

Ramses the Rat: CiberMouse Agent

University of North Carolina – Chapel Hill

Björn B. Brandenburg
bbb@cs.unc.edu

Hennadiy Leontyev
leontyev@cs.unc.edu

1 Introduction

Ramses the Rat is the name of the University of North Carolina’s entry in the 2006 CiberMouse competition held at the Real Time Systems Symposium (RTSS) 2006 in Rio de Janeiro.¹ This paper gives an overview of the design and implementation of the agent and the approaches chosen to solve the challenges posed by the competition.

During the development of Ramses the Rat, special attention was paid to the applicability of proportional-share algorithms to sensor request scheduling. The software architecture of our agent was designed with an emphasis on composability and adaptability.

The rest of the paper is organized as follows. Section 2 describes the architecture of the robot and details our approach to complex robot behavior. Section 3 describes the strategy that the robot employs to locate the beacon. Section 4 presents the strategy that is used by the robot to return to its initial position.

2 The Framework

The architecture of the robot clearly separates modules responsible for the behavior of the robot (planing, collision avoidance, control logic) from those offering an abstraction of the underlying “virtual hardware.” Core services include position estimation, mapping, sensor access, sensor scheduling, and actuator control. These services provide a high-level view of the current state of the robot and the surrounding world. A

planning and control framework that keeps track of the current goals and selects, monitors, and aborts easily exchangeable strategies resides on top of the service layer.

The modular design allowed us to evaluate — in spite of limited resources — several different strategies.

2.1 Actuator and Sensor Access

Actuator access is governed by a thin wrapper that, during each cycle, keeps track of any actuator commands issued and enacts the requested changes after the strategy framework has finished its control computation. This approach simplifies strategy development because it frees strategy implementations from keeping track of actuator states, avoids unnecessary network transmissions, and eases debugging by providing actuator command logs.

The sensor module controls access to the underlying library in a similar fashion and provides the latest sensor data, including information concerning the data’s age.

2.2 Sensor Scheduling

The limitation on the number of sensor readings per cycle requires careful scheduling of the available bandwidth. Forcing each strategy to deal with sensor requests itself would lead to both code duplication and an increase in complexity. We chose to automate sensor requests by implementing a simplified, lag-based proportional-share scheduler variant inspired by EEVDF [1].

We applied the algorithm as follows: Each sensor that must be scheduled is represented as a task T_i

¹<http://www.ieeta.pt/~lau/web.ciberRTSS/>

with unit execution cost. Only two tasks can be scheduled in any cycle. Each task has a weight $w(T_i) \in [0, 1]$, the sum of all weights of all tasks equals two. For each task T_i , we compute each cycle t the value $lag(T_i, t) = w(T_i) \cdot t - A(T_i, t)$, where $A(T_i, t)$ is the number of times the sensor T_i has been read since the start of the schedule. Each cycle, the robot requests updates for the sensors that have accumulated the highest lag (ties are broken arbitrarily). This policy ensures the frequency of sensor requests to be in accordance with their weights. Table 1 shows the default sensor configuration.

The robot can reweight the sensors at any time. This causes the information about weights, lags, and allocations to be dropped and a new schedule to be generated. To enable per-strategy sensor weights, we also allow saving and restoring sensor weights in a stack-like manner.

Sensor	Weight
Front	0.75
Left	0.5
Right	0.5
Rear	0
Beacon	0.25
Ground	0

Table 1: Default sensor configuration.

2.3 Navigation

The navigation module is responsible for maintaining a map of the arena. It monitors the commands issued to the actuators and uses the knowledge of the robot’s inertia model to estimate the position of the robot relative to its starting position. The navigation module records the estimated position and allows position data to be queried based on a time stamp or a reference position.

Note, that we do not use the compass. Relying solely on integration over the actuator commands proved be more accurate than incorporating data from the quite noisy compass.

2.4 Beacon Location

The navigation module also estimates the position of the beacon. It stores the position and direction of beacon sightings and predicts the likely position of the target location using triangulation. The approach that we use computes intersecting points between all recorded sightings and predicts the target to be at the center of the resulting point set. By assigning each intersection a weight $c \in [0, 1]$ based on the angle between the sightings — referred to as the *confidence level* — we can reason about the quality of the current prediction. This, for example, allows strategies to ignore any target location estimation with a confidence level of less than 40%.

2.5 Strategy Framework

The complete control logic of the robot is contained in the control and planning framework. The main design goal for the framework was to enable the creation of small, interchangeable control elements that allow the easy reuse of existing robot behavior in new contexts. For example, the task of approaching a point is independent of whether the robot is approaching the beacon or returning to the starting position. Implementing such control laws multiple times would be a waste of resources and an unnecessary source of errors. Instead, our framework allows multiple primitive control actions to be synthesized into intricate strategies that can issue high-level commands like “approach point” and “avoid collision” instead of having to deal with actuators directly. This is realized by decomposing the robot’s behavior into three basic building blocks: *control actions*, *goal predicates*, and *strategies*.

Control Actions Control actions define the low-level behavior of the robot. They abstract a set of activities that the robot can perform in a single simulation cycle, such as checking for imminent collisions, control law computations, and issuing actuator commands.

As an example, consider the *ApproachPoint* control action. It first checks the current sensor readings for nearby obstacles. In case of a threshold violation, it stops the motor and indicates failure. If the robot has sufficient space for movement, the control action computes the distance from the current

location to the target position and the direction relative to the current heading of the robot. Using these two inputs, a control law is computed to determine the appropriate actuator commands.

Goal Predicates As the name implies, goal predicates indicate whether a given goal has been accomplished by the robot. They do not issue any commands to the robot. Examples for goal predicates include checking whether the beacon has been found, whether a wall exists in close proximity, and whether a certain distance has been traveled. Goal predicates can be combined using boolean logic to form more complicated tests.

Sometimes, it is useful to combine control actions and goal predicates. For example, the *ApproachPoint* control action also defines a predicate that indicates whether the point has been reached.

Strategies Low-level control actions and goal predicates are used in high-level strategies that abstract long-term activities that are expected to require more than one simulation cycle to accomplish. While being active, strategies are responsible for selecting and executing the appropriate control actions required to reach their goal. Strategies heavily reuse existing control actions and even complete strategies, creating an open hierarchy of increasingly abstract and intricate behavioral components. Examples of strategies include locating the beacon, following a wall, and returning to the starting position.

The first strategy to execute is given to the robot as a configurable option. It is the responsibility of the initial strategy to start sub-strategies that accomplish partial milestones. First, the initial strategy launches a sub-strategy to locate and approach the beacon. When the target is found, the sub-strategy indicates success and a second sub-strategy guides the robot back to its starting position. Once the initial strategy finishes, the mission has been completed and the robot shuts down. Sections 3 and 4 describe the two major sub-strategies.

3 The Beacon Approach Strategy

Approaching the beacon is divided into three steps: (i) estimating the beacon's position, (ii) approaching

the estimated target area, and (iii) discovering the beacon in the vicinity of the estimated position.

3.1 Locating the Beacon

While the estimated position of the beacon is unknown or the confidence level of the estimated position does not exceed a user-defined threshold, the robot explores the arena by randomly bouncing off of obstacles. The navigation module records and incorporates incoming beacon sightings during the exploration phase and thereby continuously improves the prediction of the beacon's position. This approach works especially well in sparse mazes. A further refinement of this technique will keep track of already visited areas to prevent the robot from being trapped in a loop.

3.2 Approaching the Target Area

After the robot is confident about its estimate, it reconfigures the sensor weights in order to activate the ground sensor. The robot approaches the target area following a straight line. In case that it encounters an obstacle on the way (presumably a wall), it tries to overcome the obstacle by following its shape in a randomly selected direction. While the robot is following the obstacle, it checks whether it can advance in the direction of the estimated target area. If possible, it abandons the obstacle and tries to approach the estimated position again.

This strategy works well in sparse mazes that offer several different ways to approach the beacon, however, its performance degrades when confronted with curved walls. Plans for future developments include annotating positions near walls in the navigation history to support improved obstacle avoidance heuristics.

3.3 Beacon Discovery

After successfully reaching the target area, the robot reweights the sensors to increase the rate of ground sensor readings significantly. If the estimate does not accurately correspond to the actual position, the robot tries to approach the correct position by following the direction of incoming beacon sightings. In case of insufficient incoming sensor

data, the robot wanders randomly in the vicinity of the estimated position and awaits fresh sensor data. When the robot detects that it has reached the target, it activates the visiting LED and signals success to the initial strategy.

4 The Return Strategy

As the robot builds an internal map of the environment while searching for the beacon, getting back to the starting position is considerably easier than the task of finding the beacon. The world — especially its major obstacles — has already been (partially) explored by the robot, and there exists at least one path to the starting position that the robot has knowledge of.

4.1 Route Planning

One possible return strategy is to discover a new path based on the known locations of major obstacles and the position of the beacon relative to the starting point. In case the beacon has been located only after a long search, this approach could be quite advantageous because the only known path back to the starting position may be long and intricate. Thus, computing and following a direct path to the starting point may reduce the length of the return route significantly. Unfortunately, this approach has some drawbacks, too. For example, if a simple route connecting the starting point and the beacon position were to exist, the exploration strategy would have most likely found it already. Therefore, it is quite possible that the robot will run into blocking obstacles on the way back, necessitating time consuming evasive maneuvers. This nullifies the advantage of having found a shorter path.

In light of these uncertainties, we chose to implement a different strategy. Because the maps are generally small (with regard to the size and speed of the robot), returning via the way the robot reached the beacon does not take much time even if it is unnecessarily long. Thus, our robot inverts the recorded path and uses it as guidance. Frequently, the path will contain loops created by unsuccessful exploration attempts. To improve performance in this common case, the robot scans for crossing path

segments and takes short cuts when possible.

4.2 Overrun Mitigation

Scanning the return route for loops and filtering unnecessary segments requires many comparisons of points in the robot's movement history. In order to avoid overrunning the relative deadline of 25ms, care must be taken to reduce the computational effort spent on planning. By computing the next way point only on demand we spread out the work over time. On top of that, a simple space partitioning scheme allows us to limit the number of comparisons between points by discarding distant path segments: only segments in local grid cells need to be examined.

An extension of this scheme guarantees hard bounds on the number of comparisons (and thus on the worst case execution time) by only admitting a limited number of recorded positions into each grid cell.

5 Conclusion

Looking back at the development of Ramses the Rat, we conclude that the use of a highly modular architecture has been a success. It has allowed us to develop and evaluate more strategies than originally planned and to quickly adapt existing behavior to deal with observed problems. We are eagerly anticipating the chance to evaluate our strategy's performance in comparison to other approaches at the CiberMouse 2006 competition.

References

- [1] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C.G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299. IEEE, 1996.