GermanTeam 2007
The German National RoboCup Team

Thomas Röfer[1], Jörg Brose[2], Daniel Göhring[3], Matthias Jüngel[3], Tim Laue[4], and Max Risler[2]

[1] Deutsches Forschungszentrum für Künstliche Intelligenz, Safe and Secure Cognitive Systems, Robert-Hooke-Str. 5, 28359 Bremen, Germany
[2] Fachgebiet Simulation und Systemoptimierung, Fachbereich Informatik, Technische Universität Darmstadt, Hochschulstraße 10, 64289 Darmstadt, Germany
[3] Institut für Informatik, LFG Künstliche Intelligenz, Humboldt-Universität zu Berlin, Rudower Chaussee 25, 12489 Berlin, Germany.
[4] Fachbereich 3 - Mathematik / Informatik, Universität Bremen, Postfach 330 440, 28334 Bremen, Germany

`http://www.germanteam.org`
`germanteam@tzi.de`

1 Introduction

The GermanTeam participates as a national team in the RoboCup Four-Legged League since 2001. It currently consists of students and researchers from the Humboldt-Universität zu Berlin, the Universität Bremen, and the Technische Universität Darmstadt. After winning the technical challenge in 2003 and the world championships in 2004 and 2005, the team only reached place four in 2006. Therefore, a major overhaul of the whole software system was necessary in order to stand a reasonable chance in RoboCup 2007. As in previous years, the team description paper only contains information on improvements not published elsewhere. So in addition to the research described here, four publications were accepted for the RoboCup Symposium 2007 on image-processing [1], on cooperative world modeling [2], on self-localization [3], and on improving vision-based distance measurements [4].

This team description paper mainly focuses on changes in the infrastructure of the team's software system: first, the new module framework is presented. Afterwards, improvements in the behavior description language XABSL are briefly described. A new framework for performing experiments that require ground truth is introduced. Finally, the compromise found for modeling ball position and speed is outlined.

2 Module Framework

2.1 Original Module Framework

The major goal of the original architecture was the ability to support the collaboration between the university-teams in the German national team. Some tasks may be solved only once for the whole team, so any team can use them. Others will be implemented differently by each team, e. g. the behavior control. The building blocks of the architecture were called *modules*. For each module, several *solutions* can exist. All solutions of a certain module share the same interface. Interfaces are comprised of input and output *representations*. Representations represent the actual data exchanged between the modules. A rather static set of modules was defined to be able to play robot soccer in the Four-Legged League. To be able to easily compare the performance of different solutions for same module, it is possible to switch between them at runtime. In depictions of the module layout such as shown in [5], the modules are grouped in the four layers *Perception*, *World Modeling*, *Behavior Control*, and *Motion Control*. In its first incarnation, these layers were strict in the sense that modules in each layer only used representations provided by the previous layer. However, these rules were relaxed over time.

In general, the original architecture had a couple of drawbacks:

– The module configuration was quite static. Although adding new solutions for existing modules was rather simple, adding new modules was not. Any new control idea had to somehow fit into the existing structure.
– Since all solutions for a module share the same interface, this interface had to contain the superset of all the representations used by the individual solutions. Therefore, the interfaces lost their function to describe what representations were actually required, because many of them were only necessary for some more exotic solutions, but not for the ones actually used in the games.
– The sets of output representations of the modules were more stable, i. e. they were rarely changed. Instead, the representations themselves grew more and more complex over time, because different solutions used different ways to represent the information. For instance, the position of the ball may be represented by a 2-D position, an additional co-variance matrix, or a sample-set. All these different kinds of information were stored in a single representation, and since not every solution filled in all the fields, certain solutions for one module only worked together with certain solutions for another module. There was no support to maintain these kinds of inter-dependencies, and this resulted in several mistakes, even at competitions.

To avoid these problems in the future, a new module framework was developed that consists of the *blackboard*, the *module definition*, and a visualization component.

2.2 Blackboard

The blackboard [6] is the central storage for information, i. e. for the representations. Each process has its own blackboard. Representations are transmitted through inter-process communication if a module in one process requires a representation that is provided by a module in another process. The blackboard

```
class BallPercept;
class FrameInfo;
// ...
class Blackboard
{
protected:
  const BallPercept& theBallPercept;
  const FrameInfo& theFrameInfo;
// ...
};
```

Fig. 1. The blackboard.

itself only contains references to representations, not the representations themselves (cf. Fig. 1). Thereby, it is possible that only those representations are constructed, that are actually used by the current selection of modules in a certain process. This goes as far that a process that does not contain any module that uses a certain representation will not contain any code about that representation at all. Thus, both the file size of executables and their memory requirements stay minimal. For instance, the process *Motion* does not process camera images. Therefore, it neither requires to instantiate an image object (approximately 100 KB in size), nor to link the corresponding binary file into its executable, which reduces turn-around times.

2.3 Module Definition

The definition of a module consists of three parts: the module interface, its actual implementation, and a statement that allows to instantiate the module (cf. Fig. 2). The module interface defines the name of the module (e.g. *MODULE(SimpleBallLocator)*), the representations that are required to perform its task, and the representations provided by the module. The interface basically creates a base class for the actual module following the naming scheme *ModuleName*Base. The actual implementation of the module is a class that is derived from that base class. It has read-only access to all the required representations in the blackboard (and only to those), and it must define an *update* method for each representation that is provided. As will be described in Section 2.4, modules can expect that all their required representations have been updated before any of their provider methods is called. Finally, the *MAKE_MODULE* statement allows the module to be instantiated. It has a second parameter that defines a category that is used for a more structured visualization of the module configuration (cf. Sect. 2.6). The categories resemble the idea of the layers in the original architecture.

The module definition actually provides a lot of hidden functionality. Each *PROVIDES* statement makes sure that the representation provided can be constructed and deconstructed (remember, the blackboard only contains references), and will be available before it is first used. In addition, representations provided

```
MODULE(SimpleBallLocator)
  REQUIRES(BallPercept)
  REQUIRES(FrameInfo)
  PROVIDES(BallModel)
END_MODULE

class SimpleBallLocator : public SimpleBallLocatorBase
{
  void update(BallModel& ballModel)
  {
    if(theBallPercept.wasSeen)
    {
      ballModel.position = theBallPercept.position;
      ballModel.wasLastSeen = theFrameInfo.frameTime;
    }
  }
}

MAKE_MODULE(SimpleBallLocator, World Modeling)
```

Fig. 2. Simple example of a module (only #include statements omitted)

can be sent to other processes, and representations required can be received from other processes. The information that a module has certain requirements and provides certain representations is not only used to generate a base class for that module, but is also available for sorting the providers (cf. next section), and can be requested by a host PC. There it can be used to change the configuration, for visualization (cf. Sect. 2.6), and to determine which representations have to be transferred from one process to the other. Please note that the latter information cannot be derived by the processes themselves, because they only know about their own modules, not about the modules defined in other processes. Last but not least, the execution time of each module can be determined and the representations provided can be sent to a host PC or even altered by it.

2.4 Configuring Providers

Since modules can provide more than a single representation, the configuration has to be performed on the level of providers. For each representation it can be selected which module will provide it or that it will not be provided at all. In addition it has to be specified which representations have to be shared between the processes, i. e. which representations will be sent from one process to the other. The latter can be derived automatically from the providers selected in each process, but only on a host PC that has the information about all processes. Normally the configuration is read from a file during the boot-time of the robot, but it can also be changed interactively when the robot has a debugging connecting to a host PC.

```
Input:
  $P = [p_1 \ldots p_n]$ : List of selected providers of representations
  $S$ : Set of representations provided by other processes, $P \cap S = \emptyset$
Output:
  $P$ is ordered so that all representations are provided before they are required
Algorithm:
  $R := S$ : Set of representations already provided
  for $i := 1 \ldots n$
    $j := 0$ : Number of unsuccessful attempts
    while $\left(\exists_{r \in \text{requirements}(p_i)} r \notin R\right) \land j \leq n - i$
      $P := [p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n, p_i]$
      $j := j + 1$
    if $j \leq n - i$ then $R := R \cup \{p_i\}$ else fail
```

Fig. 3. Pseudo code of the algorithm to sort the providers

The configuration does not specify the sequence in which the providers are executed. This sequence is automatically determined at runtime based on the rule that all representations required by a provider must already have been provided by other providers before, i. e. those providers have to be executed earlier. Figure 3 shows the solution of this constraint satisfaction problem. Basically, the list of providers is traversed one by one. For each position in the list, it is check for all remaining providers whether they would be satisfied with all the representations provided so far. If one is found, it is kept at that position and the algorithm continues with the next position. If no such provider can be found, the set of providers is inconsistent and no solution can be found.

2.5 Pseudo-Module *default*

During the development of the robot control software it is sometimes desirable to simple deactivate a certain provider or module. As mentioned above, it can always be decided not to provide a certain representation, i. e. all providers generating the representation are switched off. However, not providing a certain representation typically makes the set of providers inconsistent, because other providers rely on that representation, so they would have to be deactivated as well. This has a cascading effect. In many situations it would be better to be able to deactivate a provider without any effect on the dependencies between the modules. That is what the module *default* was designed for. It is an artificial construct—so not a real module—that can provide all representations that can be provided by any module in the same process. It will never change any of the representations—so they basically remain in their initial state—but it will make sure that they exist, and thereby, all dependencies can be resolved. However, in terms of functionality a configuration using *default* is never complete and should not be used during actual games.

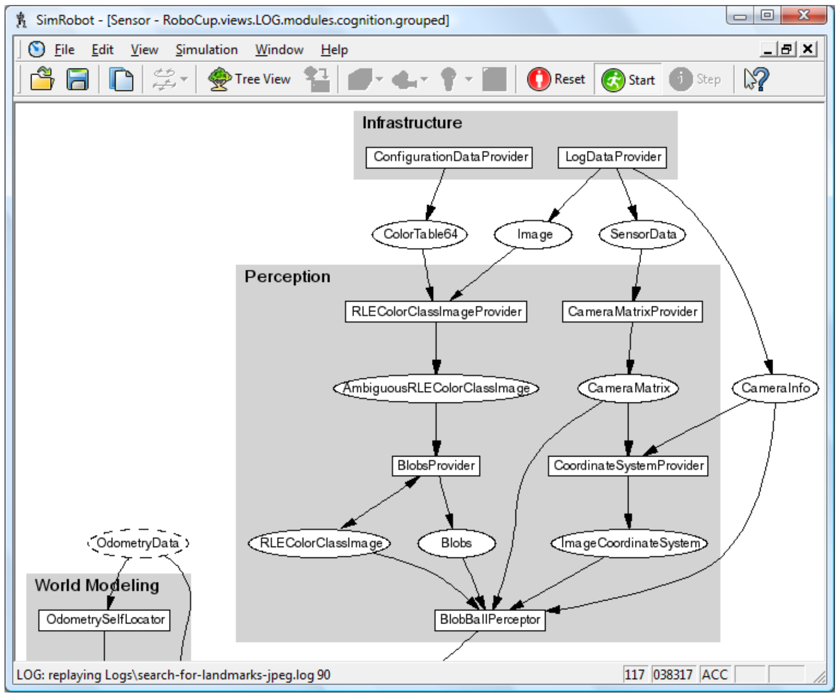
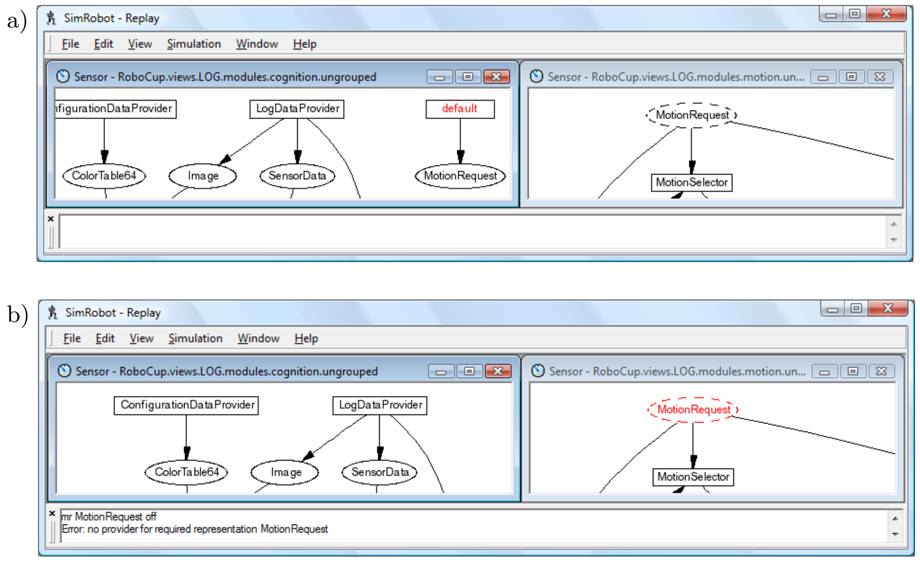


Fig. 4. A module configuration as displayed in the GUI

2.6 Visualization

Since all the information about the module configuration can be transferred to a host PC, it is possible to automatically generate a visual representation. The graphs such as the one that is shown in Figure 4 are generated by the program *dot* from the *Graphviz* package [7] in the background. Modules are displayed as rectangles and representations as ellipses. Because of its special status, the module *default* is displayed with red text (cf. Figure 5a). Representations that are received from another process have a dashed border. If they are missing completely, they are entirely displayed in red (cf. Figure 5b). The modules can be grouped by the categories that were specified as second parameter of *MAKE_MODULE*.

Figure 5 shows the effect of interactively changing the module configuration for the two processes *Cognition* and *Motion*. The representation *MotionRequest* is normally generated in the process *Cognition* and then transferred to the process *Motion*, where it is the input for several modules that generate robot motion (walking, kicking, etc.). If its generation is deactivated in *Cognition*, it cannot be sent to *Motion* anymore. Hence, it is missing there. This shows that the visualization is updated instantly for any change in the configuration.

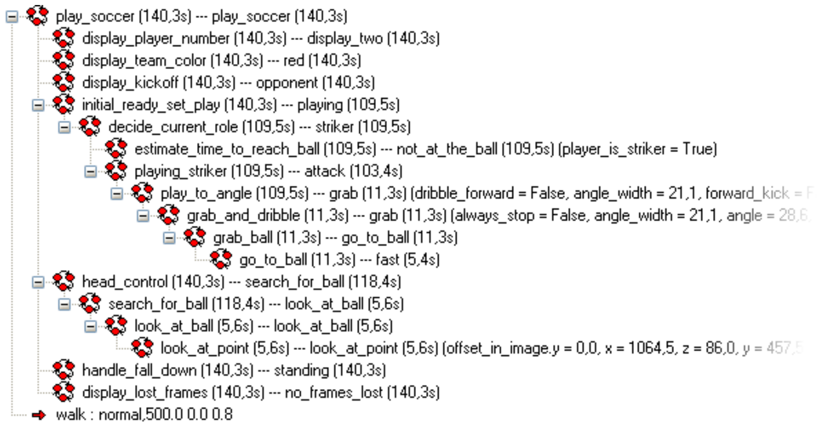

a)
SimRobot - Replay
File Edit View Simulation Window Help
Sensor - RoboCup.views.LOG.modules.cognition.ungrouped
figurationDataProvider
LogDataProvider
default
ColorTable64
Image
SensorData
MotionRequest
Sensor - RoboCup.views.LOG.modules.motion.un...
MotionRequest
MotionSelector
b)
SimRobot - Replay
File Edit View Simulation Window Help
Sensor - RoboCup.views.LOG.modules.cognition.ungrouped
ConfigurationDataProvider
LogDataProvider
ColorTable64
Image
SensorData
Sensor - RoboCup.views.LOG.modules.motion.un...
MotionRequest
MotionSelector

```
mr MotionRequest off
Error: no provider for required representation MotionRequest
```

Fig. 5. Changing the module configuration. a) *MotionRequest* is provided by pseudo module *default* in process *Cognition* (left) and transferred to process *Motion* (right). b) *MotionRequest* is not provided anymore in *Cognition*, and therefore it is missing in *Motion*.

3 Behavior Control

As in previous years, the behavior architecture is based on the successful *Extensible Behavior Specification Language*(XABSL) [8][9]. XABSL is a formalism for the pragmatic design of agent behavior through hierarchies of finite state machines. In 2006 a new version of XABSL has been released which is available online [10]. One of the most important new features in XABSL is that a state of one of the state machines can activate multiple other state machines of the hierarchy simultaneously, which are then executed in parallel. This means that the set of active state machines becomes a subtree of the hierarchy of finite state machines. The GermanTeam makes use of this new feature in order to create a very versatile behavior control module. For example the behavior control module now also controls head motions, which was done in a separate complex *HeadControl* module in previous years. State machines defining head motion behaviors are executed in parallel to state machines defining the overall behavior of the robot. The *HeadControl* module in the *Motion* process only performs low level head motion control, i.e. output of head joint angles from simple motion commands. Figure 6 shows an example of the state machines being active in parallel while playing soccer.



```
play_soccer (140,3s) --- play_soccer (140,3s)
  display_player_number (140,3s) --- display_two (140,3s)
  display_team_color (140,3s) --- red (140,3s)
  display_kickoff (140,3s) --- opponent (140,3s)
  initial_ready_set_play (140,3s) --- playing (109,5s)
    decide_current_role (109,5s) --- striker (109,5s)
      estimate_time_to_reach_ball (109,5s) --- not_at_the_ball (109,5s) (player_is_striker = True)
      playing_striker (109,5s) --- attack (103,4s)
        play_to_angle (109,5s) --- grab (11,3s) (dribble_forward = False, angle_width = 21,1, forward_kick = F
          grab_and_dribble (11,3s) --- grab (11,3s) (always_stop = False, angle_width = 21,1, angle = 28,6,
            grab_ball (11,3s) --- go_to_ball (11,3s)
              go_to_ball (11,3s) --- fast (5,4s)
  head_control (140,3s) --- search_for_ball (118,4s)
    search_for_ball (118,4s) --- look_at_ball (5,6s)
      look_at_ball (5,6s) --- look_at_ball (5,6s)
        look_at_point (5,6s) --- look_at_point (5,6s) (offset_in_image.y = 0,0, x = 1064,5, z = 86,0, y = 457,5
  handle_fall_down (140,3s) --- standing (140,3s)
  display_lost_frames (140,3s) --- no_frames_lost (140,3s)
walk : normal,500.0 0.0 0.8
```

Fig. 6. Example of activated state machines, current states, state execution times, and parameters

4 Ground Truth Environment

For efficiently calibrating and testing many parts of the robots' software, precise and reliable reference data is beneficial if not even necessary. For this purpose, a new ground truth environment is currently integrated into the GermanTeam's software architecture. To avoid to proverbially reinvent the wheel, the vision software of the B-Smart [11] Small Size League team is used. In that system, the efficient and precise tracking of objects from a global point of view has already been solved sufficiently (cf. Fig. 7).

The software runs on an external PC inside SimRobot[12], one of the tools used by the GermanTeam for years. It works with an arbitrary number of connected IEEE 1394 cameras. The perceived information is broadcasted via the

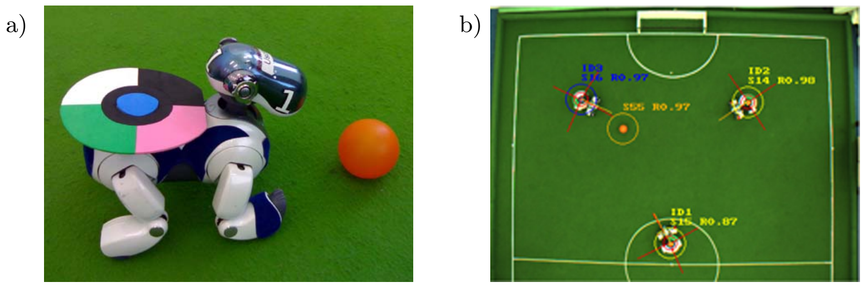


Fig. 7. The ground truth environment: a) A robot equipped with a pattern for identification and rotation detection. b) Camera view on a Small Size field with three AIBO robots and a ball. The drawings indicate the detected objects.

standard robot communication channel and may thus be provided to the robot by a simple module inside the framework.

5 Ball Model

Two different modules are in use to calculate the ball model. A Kalman filter provides the current ball speed and a Rao-Blackwellised particle filter (RBP filter) provides the ball position. The RBP filter is a method first applied in the RoboCup domain by Kwok and Fox [13]. Such a filter maintains a set of particles that represent the posterior over the ball state. Each of them uses a Kalman filter to model the position and speed conditioned on the discrete motion mode of that particle. Two different ball modeling implementations are used because testing during actual games showed that the RBP filter adjusts very well to reliable detections of the ball. Also when the ball disappears behind an obstacle, the ball position is approximated accurately for most occurrences. However, the ball speed calculated by the RBP filter is too erratic to be used to predict the trajectory of the ball. This especially impedes the behavior of the goalkeeper. In contrast, the Kalman Filter provides a continuous approximation of the ball speed, but it is not as flexible as the RBP filter in estimating the ball position, and it needs a successful detection of the ball in an image to update the ball model. Therefore a combination of these Filters proved to be most effective.

6 Conclusion

This team description paper mainly presents the infrastructural changes of the GermanTeam 2007 in comparison to the system of the previous years. Other improvements are documented in other recent publications. A major goal of the new base system is to make the development easier and more flexible. Vast parts of the modules were re-implemented to make the system leaner and faster. A first indication that this approach may have been the right decision was the success at the RoboCup German Open 2007 against the 2006 world champion.

References

1. T. Röfer, "Region-based segmentation with ambiguous color classes and 2-d motion compensation," in *RoboCup 2007: Robot Soccer World Cup XI* (U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, eds.), Lecture Notes in Artificial Intelligence, Springer. to appear.
2. D. Göhring, "Cooperative object localization using line-based percept communication," in *RoboCup 2007: Robot Soccer World Cup XI* (U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, eds.), Lecture Notes in Artificial Intelligence, Springer. to appear.
3. M. Jüngel and M. Risler, "Self-localization using odometry and horizontal bearings to landmarks," in *RoboCup 2007: Robot Soccer World Cup XI* (U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, eds.), Lecture Notes in Artificial Intelligence, Springer. to appear.

4. M. Jüngel, H. Mellmann, and M. Spranger, "Improving vision-based distance measurements using reference objects," in *RoboCup 2007: Robot Soccer World Cup XI* (U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, eds.), Lecture Notes in Artificial Intelligence, Springer. to appear.
5. T. Röfer, R. Brunn, S. Czarnetzki, M. Dassler, M. Hebbel, M. Jüngel, T. Kerkhof, W. Nistico, T. Oberlies, C. Rohde, M. Spranger, and C. Zarges, "GermanTeam 2005," in *RoboCup 2005: Robot Soccer World Cup IX Preproceedings* (A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, eds.), RoboCup Federation, 2005.
6. V. Jagannathan, R. Dodhiawala, and L. Baum, *Blackboard Architectures and Applications.* Academic Press, Inc., 1989.
7. E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Software Practice and Experience*, vol. 30, 2000.
8. M. Lötzsch, M. Risler, and M. Jüngel, "Xabsl - a pragmatic approach to behavior engineering," in *Proceedings of IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, (Beijing, China), pp. 5124–5129, October 9-15 2006.
9. 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* (D. Polani, B. Browning, A. Bonarini, and K. Yoshida, eds.), vol. 3020, Springer, 2004.
10. M. Lötzsch, M. Jüngel, M. Risler, and T. Krause, "XABSL web site," 2006. http://www.ki.informatik.hu-berlin.de/XABSL.
11. A. Burchardt, K. Cierpka, S. Fritsch, N. Göde, K. Huhn, T. Kirilov, B. Lassen, T. Laue, M. Miezal, E. Lyatif, M. Schwarting, A. Seekircher, and R. Stein, "B-Smart - Team Description for RoboCup 2007," in *RoboCup 2007: Robot Soccer World Cup XI Preproceedings* (U. Visser, F. Ribeiro, T. Ohashi, and F. Dellaert, eds.), RoboCup Federation, 2007.
12. T. Laue, K. Spiess, and T. Röfer, "SimRobot - a general physical robot simulator and its application in RoboCup," in *RoboCup 2005: Robot Soccer World Cup IX* (A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, eds.), no. 4020 in Lecture Notes in Artificial Intelligence, pp. 173–183, Springer, 2006.
13. C. Kwok and D. Fox, "Map-based multiple model tracking of a moving object," in *RoboCup 2004: Robot World Cup VIII* (D. Nardi, M. Riedmiller, C. Sammut, and J. Santos-Victor, eds.), no. 3276 in Lecture Notes in Artificial Intelligence, pp. 18–33, Springer, 2005.