Plan-Based Configuration of a Group of Robots

Robert Lundh and **Lars Karlsson** and **Alessandro Saffiotti** [1]

Abstract. We consider groups of autonomous robots in which robots can help each other by offering information-producing functionalities. A functional *configuration* is a way to allocate and connect functionalities among robots. In general, different configurations can be used to solve the same task, depending on the current situation. In this paper, we define the idea of functional configuration, and we propose a plan-based approach to automatically generate a preferred configuration for a given task, environment, and set of resources. To illustrate these ideas, we show a simple experiment in which two robots mutually help each-other to cross a door.

1 Introduction

Consider the situation shown in Figure 1, in which a mobile robot, named Pippi, has the task to push a box across a door. In order to perform this task, Pippi needs to know the position and orientation of the door relative to itself at every time during execution. It can do so by using its sensors, e.g., a camera, to detect the edges of the door and measure their distance and bearing. While pushing the box, however, the camera may be covered by the box. Pippi can still rely on the previously observed position, and update this position using odometry while it moves. Unfortunately, odometry will be especially unreliable during the push operation due to slippage of the wheels. There is, however, another solution: a second robot, called Emil, could observe the scene from an external point of view in order to compute the position of the door relative to Pippi, and communicate this information to Pippi.

The above scenario illustrates a simple instance of the general approach that we suggest in this paper: to let robots help each other by borrowing functionalities from one another. In the above example, Pippi needs a functionality to measure the relative position and orientation of the door in order to perform its task: it has the options to either compute this information using its own sensors, or to borrow this functionality from Emil.

More generally, we consider a society of autonomous robotic systems embedded in a common environment [5]. Each robot in the society includes a number of *functionalities*: for instance, functionalities for image understanding, localization, planning, and so on. We do not assume that the robots are homogeneous: they may have different sensing, acting, and reasoning capacities, and some of them may be as simple as a fixed camera monitoring the environment. The key point here is that each robot may also use functionalities from other robots in order to compensate for the ones that it is lacking, or to improve its own. In the situation shown in Figure 1, Pippi borrows from Emil a perceptual functionality for measuring the relative position between the door and itself.

[1] Center for Applied Autonomous Sensor Systems, Örebro University, Sweden. email: {robert.lundh, lars.karlsson, alessandro.saffiotti}@aass.oru.se

We informally call *configuration* any way to allocate and connect the functionalities of a distributed multi-robot system. Often, the same task can be performed by using different configurations. For example, in our scenario, Pippi can perform its door-crossing task by connecting its own door-crossing functionality to either (1) its own perception functionality, (2) a perception functionality borrowed from Emil, or (3) a perception functionality borrowed from a camera placed over the door. Having the possibility to use different configurations to perform the same task opens the way to improve the flexibility, reliability, and adaptivity of a society of robotic agents. Ideally, we would like to automatically select, at any given moment, the best available configuration, and to change it when the situation has changed.

In this context, our overall research objective is threefold:

1. To formally define the concept of functional *configuration* of a robot society.
2. To study how to *automatically generate* a configuration of a robot society for a given task, environment, and set of resources.
3. To study when and how to *change* this configuration in response to changes in the environment, task, or resources.

In this paper, we focus on the first two objectives above: the third objective will be the subject of future work. More specifically, we define a concept of configuration which is adequate from the purpose of automatically reasoning about configurations, and show how to use AI planning techniques to generate a configuration that solves a given task. We also describe an experimental system where these ideas are

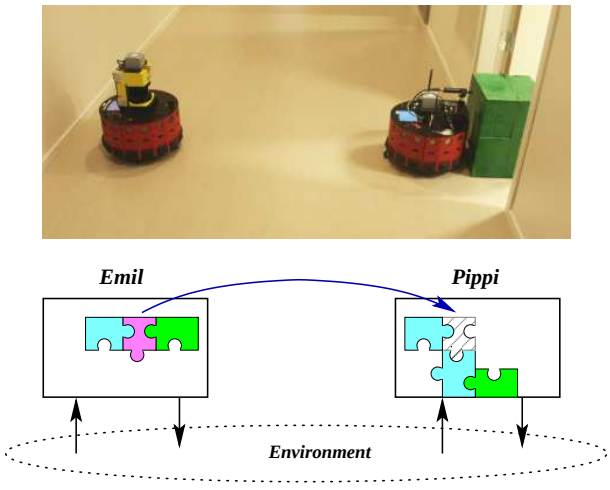

Figure 1. Top: Emil helps Pippi to push the box through the door by measuring their relative position. Bottom: the corresponding configuration, in which Emil is providing a missing perceptual functionality to Pippi.

Proc of the 17th European Conference on Artificial Intelligence (ECAI)
Riva del Garda, Italy, August 2006

implemented, and show an example of it in which two iRobot Magellan Pro robots mutually help each other to cross a door.

2 Functional configurations

The first goal in our research program is to develop a definition of configuration that is adequate for the three objectives presented in the introduction. In general, a configuration of a team of robots may include interconnected functionalities of two types: functionalities that change the internal state by providing or processing information, and functionalities that change the state of the environment. (Some functionalities can have both aspects.)

To define our notion of configurations, a clarification of the three concepts of functionality, resource and channel is in order.

2.1 Preliminaries

We assume that the world can be in a number of different states. The set of all potential world states is denoted S. There is a number of robots $r_1, \ldots, r_n$. The properties of the robots, such as what sensors they are equipped with and their current positions, are considered to be part of the current world state s_0. There are also a number of communication media CM, such as radio, network, internal message queues, which can be used to transfer information between and within robots. A medium may have restrictions on band width.

2.2 Functionality

A *functionality* is an operator that uses information (provided by other functionalities) to produce additional information. A functionality is denoted by

$$f = \langle r, Id, I, O, \Phi, Pr, Po, Freq, Cost \rangle$$

Each instance of a functionality is located at a specific robot or other type of agent r and has a specific identifier Id.[2] The remaining fields of the functionality tuple represents:

- $I = \{i_1, i_2, \ldots, i_n\}$ is a specification of inputs, where $i_k = \langle descr, dom \rangle$. The descriptor ($descr$) gives the state variable of the input data, and the domain (dom) specifies to which set the data must belongs.
- $O = \{o_1, o_2, \ldots, o_m\}$ is a specification of outputs, where $o_k = \langle descr, dom \rangle$. The descriptor ($descr$) gives the state variable of the output data, and the domain (dom) specifies to which set the data must belongs.
- $\Phi : dom(I) \rightarrow dom(O)$ specifies the relations between inputs and outputs.
- $Pr : S \rightarrow \{T, F\}$ specifies the causal preconditions of the functionality. Pr specifies in what states $s \in S$ the functionality can be used.
- $Po : S \times dom(I) \rightarrow S$ specifies the causal postconditions. It is a function that, given the input to the functionality, transforms the world state s before the functionality was executed into the world state s' after the functionality has been executed.
- $Freq$ specifies how often the functionality is to be executed.
- $Cost$ specifies how expensive the functionality is, e.g. in terms of time, processor utilization, energy, etc.

[2] When referring to a specific element in a functionality or other entity represented as a tuple, we will be using a functional notation, e.g. $r(f)$ is the r field in the tuple of f.

A typical functionality could be the measure door operation mentioned in the introductory example. This functionality takes an image from a camera as an input and measures the position and orientation of a door in the image. To produce the output, the position and the orientation of the door, this functionality has a precondition that needs to be satisfied. The precondition is that the door must be visible in the (input) image.

2.3 Resource

A *resource* is a special case of a functionality. There are two different types of resources: sensing resources and action resources. A *sensing resource* has $I = \emptyset$, i.e., no input from other functionalities, and is typically a sensor that gives information about the current state of the surrounding environment or the physical state of the robot. An example of a sensing resource is a camera, which produces images as output as long as the preconditions (e.g. camera is on) are fulfilled.

An *action resource* has $O = \emptyset$ (i.e., gives no output to other functionalities) and Po is not the identity function. An action resource is typically some actuator (e.g., a manipulator).

2.4 Channel

A *channel* $Ch = \langle f_{send}, o, f_{rec}, i, medium \rangle$ transfers data from an output o of a functionality f_{send} to an input i of another functionality f_{rec}. It can be on different media.

2.5 Configuration

A *configuration* C is tuple $\langle F, Ch \rangle$, where F is a set of functionalities and Ch is a set of channels that connect functionalities to each other. Each channel connects the output of one functionality to the input of another functionality.

In the context of a specific world state s, a configuration is *admissible* if the following conditions are satisfied:

$$\begin{array}{l} \forall f \in F \, \forall i \in I_f \exists ch \in Ch = (f_{send}, o, f_{rec}, i, m) \text{ such that} \\ \quad descr(o) = descr(i), dom(o) = dom(i), \text{ and} \\ \quad Freq(f_{send}(ch)) \geq Freq(f_{rec}(ch)) \end{array}$$

That is, each input of each functionality is connected via an adequate channel to an output of another functionality with a compatible specification (information admissibility).

$$\forall f \in F : Pr_f(s) = T$$

That is, all preconditions of all functionalities hold in the current world state (causal admissibility).

$$\begin{array}{l} \forall m \in CM : bandwidth(m) \geq \\ \quad \sum_{\{ch | medium(ch) = m\}} size(domain(i(ch))) \cdot Freq(f_{rec}(ch)) \end{array}$$

That is the combined requirements of the channels can be satisfied (communication admissibility).

Another issue is schedulability: whether the different functionalities can produce and communicate their outputs in a timely manner. However, that is a complex issue that cannot be detailed in the scope of this paper.

A configuration also has a cost. This can be based on functionality costs, communication cost, etc, but also on performance accuracy and reliability of the configuration. Currently, we compute the cost as a weighted sum of the number of components used (robots involved, functionalities, global and local channels).

2.6 Examples

To illustrate the above concepts, we consider a concrete example inspired by the scenario in the introduction. A robot is assigned the task of pushing a box from one room to another one by crossing a door between the two rooms. The "cross-door" action requires information about position and orientation of the door with respect to the robot performing the action. There are two indoor robots (including the one crossing the door) each one equipped with a camera and a compass. The door to cross is equipped with a wide-angle camera.

Fig. 2 illustrates four different (admissible) configurations that provide the information required by the action "cross-door", which include the functionalities above.

The first configuration involves only the robot performing the action. The robot is equipped with an elevated panoramic camera that makes it possible to view the door even when pushing the box. The camera produces information to a functionality that measures the position and orientation of the door relative to the robot.

The second configuration in Fig. 2 shows the other extreme, when all information is provided by the door that the robot is crossing and the robot is not contributing with any information. The door is equipped with a camera and functionalities that can measure the position and orientation of the robot relative to the door. This information is transformed into position and orientation of door with respect to the robot before it is delivered to robot A.

In the third configuration in Fig. 2, robot A (the robot performing the "cross-door" action) contributes with a compass, and the second robot (B) contributes with a compass and a camera. The camera provides information to two functionalities: one that measures the position and orientation of the door, and another one that measures the position of robot A. All these measurements are computed relative to robot B. In order to compute the position and orientation of the door relative to robot A, we need to use a coordinate transformation. This in turn requires that we know, not only the relative position, but also the orientation of robot A relative to B. The later can be obtained by comparing the absolute orientations of the two robots, measured by their two on-board compasses.

The fourth configuration in Figure 2 is similar to the third one, except that the orientation of robot A relative to B is obtained in another way, i.e., no compasses are used. Both robots are equipped with cameras and have a functionality that can measure the bearing to an object. When the robots look at each other, each robot can measure the bearing to the other one. By comparing these two measurements, we obtain the orientation of robot A relative to robot B.

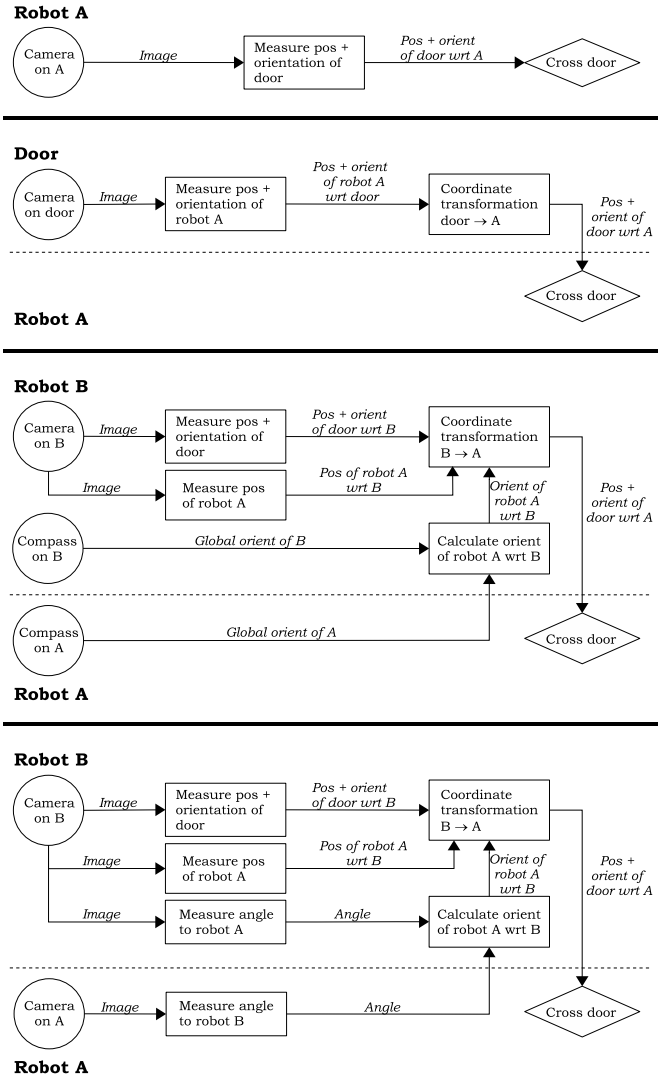

Figure 2. Four different configurations that provide the position and orientation of a given door with respect to robot A. (See text.)

3 Configuration generation

Our second objective is to generate configurations automatically. This process requires a declarative description of:

- the functionalities and methods for connecting them,
- the current world state,
- the goal for what the configuration should produce.

The world state, as already mentioned, declares the available robots and their physical capabilities, as well as the state of the surrounding environment. It is encoded as a set of clauses, such as `robot(r1), door(d1), visible(r1, door1)`. The goal for the configuration generation process is to produce a desired information output. For example, in the cross-door example, the information goal is the position and orientation of the door that is required as input by the cross-door behavior.

The description of available functionalities is realized using operator schemas similar to those of AI action planners. One functionality operator schema from the scenario in the previous section, `measure-door`, is shown in the upper half of Figure 3.

The `name` field correspond to the Id component of a functionality, and the `r` parameter in the name is r in the functionality. The fields `input` and `output` correspond to I and O, and `precond` and `postcond` encode the Pr and Po components. In the `measure-door` example we have an image taken by the camera of `r` as input and from that we are able to compute the position and orientation of the door `d` relative to `r` as output. The precondition for `measure-door` is that the door `d` is fully visible in the input image. Note that the Φ function is not part of the operator schema; it corresponds to the actual code for executing the functionality.

In order to combine functionalities to form admissible configurations that solve specific tasks, we have chosen to use techniques inspired by hierarchical planning, in particular the SHOP planner [3]. Configuration planning differs from action planning in that the

```
(functionality
  name:     measure-door(r, d)
  input:    (image(r, d), image)
  output:   (pos(r, d), real-coordinates)
            (orient(r, d), radians)
  precond:  visible(r, d)
  postcond:
)

(config-method
  name: get-door-info(r, d)
  precond: ( camera(r), in(r, room),
             in(d, room), robot(r), door(d))
  in:  -
  out: f2: pos(r, d)
       f2: orient(r, d)
  channels: (f1, f2, image(r, d))
  body:
    f1: camera(r, d)
    f2: measure-door(r, d)
)
```

Figure 3. A functionality operator schema, and a method schema for combining functionalities.

functionalities, unlike the actions in a plan, are not temporally and causally related to each other, but related by the information flow as defined by the channels. Functionalities are executed in parallel, and a functionality becomes active when the input data is present. In order to achieve the data flow between functionalities, a mechanism that creates channels between functionalities is required. Such connections are not required for action planning, and obviously, no existing planning technique facilitate such a mechanism. Therefore we need to outline how the planner works.

Our configuration planner allows us to define methods that describe alternative ways to combine functionalities (or other methods) for specific purposes, e.g. combining the camera functionality and the measure-door functionality above with a channel on the same robot in order to obtain door measurements of a specific door.

The lower half of Figure 3 shows an example of a method schema that does exactly that. There is a channel inside the method connecting two functionalities (labeled `f1` and `f2`). The descriptor of the channel (`image(r, d)`) tells which specific input and output of the functionalities should be connected. In addition, the outputs (`pos(r, d), orient(r, d)`) of `f2` is declared in the `out` field to be the output of the entire method. Thereby, any channel that in a method higher up in the hierarchy is connected to the output of `get-door-info` will be connected to the output of `measure-door`.

The configuration planner takes as input a current causal state s, a method "stack" with initially one unexpanded method instance $l : m(c_1, c_2, ...)$ representing the goal of the robot (l is a label), and a set of methods M and a set of functionality operators O. It also maintains an initially empty configuration description C and an initially empty sets of causal postconditions P. It works as follows:

1. Take the unexpanded method instance $l : m(c_1, c_2, ...)$ at the top of the stack.
2. If $l : m(c_1, c_2, ...)$ matches the name of a functionality operator O, check if that functionality already exists in the current configuration description C. If it does not, add it to C, and add the clauses in `postcond` to P. If there is an $l' : m(c_1, c_2, ...)$ reuse it and reconnect any channel from/to l from/to l' instead. Go back to 1.
3. If $l : m(c_1, c_2, ...)$ matches a method schema in M, select an instantiated version of the method schema from M with preconditions which are applicable in s (this is a backtrack point). If there is none, report failure and backtrack.
4. Expand the method schema as follows:
   (a) Instantiate the remaining fields of the selected method, and generate new unique labels instead of the general labels `f1`, `f2` etc.
   (b) Add the channels of the method body (with new labels) to the current configuration C.
   (c) Add the method instances (with new labels) of the method body to the top of the stack.
   (d) Use the in and out fields of the method to reconnect any channels in C from the method instance being expanded (i.e. with label l) to the new method instances as labeled in the method body.
5. If the stack is empty, return C and P. Otherwise go back to 1.

Step 4 may need some more explanation. Assume we are expanding `l5: get-door-info(robot1, door4)`, and we have chosen the method in Figure 3. First, we need to replace `r` with `robot1` and `d` with `door4` everywhere in the schema, and `room` is bound to the room of `robot1` (step a). New labels, say `l7` and `l8`, replace `f1` and `f2` (step a). The channel is added to C (step b) and the two functionalities are added to the top of the stack (step c). Finally, according to the `out` field, any existing channel in C with label `l5` (i.e. `get-door-info`) for its out connection is reconnected to `l8` (i.e. `measure-door`) (step d).

The first output from the planner is a configuration description C, which essentially consists of a set of functionality names with labels, e.g. `l8: measure-door(robot1, door4)`, and set of channels, e.g. `(l7, l8, image(robot1, door4), local(robot1))`.

The second output from the planner is the set of postconditions P, which can be used to update the current state, which then in turn can be used as input for generating the configuration following the current one (if any).

It is possible to accidentally specify methods that can result in configurations with cycles. However, these cycles can easily be detected and the faulty configurations can be excluded.

Generally, there are several configurations that can address a problem, but obviously, only one configuration per problem can be performed at the time. By trying different applicable method versions, guided by the cost of configurations, our planner generates the admissible configuration with the lowest cost first.

The planning takes place on the robot that has the information goal. The selected configuration description is then implemented in two steps. First, the different functionalities and channels are distributed according to their location parameter. Then each robot launches a process for executing its assigned functionalities and sets up its channels.

4 Experiments

We have implemented the approach described above, and we have conducted a series of experiments using real robots equipped with different sensors. We report here a representative experiment based on the third and fourth configurations in Figure 2. The platforms used were two Magellan Pro robots from iRobot, shown in Figure 1. Both robots are equipped with compasses and fixed color cameras. The environment consists of two rooms (R1 and R2) with a door connecting them. The door and the robots have been marked with uniform colors in order to simplify the vision task.

The following scenario describes how the two configurations were used, and demonstrates the importance of being able to reconfigure dynamically. Robot A and robot B are in room R1. Robot A wants to go from room R1 to room R2. Since the camera on A is fixed and it has a narrow field of view, the robot cannot see the edges of the door when it is close to it. Therefore, robot A is not able to perform the action on its own. Robot B is equipped with the same sensors as robot A, but since robot B is not crossing the door it is able to observe both the door and robot A from a distance during the whole procedure. We therefore configure our team according to the third configuration in Figure 2, and execute the task. Robot A continuously receives information about the position and orientation during the execution of "cross-door".

When robot A enters room R1 it signals that the task is accomplished. This signal is received by robot B and the current configuration is played out. Next, robot B is assigned the task of going from room R1 to room R2. The same configuration as before is used to solve this task, but with the roles exchanged — i.e., robot A is now guiding robot B. This time, however, during the execution of the "cross-door" behavior a compass fails due to a disturbance in the magnetic field. This makes the current configuration not admissible, and a reconfiguration is necessary to proceed. The fourth configuration in Figure 2 is still admissible even with no compass, and we therefore use this one to carry out the remaining part of the task.

5 Discussion

We have presented a general approach to automatically synthesize a team configuration using knowledge-based techniques. Our approach combines resources and functionalities, residing in different robots, into a functional configuration, in which the robots cooperate to generate the information required to perform a certain task.

Even though the example that we use in this paper to describe our approach is simple, our system is able to generate configurations for more complex problems. For instance, we have also used our configuration planner to address two object transportation tasks. The first task is considering two robots carrying a bar. The second tasks involves three robots building a wall with several wall blocks. In this task, two robots are pushing a wall block together and a third robot is guiding them in order to get the block aligned with the rest of the wall. Both tasks require tight coordination between the involved robots.

An interesting point to note is the relation between our functional configurations and task allocation/assignment [1]. Task allocation typically deals with the question: "Who should do which task?". That is enough for tasks that only require loose coordination. For task that require tight cooperation (that is, they cannot be neatly divided among robots or modules), we must address the additional question: "How to execute a task in a cooperative manner?". Configuration generation answers this question and can be seen as a proceeding step to task allocation, done after a task is assigned to a robot.

A related problem is how to motivate the robots involved in a configuration to commit to it, and to stay committed during the entire execution. For example, in the experiment above, we must guarantee that the observing robot does not run away from the scene. For most tasks, robots are motivated by the individual or the team goals. However, if the task is beneficial only to another individual, as in the case of robots that help each other, it may be harder to motivate a commitment. Task assignment, commitment and functional configurations all address different aspects of the multi-robot cooperation problem. In our work, we concentrate on the last one.

Problems similar to the work on automatic generation of configurations have been studied in several different research areas, e.g. in program supervision [7], automatic web service composition [4], coalition formation [6], and single robot task performance [2]. However, in the field of cooperative robotics, few works address similar problems. Tang and Parker [8] present an approach called ASyMTRe. The principle of ASyMTRe is to connect different schemas (similar to instantiated functionalities) in a way such that a robot team is able to solve tightly-coupled tasks by information sharing. ASyMTRe uses a greedy algorithm that starts with handling the information needs for the less capable robots and continues in order to the most capable robots. For each robot that is handled, the algorithm tries to find the robots with least sensing capabilities and maximal utility to provide information.

In contrast to the ASyMTRe approach, the approach presented in this paper uses a hierarchical planner to automatically generate configurations. We expect that the use of a hierarchical planner will make it easier to deal with the problem of when and how to change (replan) a configuration. This problem is related to the third objective stated in the introduction: how to monitor the performance of a configuration while executing it. We also believe that the use of a hierarchical planner will be beneficial for the next important step, to consider sequences, or plans, of configurations, in order to address more complex tasks. Our current system only considers the generation of configurations for performing one step of a particular task, and cannot deal with situations requiring several steps. In our box pushing example, if a second box is blocking the door, a configuration for removing that box would have to precede the configuration for getting the first box through the door. We are investigating the extension of our planning techniques to generate plans involving sequences of configurations.

Acknowledgments

This work was supported by the Swedish National Graduate School in Computer Science (CUGS), the Swedish Research Council (Vetenskapsrådet), and the Swedish KK Foundation.

REFERENCES

[1] B. Gerkey and M. Matarić, 'A formal analysis and taxonomy of task allocation in multi-robot systems', *International Journal of Robotics Research*, **23**(9), 939–954, (September 2004).

[2] B. Morisset, G. Infante, M. Ghallab, and F. Ingrand, 'Robel: Synthesizing and controlling complex robust robot behaviors', in *Proceedings of the Fourth International Cognitive Robotics Workshop, (CogRob 2004)*, pp. 18–23, (August 2004).

[3] D. Nau, Y. Cao, A. Lothem, and H. Munoz-Avila, 'SHOP: simple hierarchical ordered planner', in *IJCAI*, (1999).

[4] J. Rao and X. Su, 'A survey of automated web service composition methods', in *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, San Diego, California, USA, (July 2004).

[5] A. Saffiotti and M. Broxvall, 'PEIS Ecologies: Ambient intelligence meets autonomous robotics', in *Proc of the Int Conf on Smart Objects and Ambient Intelligence (sOc-EUSAI)*, pp. 275–280, Grenoble, France, (2005). www.aass.oru.se/~peis/.

[6] O. Shehory and S. Kraus, 'Methods for task allocation via agent coalition formation', *Artificial Intelligence*, **101**, 165–200, (1998).

[7] C. Shekhar, S. Moisan, R. Vincent, P. Burlina, and R. Chellappa, 'Knowledge-based control of vision systems', *Image and Vision Computing*, **17**, 667–683, (1998).

[8] F. Tang and L. Parker, 'Coalescing multi-robot teams through ASyMTRe: A formal analysis', in *Proceedings of IEEE International Conference on Advanced Robotics (ICAR)*, (July 2005).