2011 IEEE-RAS International Conference on Humanoid Robots
Bled, Slovenia, October 26-28, 2011

Towards an open-source social middleware for humanoid robots

Miguel Sarabia, Raquel Ros, Yiannis Demiris
Department of Electrical and Electronic Engineering
Imperial College London, UK
Email: {miguel.sarabia, raquel.ros, y.demiris}@imperial.ac.uk

***Abstract*—Recent examples of robotics middleware including YARP, ROS, and NaoQi, have greatly enhanced the standardisation, interoperability and rapid development of robotics application software. In this paper, we present our research towards an open source middleware to support the development of social robotic applications. In the core of the ability of a robot to interact socially are algorithms to perceive the actions and intentions of a human user. We attempt to provide a computational layer to standardise these algorithms utilising a bioinspired computational architecture known as HAMMER (Hierarchical Attentive Multiple Models for Execution and Recognition) and demonstrate the deployment of such layer on two different humanoid platforms, the Nao and iCub robots. We use a dance interaction scenario to demonstrate the utility of the framework.**

I. Introduction

It has been stated that the new generation of robots will be human centred [1] as opposed to robots currently used in factories which mechanically repeat what they have been programmed to do. One of the main remaining obstacles lying in the way of human centred robotics is that of understanding the intentions and abilities of the user [2]. This understanding is fundamental for a social robot in order to react and respond accordingly to a human user.

However, despite a wealth of very useful middleware for the control and development of a basic skill set of humanoid robots, to the best of our knowledge there is not any equivalent middleware for the social aspect of the humanoid robot abilities. In this paper we present a middleware that will provide a fast starting point for anyone that requires a robot that recognises the abilities and intentions of its users. Throughout this paper our focus will be in humanoid robots, although there is no reason why the library could not be used for other types of robots.

The main aim of the middleware is to allow flexible deployment, employing hierarchies of inverse-forward model pairs (described in section III-A). HAMMER [3] has been successfully used in many different contexts; for example, to recognise and imitate a human moving an object between two tables [4], to recognise compound and single actions of multiple robots [5] and to predict the intention of opponents in a real-time strategy game [6].

This paper is organised as follows. We start by presenting the design goals of the middleware. Next we introduce a brief overview of the HAMMER architecture and the main characteristics of the framework. Finally we show two preliminary examples of the framework being used to recognise dance steps with the Nao robot and the iCub simulator.

II. HAMMER middleware design goals

Here we delineate the design goals that we set out to fulfil when coding the framework.

a) Generic: We have strived not to make any assumption on the final usage of the middleware . The middleware is very flexible and its operation can be fine-tuned.

b) Adaptable: Right from the start HAMMER was conceived to work with other popular environments within the robotics community. This is what influenced our choice of programming language, C++, as it seems to be the community's language of choice. Because HAMMER may be needed to work in a variety of environments it has only one dependency, the Boost libraries.

c) Interoperable: This is closely related to the previous goal. As our framework is adaptable it could easily interoperate with other robotic middlewares such as ROS [7], Player [8], YARP [9], Urbi, NaoQi, OpenNI, CARMEN [10], DAC [11] and knowledge databases like the one proposed in the RoboEarth project [12]. For a survey on the merits and challenges of robotic middlewares refer to [13]. HAMMER could also work with robot hardware abstractions layers, such as the one created for the CHRIS project [14]; nevertheless, it does not have to use any such abstraction if not needed. Briefly, it is fundamental that HAMMER can be integrated with any robotic framework without any major issues.

d) Ease-of-use: When it did not conflict with the need for generality we aimed to make the middleware as simple as possible to use. For example, this is our motivation behind the pervasive use of shared pointers which deallocate memory automatically without any user intervention. Every attempt was made to keep the middleware consistent. As such, once the basic mindset of HAMMER has been learnt, using the framework should prove a simple task.

e) Multi-threaded: From the very beginning HAMMER was meant to be a concurrent system where all the inverse-forward pairs ran in parallel. The middleware provides support for this without any user mediation, all registered inverse models are executed concurrently, unless there is a hierarchical dependency between them (cf. section III-C). In any case, users must be careful about the use of shared structures in

inverse and forward models as HAMMER only protects its own structures. Consequently any shared variable between different models should be adequately insulated, lest it be accessed or written by two threads at the same time.

f) Open: The HAMMER social middleware is open source and publicly available.

III. HAMMER Middleware Implementation

In this section we describe the theory behind the HAMMER middleware and its main features. Figure 2 shows how the HAMMER middleware fits in with the rest of the robotics ecosystem.

A. Underlying theoretical structures

The field of cognitive robotics was drastically changed with the discovery of the mirror neuron system in primates (for an overview see [15]).The primate mirror neuron system is hypothesised to use the same structures for executing actions as well as recognising them when executed by others. Subsequently theories were developed stating that humans use internal inverse and forward models for motor control [16]. HAMMER is inspired by both these concepts.

An inverse model is a function which takes the state of the world as input, and an optional explicit target state. It outputs the action signals to reach the target state, which could be implicitly hard coded in the model or explicitly passed as an input parameter.

A forward model is defined as a function that takes an action signal and outputs the predicted state of the world after the signal has been executed. The term forward model has been used in the literature to represent many different concepts. For us it is an output predictor, following the analysis in [17].

Pairing together an inverse model and a forward model, we obtain a system that generates a hypothesis about the next state of the world. By combining several inverse-forward pairs, which are normally run in parallel, numerous hypotheses are proposed. These hypotheses are compared against the actual state of the world at the next time-step. A confidence value is computed for every inverse-forward pair, representing the confidence on that inverse-forward pair being the behaviour that is being currently executed. By repeating this process iteratively, confidences for different behaviours can be observed over time. This is HAMMER's basic mode of operation, as shown in figure 1.

Until now we have used 'state' in a loose manner. 'State' is a set of variables both external (environmental) and internal (proprioceptive) to the robot. From now on, we will refer to these variables as *aspects* to differentiate from other code variables. Aspects contain all the information needed to recognise and execute the behaviour contained in an inverse-forward pair. 'State' can represent the actual state of the world around the robot, or the state around a demonstrator. If we take the latter approach, the robot can effectively recognise the actions of a demonstrator, as long as it has the inverse-forward pairs to represent that action. In other words, by generating the state

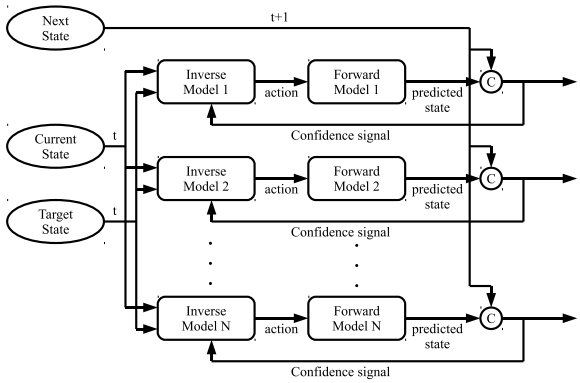


Fig. 1. Diagrammatic representation of HAMMER. Each pair of inverse and forward model represents a hypothesis about the next state, which is then evaluated against the actual state to generate a confidence value.

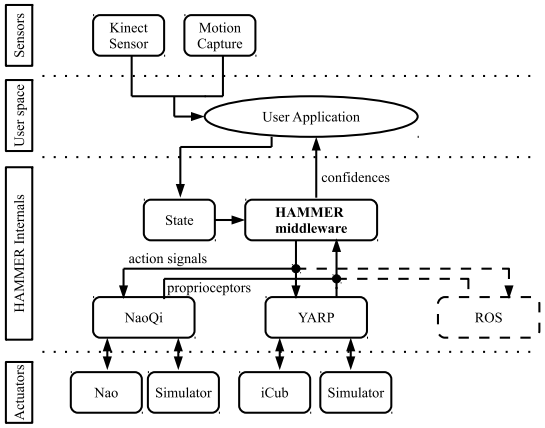


Fig. 2. Data-flow between HAMMER, other major robotics components and the user application. Dashed connections refer to forthcoming features of the middleware.

from the observations of a demonstrator the robot can place itself on the "demonstrator's shoes" [2]–[4].

HAMMER is also able to distribute the robot resources, or attention, according to the needs of the inverse-forward pairs by making the saliency of their resource requests a function of their confidence. However, this issue goes beyond the scope of this paper and we will not explore it further. The reader is referred to [2], [3] for a description of this functionality.

B. Framework Description

In order to create a HAMMER-based architecture the following components need to be specified.

- State of the world, for every time step.
- Inverse Models
- Forward Models
- Action Signals, which will be used to send commands between the inverse and the forward models
- Confidence Evaluation Function

The HAMMER middleware provides structures to aid in the design of the above components, while at the same time making as few assumptions as possible about the intended end application of the architecture.

We define the `State` class as a polymorphic container. Each variable in the container is designated as an `aspect` of the world state. Aspects may be of any C++ type and must be labelled with a string, which is used for storage and retrieval. The `State` class is thus akin to a polymorphic dictionary.

The class `Signals` is similar to the `State` class. It shares the same interface of a polymorphic dictionary. Yet, its semantics within the context of HAMMER are very different. `Signals` provide the means for inverse models to send action commands to the forward models. Alternatively, they could be sent to the robot for execution of the commands contained within. Hence it is desirable, but not actually enforced, that any `Signals` instance contains low-level robot commands.

Our middleware offers two ways to create inverse and forward models. For ordinary memoryless functions, a simple function definition with some predefined arguments will suffice. For more complex inverse and forward models it is recommended to create a class which inherits `InverseModel` or `ForwardModel` as needed. In any case, creating an inverse model essentially consists of defining a function that takes as arguments the current `State` and the –possibly null– target `State` and returns a `Signals` object (see figure 3 for an example). Similarly, all forward models require a function that takes the `Signals` generated by the inverse model and returns a new predicted `State`.

One important concept in our framework is that of subscriptions. Every inverse-forward pair subscribes to a subset of the `aspects` of the world state. Users must decide which aspects of the world state are needed for the operation of inverse models and subscribe to them. HAMMER will then create a `State` containing the appropriate subset of the `aspects` and will give that as input to the inverse model. At the same time, subscription to an aspect of the state entails the commitment to predict the value of that aspect at the next time-step. That is, the `State` generated by the forward model must contain all aspects to which the inverse model is subscribed to.

HAMMER requires an evaluation step to classify the performance of the competing inverse-forward pairs. The internals of the confidence function are left up to the user to decide. The only requirement on the confidence function is to return a `double` representing the change in confidence. HAMMER provides a set of aspects with their predicted and actual values as input for this type of function. In section IV-B we show an example of how the confidence function might be defined.

All interactions between the different components explained above are controlled by an instance of `Core`. This class orchestrates all the structures above so that they work following the principles of HAMMER. From the user point of view, `Core` has a few crucial functions. All inverse and forward models must be registered with it. Additionally, it must also be fed the new `State` at every time-step, note that it is possible to wait until all inverse-forward pairs have finished processing all States added to Core. Moreover `Core` can be used to read the current confidences of the inverse-forward pairs.

```
//Bring HAMMER structures into scope
using namespace HAMMER;

Signals::KPtr inverseModelFunction(
      const State::KPtr& current, const State::KPtr& target){
    //Read data from state and target
    int currentValue = current->get<int>("aspectName");
    int targetValue = target->get<int>("aspectName");

    //Compute robot command based on the target and current values
    int command = targetValue - currentValue;

    //Create signals and put command(s) there
    Signals::Ptr result = Signals::make();
    result->put("robotCommand", command);
    return result;
}
int main (void){
    //Declare inverse model subcriptions
    StringVector subscriptions;
    subcriptions.push_back("aspectName");
    //Create an inverse model which executes inverseModelFunction
    InverseModel::Ptr inverseModel = SimpleInverseModel::make(
        "nameOfInverseModel",
        subscriptions,
        &inverseModelFunction
    );

    //Rest of the functionality here...
}
```

Fig. 3. C++ example of a `SimpleInverseModel` instantiation.

C. Hierarchical models

Our framework has the ability to create hierarchies of inverse-forward pairs as we now explain. A similar mechanism to that of aspect subscriptions is used to manage hierarchies. An inverse model just needs to declare the list of lower level inverse-forward pairs that it wants to follow (known in the framework as dependencies). The middleware will then provide the confidence value at the previous time-step of those pairs. Cyclic dependencies between inverse models are avoided since all dependencies are resolved at registration time; therefore if an inverse model requests information about another inverse module which has not yet been registered, `Core` will raise an error. Users must be aware that hierarchies necessarily reduce concurrency as inverse models with dependencies cannot be executed at the same time as those without dependencies or less stringent ones.

IV. Dancing Humanoids

We have presented a generic middleware for action prediction and recognition. In this section we present an application of the HAMMER middleware being used in a social context, i.e., dancing. The robot must be able to understand the actions of the user and act accordingly; in the following examples by imitating the user, but more complex plans could well be devised. Note the following examples are a proof of concept rather than a full working system.

The task at hand is to recognise the arm movements of the well known latin song "Macarena". Our system understands ten arm positions, five for each arm. The poses recognised are: *resting*, *extended*, *crossed*, *to-head* and *to-hip* (see Figure 4a for a visualisation of these poses). We use this architecture with Aldebaran's Nao –a small humanoid robot with 25

degrees of freedom– and the iCub simulator –the iCub is a child-like humanoid with 53 degrees of freedom [18].

A. State Representation and Acquisition

To avoid the correspondence problem (that is, the translation of actions across dissimilar embodiments [19]) we use an abstract state representation that can be easily calculated for humans, the Nao and the iCub. This representation consists of six angles, three for each arm; namely the angle between the arm and the shoulder, the angle between the arm and the hip, and the elbow angle. This state representation clearly does not cover the whole space of arm positions in a human or a robot, however it does suffice to distinguish the end positions previously listed.

In order to generate a `State` variable at each time-step, the Kinect motion sensor in conjunction with OpenNI[1] and NITE[2] are used. This allows us to easily obtain a 15 point skeleton and apply a few geometric transformations to obtain the state aspects, ie. the angles described in the previous paragraph. Once created, the `State` is fed into HAMMER where the inverse and forward models will be executed and hypotheses put forward (in the form of a prediction about the next state). These inverse and forward models behave differently for the Nao and the iCub simulator and will be described in latter sections. First we describe the confidence evaluation function, common to both platforms.

B. Confidence Evaluation Function

The confidence evaluation function is crucial for HAMMER as it determines which aspects of the world state are more relevant in order to detect a given behaviour. HAMMER does not place any restrictions on the form of this function. For convenience, we introduced two intermediate values used to calculate the change in confidence, the error and the negative reward. The world state is composed solely by angles measurements. Thus the error function needs to be defined only for angles:

$$e_a = |s_a - p_a| \bmod 2\pi \tag{1}$$

where $a \in 1 \dots N$ is one of the aspects of the world state, N is the total number of world state aspects, s_a is the sensed value of a and p_a its corresponding predicted value.

Before introducing the confidence function, it is necessary to define the negative reward $r \in 0 \dots 1$, which represents how far from the actual aspect the predicted aspect was.

$$r_a = \begin{cases} \dfrac{e_a}{\theta_a} & \text{if } e_a \leq \theta_a \quad (2a) \\ 1 & \text{if } e_a > \theta_a \quad (2b) \end{cases}$$

where θ_a are the aspect-specific thresholds, above which the negative reward will be maximal. These thresholds were fine-tuned by experimentation. Different values for different aspects be must allowed as the precision of the Kinect sensor varies for different parts of the body. For reference we used $\theta_a = \pi/2$ rad when a represented any of the elbow angles, and $\theta_a = 0.6$ rad for all the other aspects.

Next we define the confidence as,

$$\Delta c = \sum_{a=1}^{N} (1 - r_a) = N - \sum_{a=1}^{N} r_a \tag{3}$$

Δc is the total number of aspects minus the sum of the negative rewards. Note that in this particular instance of HAMMER, $0 \leq \Delta c \leq 1$ which means that confidences can only go up. This will have some repercussions that will be explained later.

For clarity the above formulae did not include sub-indices for a given inverse-forward pair nor for the corresponding time-step. These calculations must be repeated for every inverse-forward pair at every time-step. Equation 4 shows how the delta of the confidence is applied at a given time-step t for a certain inverse-forward pair i to yield the new confidence at time $t + 1$.

$$c_{t+1,i} = c_{t,i} + \Delta c_{t,i} \tag{4}$$

The confidences on this system are always rising. This means that at some point all confidences must be reset to allow for new behaviours to be detected. Ideally the system would reset every time the confidence of a behaviour is above a certain value. This does not work well in practice, mainly because the sensors are noisy and some postures –such as arms crossed– tend to be detected with a relatively low confidence. It is the case however that when the user is doing an action the corresponding behaviour tends to receive a higher Δc than other competing behaviours. With this in mind, we found out that periodically taking the inverse-forward pairs with the highest confidence value as the actions executed by the user and then resetting all the confidences produces good detection results. For our experiments we used a period of 15 frames, which is equivalent to 0.5 seconds[3]. This interval is sufficiently small to detect all movements by the user, yet large enough to average out any sensing error.

C. Nao Inverse and Forward Models

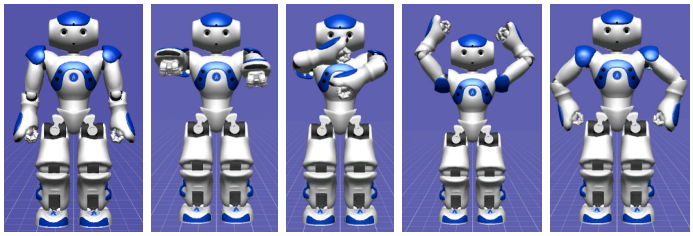
Even though most of the architecture is shared between the iCub simulator and the Nao examples, the inverse models are necessarily different. This is because the motor instructions of the iCub and the Nao are different and so are the effects of those signals in the environment and the robots themselves. We proceed to describe our approach to code the inverse and forward models for the Nao.

We define an inverse model for every arm position to be detected. For this particular instantiation of HAMMER, every inverse model sends the end effector position to the forward model, independently of the current world state. This has the disadvantage that a behaviour will not be predicted until it has been finished by the user. Yet this approach yielded good results for our set-up. The output `Signals` contained

[1]OpenNI is a set of open source programmer interfaces for natural interaction devices such as the Kinect sensor.

[2]NITE is a middleware built on top of OpenNI to generate skeleton and track users.

[3]The Kinect sensor runs at 30 frames per second



(a) Recognised poses shown in the Nao robot. From left to right: *resting*, *extended*, *crossed*, *to-head*, *to-hip*.

(b) Nao imitating the movements of a user which were recognised through HAMMER

Fig. 4. Dancing Nao.

the motor commands required to reach those positions, which were obtained using kinaesthetics.

For the forward models we first simulated the effects of the motor signals on the Nao simulator and used those results as the basis to generate the predicted state. Despite being functional, this approach had the major disadvantage of not being able to run in real-time due to the execution time required by the simulator. Instead we assumed an ideal robot, where motor commands would be executed with perfect fidelity and instantaneously. In order to generate the predicted `State`, the three-dimensional end positions of the elbow and hand joints were calculated using standard 3D rotations. These end positions were, in turn, used to compute the angular aspects of which the world state is composed, via a straightforward geometric transformation.

Since the inverse-forward pairs only have their confidence increased when the user's position matches their predictions, it was necessary that Δc was not negative. Otherwise, an inverse model might have been punished for predicting a correct behaviour which had not yet finished. This restricts the system to detection of movements only after they have been completed, and not whilst being performed. Due to the assumptions made, the current set-up is somewhat limited on the scope of actions that can be recognised. Still, it does show how the HAMMER middleware can be used to detect human behaviours which is precisely the point of this proof-of-concept. Figure 4b shows the Nao repeating the movements of a user.

Finite State Machine for Full Dance Detection: By taking advantage of the already described dependencies feature (section III-C), we were able to build a hierarchy that would detect the full "Macarena" dance using a new inverse model with dependencies to every inverse-forward pair introduced earlier. This inverse model kept track of the total confidence of every inverse-forward pair and detected which ones were being executed by the user, using the same interval based approach presented above. A Finite State Machine, whose states were the current position of the left and right arms, was coded and incorporated into the high-level inverse model. The only valid Finite State Machine transitions were those of the dance. This technique allows the robot to understand when has the user performed the full choreography and not simply a few steps.

D. iCub Simulator Inverse and Forward Models

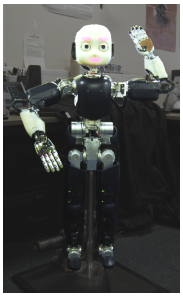
Despite using a different robot, the dancing iCub Simulator shares most of the architecture with the dancing Nao. `State` representation and acquisition –using the Kinect sensor – and confidence evaluation have all virtually the same code as the Nao example. The main distinction between the two examples lies in the inverse and forward models. For the iCub Simulator the aim was to provide structures to design such models in a generic fashion. This was achieved through the creation of instructions and `Interpreter`.

`Instructions` are structures that group together all motor commands to the robot plus some switches to change a few aspects of its operation –like setting the motor speeds or the motor angles. There are two types of instructions: `PositionInstruction` and `VelocityInstruction`. Instructions are intended to be wrapped inside `Signals` as the output of an inverse model.

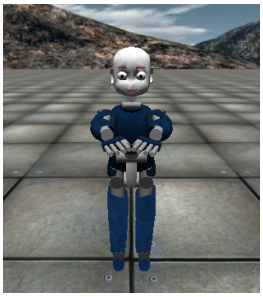
An `Interpreter` is the class that holds the YARP connections and is therefore able to execute instructions. Upon instantiation the interpreter will create connections to the YARP control ports of the iCub (for an explanation of YARP refer to [9]). It has methods to execute both `PositionInstruction` and `VelocityInstruction`. Moreover `Interpreter` can read the motor encoder positions, speeds and accelerations. `Interpreter` is meant to be embedded into a forward model for easy execution of instructions and retrieval of motor values.

Ten inverse models were created for this example. Each one sends an instruction with the hard-coded values for the behaviour the inverse model represents. All inverse models are paired with the same forward model, which executed the instruction in the iCub simulator, waited for four seconds, read the values of the motor encoders and converted them to the relevant aspects of the world state.

To avoid interference from one inverse model on the results of another, we had to protect the execution and reading steps of the inverse model using mutexes. This necessarily means that concurrency was diminished. This effect was palliated by allowing inverse models with non-competing aspect subscriptions to run simultaneously in the forward model. In other words, behaviours representing the right arm ran alongside behaviours representing the left arm. Because the forward model waits for the simulator, this example cannot be executed



(a) iCub executing two inverse models



(b) iCub Simulator rehearsing two hypotheses

Fig. 5. HAMMER on the iCub and the iCub simulator. Note that execution on actual robot is preliminary work.

in real-time as it was the case with the Nao.

E. Discussion

Section IV serves as an example of the HAMMER middleware being used in the context of a social interaction. In order to demonstrate the functionality of the architecture we developed basic inverse models, forward models that connected to different robot simulators and a confidence evaluation function. Qualitatively speaking both systems could detect the behaviours of the user without any major errors. There were a few cases of behaviour aliasing, where a behaviour would be confused for another, but they mostly happened on transitions between poses where the noise of the sensors is critical. Further tuning of the parameters would bring these few misclassifications to zero.

V. Further Work

The main functionality of the HAMMER social middleware is established and we have demonstrated its use in a dancing scenario in two different humanoid platforms. Efforts are underway to integrate HAMMER with other robotic middleware platforms, including ROS (Robot Operating System), and increase compatibility with other programming languages like MATLAB and Python.

VI. Conclusion

In this article we have described the HAMMER middleware for social robots which is freely available online and interoperable with most robotics application-stacks. The framework allows the user to instantiate its own HAMMER architecture for behaviour recognition. We have shown examples of two such instantiations, in the context dancing humanoids. We believe these two examples prove the ease of use of the HAMMER middleware. They further show that the framework allows for a great level of code re-usability as most of the code was shared between the two examples. It is also remarkable how little effort was required to integrate the middleware with NaoQi or YARP – in particular, porting the already existing code for the Nao to the iCub was very straightforward.

We hope that by providing a generic and flexible middleware that adapts to the current robotics ecosystem, social robotic applications could be more rapidly bootstrapped.

Acknowledgement

The authors would like to thank Yan Wu and the rest of the BioART Lab. This work has been partially funded by the EU FP7 ALIZ-E project (no. 248116) and the EU FP7 EFAA project (no. 270490).

References

[1] S. Schaal, "The New Robotics-towards human-centered machines." HFSP journal, vol. 1, no. 2, pp. 115–26, 2007.

[2] Y. Demiris, "Prediction of intent in robotics and multi-agent systems." Cognitive processing, vol. 8, no. 3, pp. 151–158, 2007.

[3] Y. Demiris and B. Khadhouri, "Hierarchical attentive multiple models for execution and recognition of actions," Robotics and Autonomous Systems, vol. 54, no. 5, pp. 361–369, 2006.

[4] "Abstraction in Recognition to Solve the Correspondence Problem for Robot Imitation," in Proc. of the Conf. Towards Autonomous Robotics Systems, 2004, pp. 63–70.

[5] M. F. Martins and Y. Demiris, "Learning multirobot joint action plans from simultaneous task execution demonstrations," in Proc. of the Intl. Conf. on Autonomous Agents and Multiagent Systems, vol. 1, 2010, pp. 931–938.

[6] S. Butler and Y. Demiris, "Partial Observability During Predictions of the Opponent's Movements in an RTS Game," in Proc. of the Conf. on Computational Intelligence and Games, 2010, pp. 46–53.

[7] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, ROS: an open-source Robot Operating System. IEEE, 2009.

[8] B. P. Gerkey, R. T. Vaughan, K. Stø y, A. Howard, G. S. Sukhatme, and M. J. Matarić, "Most valuable player: a robot device server for distributed control," in Proc. of the Intl. Conf. on Intelligent Robots and Systems, 2001, pp. 1226–1231.

[9] "YARP : Yet Another Robot Platform," Advanced Robotics, vol. 3, no. 1, pp. 43–48, 2006.

[10] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit," in Proc. of the Intl. Conf. on Intelligent Robots and Systems, vol. 3, 2003, pp. 2436–2441.

[11] A. Duff, C. Renn-Costa, E. Marcos, A. Luvizotto, A. Giovannucci, M. Sanchez-Fibla, U. Bernardet, and P. Verschure, "Distributed adaptive control: A proposal on the neuronal organization of adaptive goal oriented behavior," in From Motor Learning to Interaction Learning in Robots, 2010, vol. 264, pp. 15–41.

[12] M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Gálvez-López, K. Häussermann, R. Janssen, J. Montiel, A. Perzylo, B. Schiessle, M. Tenorth, O. Zweigle, and R. van de Molengraft, "RoboEarth," IEEE Robotics & Automation Magazine, vol. 18, no. 2, pp. 69–82, 2011.

[13] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in Proc. of the Conf. on Robotics, Automation and Mechatronics, 2008, pp. 736–742.

[14] S. Lallée, S. Lemaignan, A. Lenz, C. Melhuish, L. Natale, S. Skachek, T. van Der Zant, F. Warneken, and P. Dominey, "Towards a platform-independent cooperative human-robot interaction system: I. Perception," in Proc. of the Intl. Conf. on Intelligent Robots and Systems, 2010, pp. 4444–4451.

[15] G. Rizzolatti and L. Craighero, "The mirror-neuron system." Annual review of neuroscience, vol. 27, pp. 169–92, 2004.

[16] D. Wolpert, R. C. Miall, and M. Kawato, "Internal models in the cerebellum," Trends in cognitive sciences, vol. 2, no. 9, pp. 338–47, 1998.

[17] A. Karniel, "Three creatures named 'forward model'." Neural Networks, vol. 15, no. 3, pp. 305–7, 2002.

[18] G. Metta, G. Sandini, D. Vernon, L. Natale, and F. Nori, "The iCub humanoid robot: an open platform for research in embodied cognition," in Proc. of the Workshop on Performance Metrics for Intelligent Systems, 2008, pp. 50–56.

[19] A. Alissandrakis, C. L. Nehaniv, and K. Dautenhahn, "Imitation with ALICE: learning to imitate corresponding actions across dissimilar embodiments," IEEE Transactions on Systems, Man, and Cybernetics, vol. 32, no. 4, pp. 482–496, 2002.