the center of the field without any information about its current location and after several steps the estimated location is close to the true one.

2.5 Special Actions

The Kouretes Motion Editor (KME) is an interactive software tool [6, 7] we have developed for designing complex motion patterns on the Nao, but also on other robots with many degrees of freedom, using intuitive means. The main idea behind KME is the ability to generate, capture, store, manipulate, edit, replay, and export timed sequences of complete robot poses, which resemble the desired complex motion pattern. KME allows for interactive design through a TCP/IP network connection to a real or simulated robot, over which various robot poses can be communicated to or from the robot and manipulated locally using the KME graphical user interface. This portability and flexibility enables the user to work under different modes (configuration or cartesian space), with different robots (real or simulated), using different host machines (for running KME itself). KME was originally designed for and currently supports only the Nao robot and its simulated model on the Webots simulator [5]. However, the main features of KME can be easily reconfigured for other robots through an XML configuration file and the tool itself could be used for a variety of purposes, such as providing complex motion patterns as starting points for learning algorithms and supporting educational activities in robot programming courses.

The goal behind the development of KME is to provide an abstract motion design environment for the common robot practitioner, which hides away the technical details of low-level joint control and strikes a balance between formal motion definition using precise joint angles in the configuration space of the robot and intuitive motion definition using manual joint positioning in the real-world work space of the robot. Such an abstraction yields a number of benefits: (a) arbitrarily complex motion patterns can be easily designed without ever writing a single line of code, (b) motion patterns can be rapidly designed and
tested through a simple and friendly interface, (c) motion patterns designed by one user can be easily shared, understood, used, and modified by other users, (d) various real and/or simulated robots can be accommodated simply by reconfiguring the back-end of the tool, (e) resulting motion patterns can be used as seeds in learning algorithms for further fine-tuning, and (f) proprietary motion patterns could be reverse-engineered as recorded sequences of complete or partial robot poses and subsequently manipulated at will. Of particular interest to RoboCup teams is KME’s ability to export the symmetric motion (with respect to the sagittal plane of the robot), as well as the temporally reverse motion. In practice, that implies that it is sufficient to design a right kick; the left kick can be automatically exported. Similarly, a stand-up motion if reversed will result in a sit-down motion. We have used KME to design a number of complex motions, such as kicks, goalkeeper dives, stand-up routines, and recently complete choreographies. Figure 6 shows a right kick motion designed using KME.

2.6 Walk
Stable walk on the Nao robots is probably the grand challenge for all participating teams. Our efforts to produce a walk engine using open-loop gait patterns led in little progress, therefore we currently use the proprietary walk functions provided by Aldebaran. The provided walk functions, however, do not consist an off-the-self solution. The various walk parameters need careful tuning separately.
for each surface, for each robot, and for each walk type. Despite the tedious process of manual tuning, we can derive a separate set of good parameters for each one of our robots and for each one of the walk types we used (forward, backward, turning, side-step) on a given carpet using some simple Python scripts.

Our current efforts focus on learning closed-loop gait patterns by formulating the problem as sequential decision making and using reinforcement learning techniques. The high-dimensional, continuous action space (11 joints just for the lower robot body) poses a grand challenge for most reinforcement learning algorithms. However, our recent work on Adaptive Action Modification [8] and Binary Action Modification [9] indicates that realistic problems with continuous action spaces can be tackled efficiently. The benefits of a learning approach include the derivation of a closed-loop walk policy, as well as the ability to adapt the walking gaits for optimizing several factors (accuracy, speed, stability, power consumption, mechanical stresses).

2.7 Behavior

The behavior of each player is currently determined by a simple behavior module which is implemented as a Finite State Machine (FSM) shown in Figure 7. The logic behind this FSM is straightforward and varies depending on the role of the player (goal keeper or attacker).

The goal keeper searches for the ball using only the pan and tilt capabilities of the head. Once the ball is found, the goal keeper uses side-steps to align himself between his goal and the ball, provided that the ball is still far away and poses no danger. However, if the ball is close and approaches from the left or the right side (indicated by the head pan exceeding some threshold value), then the goal keeper initiates a left or a right fall respectively. This action protects the goal on either side and upon completion brings the robot back to its initial position from where the FSM restarts.

The attacker searches for the ball using a combination of head pan and tilt motions (to cover the effective visual field), rotation in place (to cover all angles around the current position), and forward or backward walk (for repeating the search in another location). As long as the ball is not found this search procedure continues, however if the own goal becomes visible the player will prefer to walk towards it as opposed to walking towards the opponent goal; the purpose is to get behind the ball to initiate a new attack on the opponent once the opportunity appears. Once the ball is found, the attacker turns towards the ball using rotation
in place and approaches it by straight walk. When close to the ball, the Attacker uses side-steps to align the ball with the kicking leg. Upon successful alignment, a kick is initiated and the procedure is repeated.

Acknowledgements

Team Kouretes would like to thank the administration of the Technical University of Crete for research and travel funding. The team was also partially supported by the European Grant MCIRG-CT-2006-044980.

References