Wiktor W Brodlo (s0927919) – SDP Performance Review 2 February 16, 2012

I have developed a simulator of the robot to help the group with developing strategy. As of writing of this report, the simulator is almost done, it only requires some "glue" with the rest of the system.

I have started developing the simulator by writing a simple class that implements the communication interface, replacing the usual Bluetooth communication. Once I was sure I can receive the commands the robot would, I went on to build a system that would be able to receive inputs from two control systems, perform the actions the robot would, and assemble them into a fake camera feed. My original plan was to connect this fake feed to the vision system, but after some discussions with the group I have abandoned the idea, and instead decided to create a simulated world state and pass it to the strategy system. This part still needs some work as the world state needs some revamp and documentation.

The simulator runs in a separate process from the rest of the system, as including a simulator in the already "heavy-duty" loop in the main system would slow things down to a crawl. The simulator process communicates with the rest of the system though the socket API. Sockets were chosen because they are a well-supported, simple IPC solution. It also allows for the simulator to be run on a separate machine if needed. The simulator opens server sockets on three ports (default ports are defined on compile-time, and can be set upon execution through command-line parameters). Two of these are for the fake robots the control system will connect to, the other one is for communicating the world state back. Once the sockets are opened, three threads are created – one for each fake robot and one for world state. The threads are passed the appropriate sockets and will accept a connection.

Once at least one robot is connected, the simulation begins. The two robot treads each create an "action" thread. This thread simply executes a timed loop. The robot threads accept commands from the control system, decode them and write them into a shared memory structure. Every time the action loop restarts, it checks for the command currently in memory and executes it, updating the simulator-internal world state.

When the world state thread receives a connection, it starts a timed loop which reads the world state from memory, simulates the ball, and sends the updated world state through the socket. The world state is updated and the strategy system can continue executing its plan.

The simulator-internal world state is simply a set of three structures. Two store information relating to the respective robot. The stored information is position along x-axis, position along y-axis, orientation angle, and the state of the kicker (up or down). The third structure stores the position of the ball.

The simulator can accept a control connection for both yellow and blue robots, or just one (creating a static "dummy" for the other). This allows us to play matches against ourselves. I am also planning to enable the operator to control the dummy manually, as well as recording and replaying sessions.

The threaded approach allows for all actions to happen asynchronously of each other. This way the simulator will work fast and there is no need for a large processing loop. This approach also felt "natural" to me – when I thought of simulating independent actions, threads seemed just the right thing to do. Because all operations are atomic (reading/writing integers of x- and y-positions), there is no need for access control. The world state thread is free to read the state as often as it needs to, while the action threads will update their respective robot states at their own pace.