Autonomous Functional Configuration of a Network Robot System

Robert Lundh [*] Lars Karlsson Alessandro Saffiotti

AASS Mobile Robotics Lab
Örebro University, 70182 Örebro, Sweden

Abstract

We consider distributed systems of networked robots in which: (1) each robot includes sensing, acting and/or processing modular functionalities; and (2) robots can help each other by offering those functionalities. A *functional configuration* is any way to allocate and connect functionalities among the robots. An interesting feature of a system of this type is the possibility to use different functional configurations to make the same set of robots perform different tasks, or to perform the same task under different conditions. In this paper, we propose an approach to automatically generate, at run time, a functional configuration of a network robot system to perform a given task in a given environment, and to dynamically change this configuration in response to failures. Our approach is based on artificial intelligence planning techniques, and it is provably sound, complete and optimal. Moreover, our configuration planner can be combined with an action planner to deal with tasks that require *sequences* of configurations. We illustrate our approach on a specific type of network robot system, called PEIS-Ecology, and show experiments in which a sequence of configurations is automatically generated and executed on real robots. These experiments demonstrate that our self-configuration approach can help the system to achieve greater autonomy, flexibility and robustness.

Key words: Cooperative robotics, Planning, Ubiquitous robotics, Robot ecology, PEIS Ecology, Distributed systems, Self-configuration

[*] Corresponding author. Email address: `robert.lundh@aass.oru.se`

1 Introduction

Consider the situation shown in Fig. 1 (left), 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 cannot rely on odometry for this, since the wheels may slip during the push operation, so it needs to use fresh perceptual information. Unfortunately, its camera view is covered by the box, so this cannot be used to measure the position of the door. There are, however, other solutions: a second robot, called Emil, could observe the scene from an external point of view, compute the position of the door relative to Pippi, and communicate this information to Pippi; alternatively, a static camera mounted on the door could provide Pippi the same information.

The above scenario illustrates a general approach to realize cooperation within a team of networked robots — or, more generally, robotic devices. Each robot includes a number of modular *functionalities*: e.g., functionalities for image understanding, localization, planning, and so on. To perform a given task, each robot may use functionalities from other robots in order to compensate for the ones that it is lacking, or to improve its own. Fig. 1 (right) shows how Pippi borrows a perceptual functionality from Emil in the above scenario. In general, we call *functional configuration* a way to activate, parametrize and connect the functionalities among the robots in the system.

Systems consisting of many heterogeneous robots which are pervasively distributed in the environment, and which cooperate to perform possibly complex tasks, are becoming increasingly popular in the field of autonomous robotics. Systems of this type are being studied under different names, including network robot systems [1], intelligent spaces [2], sensor-actuator networks [3], ubiquitous robotics [4], and PEIS-Ecology [5]. Common to these systems is the fact that the term "robot" is taken in a wide sense, including both mobile robots, fixed sensors or actuators, and automated home appliances.

One of the most interesting features of a system of this type is the possibility

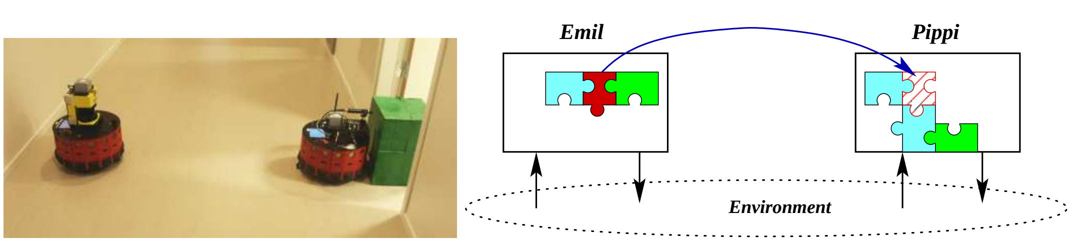


Fig. 1. Left: Emil helps Pippi to push box through the door by measuring the position of the door relative to Pippi. Right: this form of cooperation can be viewed as Emil "lending" a perceptual functionality to Pippi.

to use different functional configurations to make the same system perform different tasks, or perform the same task under different conditions. This capability provides a great potential for flexibility and robustness, by taking advantage of the diversities and the redundancies within the system. This potential, however, is not fully exploited today. Most existing network robot systems are configured by hand, or through hand-written scripts that can only account for a very limited set of contingencies [6–8].

The goal of this paper is to start filling this gap. We propose an approach to automatically generate, on-line, a functional configuration of a network robot system, given information about the target task, the functionalities available in the system, and the current state of the environment. Our approach can also dynamically change the configuration in case of failures. We claim that this approach can endow a network robot system with more *autonomy*, *flexibility*, and *robustness* — in a sense that will be explained in section 6 below.

Our approach to self-configuration uses techniques derived from the field of artificial intelligence planning, and it consists of two parts. The first part is a *configuration planner*, which automatically acquires the current state of the system and generates a functional configuration for a given task. This configuration defines a tight cooperation pattern suitable to perform that task in the current state. The second part is an *action planner*, which automatically generates a sequence of tasks needed to achieve a given goal. Together, the two planners generate at run-time sequences of functional configurations that allow the network robot system to achieve, as a whole, a desired goal.

This paper puts together, in a full coherent system, several pieces that were developed in our previous work on functional configurations [9–11]. In addition, this paper presents two novel contributions: (1) a rigorous formalization of the self-configuration problem, which allows us to prove formal properties of our configuration algorithm; and (2) a systematic series of experiments, performed on a real network robot system, called Peis-Ecology, which allows us to demonstrate the above claims regarding autonomy, flexibility and robustness.

The rest of this paper is organized as follows. In section 2 we discuss related work. In section 3 we give the notion of functional configuration a precise definition. In section 4 we present the top-level process used for self-configuration. Section 5 gives the details of our configuration generation algorithm. Section 6 presents the experiments, and section 7 concludes.

2 Related Work

The general problem of self-configuration of a distributed system is addressed in several fields, including ambient intelligence [12,13], web service composition [14], distributed middleware [15], and autonomic computing [16]. Work on automatic configuration has also been done in other research areas such as: program supervision [17], dynamic software architectures [18], coalition formation [19], and single robot task performance [20,21]. These works, however, do not address the same type of problem considered here: functional configuration of a *distributed system*, in which the components of the system exchange *continuous streams of data* and can *interact with the physical world*.

A few works that address problems closer to ours can be found in the areas of network robot systems and robots in intelligent environments. Intelligent Spaces [7] deal with the problem of how an intelligent environment can actively provide humans and robots with information. In these, distributed intelligent networked devices (DINDs) act as providers of information. In contrast to the work in this article, cooperation in Intelligent Spaces is hard coded, and it can only handle situations that do not require concurrent operations. Ha *et al.* [22] present an approach for automated integration of networked robots into intelligent environments. They use a hierarchical planner to generate sequences of services for a given task. As in traditional web service composition, services are executed in a sequence like actions of a plan. In contrast, functionalities in our approach are executed in parallel and exchange continuous streams of data, which allows us to address tasks that require tight coordination.

Baker *et al.* [23] and Gritti *et al.* [24] both consider configuration problems similar to the one addressed in this paper. In Baker *et al.*, tasks are given to the system in the form of task modules. A suitable configuration for a task is generated by finding the (sensor, effector or computational) components in the system that comply with the constraints of the corresponding task module. Gritti *et al.* proposes a framework inspired by ideas from the field of semantic web service composition. Configurations are created by instantiating generic templates, that express abstract interaction patterns for specific tasks. In both cases, the search for a suitable configuration is done in a reactive way: the component instances that are found first are selected, and the search horizon is limited to one-step lookahead. This is different from the configuration algorithm proposed in this paper, which performs a plan-based search that is sound, complete and optimal. A reactive approach might cope better with large and/or highly dynamic environments, but it cannot guarantee that a sound configuration is found, and that it has the lowest cost.

In the area of cooperative robotics, many works address the problem of task allocation [25]. Task allocation typically deals with the question: “Who should do which task?". Our configuration generation problem, by contrast, deals with the question "How to execute a task in a cooperative manner?". This can be seen as a succeeding step to task allocation. Parker and Tang [26,27] present an approach called ASyMTRe that also addresses this question. The principle of ASyMTRe is to connect different schemas (similar to instantiated functionalities) in such a way that a robot team is able to solve tightly-coupled tasks by information sharing. The approach presented in this paper addresses a similar problem, but it goes one step further by also addressing the automatic generation of *sequences* of configurations.

3 Functional Configurations

As a prerequisite to the presentation of our approach to self-configuration, we need to give a precise definition of the notion of functional configuration.

3.1 Ingredients

We assume that the world can be in a number of different states. The set of all 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 state s_0.

A **functionality** is an operator that uses information to produce additional information. A functionality is specified as

$$f = \langle r, Id, I, O, \Phi, Pr, Po, Cost \rangle \ . \tag{1}$$

Each instance of a functionality is located at a specific robot r and has an identifier Id. The remaining elements of the functionality tuple represent:

- $I = \{i_1, i_2, \ldots, i_k\}$ is a specification of *inputs*, where $i_j = \langle \mathrm{desc}, \mathrm{dom} \rangle$. The descriptor (desc) gives the state variable of the input data, the domain (dom) specifies to which set the data belong. We let $\mathrm{dom}(I)$ denote $\times_{j=1}^{k} \mathrm{dom}(i_j)$.
- $O = \{o_1, o_2, \ldots, o_m\}$ is a specification of *outputs*, where $o_j = \langle \mathrm{desc}, \mathrm{dom} \rangle$ as above.
- $\Phi : \mathit{dom}(I) \rightarrow \mathit{dom}(O)$ specifies the *transfer function* from inputs to outputs.
- $Pr : S \rightarrow \{T, F\}$ specifies the *causal preconditions* of the functionality, i.e., Pr specifies in what states $s \in S$ the functionality can be used.
- $Po : S \times \mathit{dom}(I) \rightarrow S$ specifies the *causal postconditions*. For a given input, Po transforms the world state s before the functionality was executed into

the world state s' after the functionality has been executed.
- *Cost* specifies how expensive the functionality is, e.g., in terms of time, processor utilization, energy, etc. In the remainder of this paper, cost is a single scalar, that summarizes these factors.

Functionalities can incorporate sensors and/or actuators: e.g., a camera can be encapsulated in a camera functionality. Functionalities usually run continuously while active, but some may terminate after some internal condition is met — e.g., a navigation functionality terminates when the goal is reached. We call a functionality of this type a *terminating* functionality.

A **channel** $ch = \langle f_{send}, o, f_{rec}, i \rangle$ transfers data from an output o of a functionality f_{send} to an input i of another functionality f_{rec}.

A **configuration** C is a pair $\langle F, Ch \rangle$, where F is a set of functionalities and Ch is a set of channels. Each channel connects the output of one functionality in F to the input of another functionality in F. In the context of a specific world state s, a configuration $\langle F, Ch \rangle$ is *admissible* if and only if the following two conditions are satisfied:[1]

(1) Each input of each functionality is connected to an output of another functionality with a compatible specification (information admissibility):

$$\begin{aligned} &\forall f \in F\ \forall i \in I_f\ \exists ch \in Ch, f_{send} \in F, o \in O_{f_{send}} : \\ &\quad ch = (f_{send}, o, f, i) \text{ and } \mathrm{desc}(o) = \mathrm{desc}(i) \text{ and } \mathrm{dom}(o) = \mathrm{dom}(i) \end{aligned}$$

(2) All preconditions of all functionalities hold in s (causal admissibility):

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

A configuration also has a **cost**, which is computed by aggregating the costs of the involved functionalities. How aggregation is done is not relevant here.

Note that we view functionalities as black boxes that interact in a simple way. If we wanted to capture functionalities that can interact in more complex ways, a model describing the internal workings of each functionality may be needed. Since functionalities are communicating processes, they could be formally specified using some form of process algebra such as the π-calculus [28]. The simpler model adopted here, however, suffices for the goals of this paper.

[1] We use subscripts to refer to specific elements in a functionality tuple, e.g., r_f refers to the field r in the tuple of f.

3.2 Examples

To illustrate the above concepts, we consider an example inspired by the scenario in the introduction. A robot A is assigned the action of pushing a box through a door. The "cross-door" functionality requires the position and orientation of the door with respect to A. There is a second robot B, and both A and B are equipped with a camera and a compass. The door to cross has a camera on its upper frame. Fig. 2 shows four different (admissible) configurations for the action "cross-door", using the available functionalities. Each configuration $\langle F, Ch\rangle$ is represented by a directed graph, in which nodes represent functionalities in F and arrows represent channels in Ch.

The first configuration (a) involves only A, the robot performing the action, assuming that its camera can view the door's edges while pushing the box (e.g., if the camera is mounted on a tall stand). The camera delivers information to a functionality that measures the pose of the door relative to the robot.

In the second configuration (b), all information is provided by the camera on the door. This is connected to a functionality that measures the pose of A relative to the door. This pose is transformed to the pose of the door relative to A, and is delivered to A.

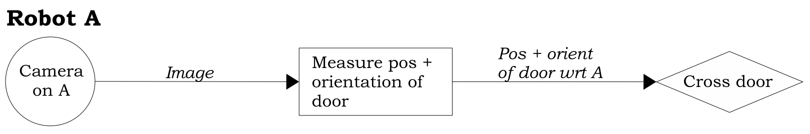
In the third configuration (c), robot B provides the needed information to A. The camera on B is connected to a functionality that measures the pose of the door, and to another one that measures the position of A. These measurements are relative to B. In order to compute the pose of the door relative to A, we use a coordinate transformation, for which we need the position and orientation of robot A relative to B. The latter is obtained by comparing the absolute orientations of the two robots measured by their on-board compasses.

The fourth configuration (d) is similar to (c), except that the orientation of A relative to B is obtained in a different way. Both robots have cameras and have a functionality to 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 measurements, we obtain the orientation of A relative to B.

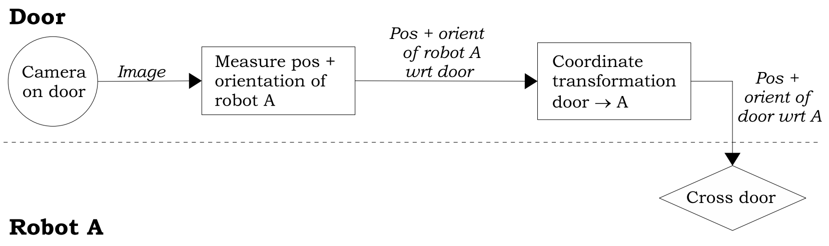
3.3 The configuration problem

Let Σ be a network robot system, and let D be a domain describing, in some formalism, all the functionalities that exist in Σ. D implicitly defines the set $\mathcal{C}(D)$ of all the configurations that can be built in Σ (both admissible and not admissible). Let then A denote an action (or task), and s denote the current

(a)

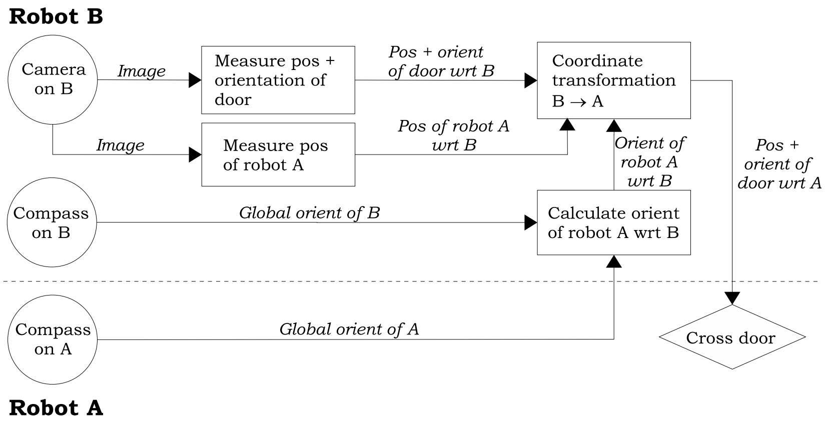


(b)



Door
Camera on door
Image
Measure pos + orientation of robot A
Pos + orient of robot A wrt door
Coordinate transformation door → A
Pos + orient of door wrt A
Cross door
Robot A

(c)



(d)

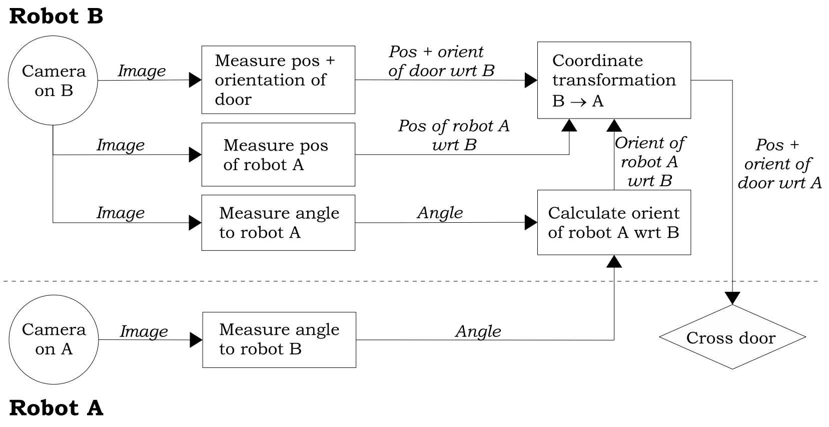


Fig. 2. Four configurations that provide the position and orientation of a door to a "cross door" functionality. Boxes represent functionalities. Circles and diamonds represent functionalities that have embedded sensing and actuation capabilities, respectively. Arrows represent channels.

state A *configuration problem*[2] $\langle A, D, s\rangle$ for Σ is the problem of finding a configuration $C \in \mathcal{C}(D)$ to perform A, which is admissible in state s.

4 The Top-Level Process

The main goal of our work is to provide a sound approach to solve any given configuration problem. In order to be used in a practical network robot system, however, this approach must be embedded in a larger process. In particular, the following aspects should be considered.

First, the configuration problem depends on the current state, including both the state of the network robot system and the one of the environment. Hence, this state should be dynamically acquired before the search for an admissible configuration is started. Second, the configuration problem concerns finding a single configuration to perform one (collective) action A. Many tasks, however, require the performance of a sequence of actions. For instance, in the box-pushing example the robot must first reach the corridor, then move in front of the box, and finally push it trough the door. Hence, an action planner should be included to properly decompose a top-level task into atomic actions. Each one of these actions in turn induces a configuration problem to solve. Finally, each generated configuration should be instantiated in the actual network robot system, and execution should be monitored in order to decide when to switch to the next action, and to detect possible failures.

Fig. 3 gives an overall view of the top-level process that includes all the aspects above. The top-level process is run by one single robot that configures the network robot system to help it solving the top-level task. Note that from now on we reserve the term *task* to denote the top-level task, and use the term *action* to denote each individual sub-task achieved by each configuration. The rest of this section briefly describes the different steps in Fig. 3; step 5 will be described in greater detail in the following section.

4.1 State acquisition

As noted above, our state consists of two parts: system state and world state.

The *system state* contains information relative to the system itself, i.e., which functionalities are currently available, which robotic devices are on/off, and what is their current cost. These facts are represented as literals, e.g.: *rd(pippi)* – pippi is a robotic device; *f(pippi, camera, c1)* – pippi has a functionality

[2] This is a pre-theoretical definition. A more formal one is given in appendix A.

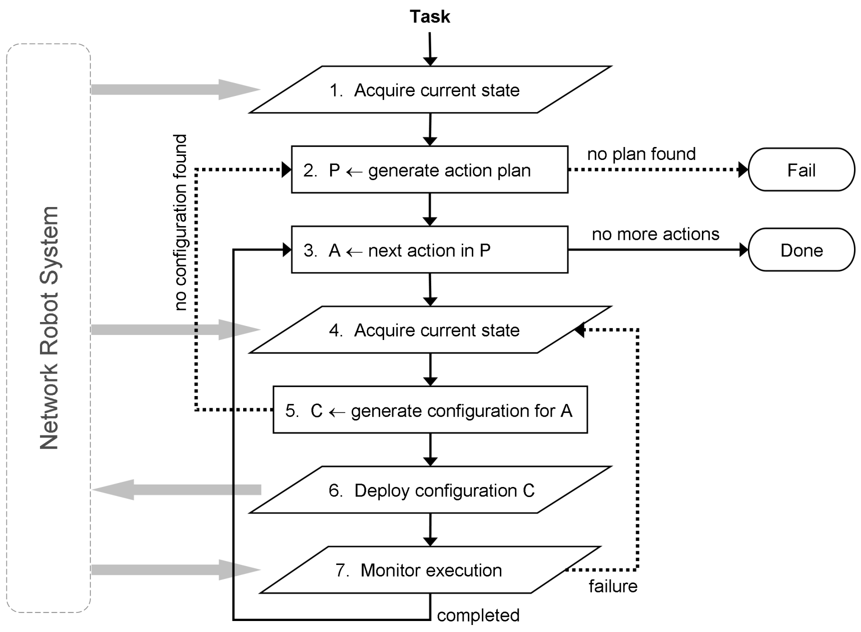


Fig. 3. Flow chart of the top-level process.

c1 that is a camera; *cost(pippi, c1, 100, on)* – c1 costs 100 units to use, and it is currently active, i.e., it does not need to be initialized before it can be used; *f(pippi, navigation-m, nm1)* – pippi has a functionality nm1 that is a navigation module; and *termc(pippi, nm1)* – nm1 is a terminating functionality. Fig. 8 below shows the state acquired in one of our experiments.

The *world state* is a representation of the facts that currently hold in the environment, e.g., information about rooms and places, how they are connected, etc. These facts are represented in a similar way as for the system state, e.g.: *room(kitchen), room(living-room)* – kitchen and living-room are rooms; *conn(kitchen, living-room)* – kitchen and living-room are connected; and *place(pippi, kitchen)* – the current place of robot Pippi is the kitchen.

State information is used to ensure that both action plans and configurations are admissible. How this information is acquired depends on the underlying infrastructure of the specific robot network system. In Section 6.2, we discuss how this is done in the PEIS-Ecology system used in our experiments.

4.2 Action planning and sequencing of actions

In order to generate plans that specify the different steps needed to complete a particular task, we employ a state of the art action planner, called PTLplan [29]. This is a probabilistic planner capable of generating conditional plans with perceptual actions. Although this power is not fully exploited in the experiments reported in this paper, other applications may use these capabilities to deal with more complex situations. The action planner requires as input a domain and a world state. The domain describes all the actions potentially available, and it is hand-coded. The state, acquired right before the planning is done, determines which actions are actually available in the current situation.

An action plan consists of actions like “move(pippi, corridor-3)”, “dock(pippi, box-22)”, and “push(pippi, box-22)”. This plan is given to a sequencer (step 3 in Fig. 3) that is responsible for task execution at the action level. When an action is reported to be completed (step 7), the sequencer takes the next action in the plan and sends it to the configuration planner.

4.3 Configuration generation

We use artificial intelligence planning techniques to generate a description of a functional configuration that performs a given action. This is done by searching the space of configurations to find one which is admissible in the current state and which has the lowest cost, as described in Section 5 below.

4.4 Deployment of a configuration

Once a configuration description is generated, it must be deployed on the network robot system. This involves three steps: first, to activate the functionalities in the configuration which are not already active; second, to set up the channels between the functionalities; third, to subscribe to the appropriate signals from the functionalities in the configuration, which announce termination or failure conditions. As for state acquisition, how deployment is done in practice depends on the infrastructure of the specific network robot system.

4.5 Configuration execution and monitoring

After a configuration has been deployed, execution continues until its main functionality (a terminating one) signals termination. For instance, a “move to” functionality signals termination when its destination is reached. When this happens (step 7 in Fig. 3), the next action is selected.

The normal flow of execution above is marked by the solid arrows in Fig. 3. Our framework also accounts for possible failures, indicated by the dotted lines in Fig. 3. This is done at two levels. At the configuration level, if a functionality in the current configuration fails it sends a `fail` tuple, and the top process tries to generate an alternative configuration for the current action.[3] At the action level, if the configuration planner cannot find an admissible configuration for the current action in the current state, then this action is marked as not available, and the action planner is invoked to replan the entire task, or repair the current plan [31]. If this cannot be done, the task fails.

5 Configuration Generation

We treat configuration generation as a planning problem. Hence, we discuss in this section: how we describe the planning domain, which defines the search space; how we search in this space; and the properties of the search algorithm.

5.1 Domain description

The description of available functionalities uses functionality operator schemes (or simply *operators*) similar to those of AI action planners. Fig. 4 (left) shows an operator from the scenario in the previous section, `measure-door`. With respect to the formal elements of a functionality (Eq. 1 in Sec. 3.1), the meaning of the fields in this operator is as follows. The name `measure-door` together with the second parameter `md` is the Id of the functionality. The `r` parameter corresponds to the r of the functionality. We have an image taken by the camera of `r` as input (I) and from that we compute the position and orientation of the door `d` relative to `r` as output (O). The precondition (Pr) for `measure-door` is that the door `d` is fully visible in the input image. The `cost` gives the default cost of the functionality, if a specific one is not present in the current state. Note that the Φ transfer function is not part of the operator; it corresponds to the actual code for executing the functionality.

In order to combine functionalities to form admissible configurations, the configuration planner uses methods that describe alternative ways to combine functionalities (or other methods) for specific purposes. This is a technique inspired by hierarchical planning, in particular the SHOP planner [32].

A *method* m has the form $m = \langle Id, Pr, I, O, B, Ch, Iconn, Oconn \rangle$ where: Id is the method id, i.e., a term of the form $\alpha(c_1, \ldots, c_n)$; Pr is the set of preconditions; I and O are specifications of inputs and outputs, given as sets of $\langle \text{desc}, \text{dom} \rangle$ pairs; $B = \{m_1, \ldots, m_k\}$ is the body of the method, where each m_i is either a functionality or method id; Ch is a set of internal channels; and $Iconn$ and $Oconn$ are sets of $\langle \langle \text{desc}, \text{dom} \rangle, m' \rangle$ that connect inputs

[3] This assumes that functionalities are able to detect their own failure. Smarter monitoring techniques could be used [30] but this is beyond the scope of this paper.

```
(functionality                                (config-method
  name:     measure-door(r, md, d)              name: get-door-info(r, d)
  precond:  visible(r, d)                       precond: ( f(r, cam, c), f(r, meas-door, m),
  postcond:                                                in(r, room),  in(d, room), rd(r), door(d) )
  input:    (image(r), image)                   in:  -
  output:   (pos(r, d), real-coordinates)       out: f2: pos(r, d)
            (orient(r, d), radians)                  f2: orient(r, d)
  cost: 2 )                                     channels: (f1, f2, image(r), image)
                                                body:
                                                  f1: camera(r, c)
                                                  f2: measure-door(r, m, d) )
```

Fig. 4. Left: a functionality operator. Right: a method for combining functionalities.

in I and outputs in O, respectively, to methods and functionalities in the body B. Each internal channel $ch \in Ch$ has the form $\langle m_1', o_1, m_2', i_2 \rangle$, where $m_1', m_2' \in B, o_1 \in O_{m_1'}$ and $i_2 \in I_{m_2'}$. Intuitively, $\langle I \cup B \cup O, \mathit{Ch} \cup \mathit{Iconn} \cup \mathit{Oconn} \rangle$ can be seen as a directed graph, with input nodes I and output nodes O.

A *method schema* represents a class of methods, and has the same form as a method, but contains variables. The Id component contains parameter variables, and Pr may contain additional (free) variables, which also need to be bound. For simplicity, we use the term method to refer to both methods and method schemas.

Fig. 4 (right) shows an example of a method schema with $Id =$ `get-door-info` that connects a camera functionality and a measure-door functionality on the same robot in order to obtain position and orientation of a given door. The `precond` field corresponds to Pr, `in` to I and *Iconn*, and `out` to O and *Oconn*. In `channels`, which corresponds to Ch, a channel is specified that connects two functionalities in the `body` (labeled `f1` and `f2`). This channel links the output (O) of functionality `f1: camera(r, c)` to the input (I) of functionality `f2: measure-door(r, m, d)`, and has desc = `image(r)` and dom = `image`. The output of `f2`, `pos(r, d)` and `orient(r, d)`, is in the `out` field, so this is the output of the entire method. Thereby, any channel in a method higher up in the expansion hierarchy which is connected to the output of `get-door-info` will be re-connected to the output of `measure-door`.

A method m with $Id = \alpha(v_1, \ldots, v_n)$ is expanded given a method Id $\alpha(c_1, \ldots, c_n)$ as follows. Let σ be the substitution that unifies the two terms. The preconditions $Pr\sigma$ are tested, yielding a new set of substitutions $\{\sigma_1, \ldots \sigma_h\}$ also including bindings for the free variables in Pr; if the preconditions do not hold, that set is empty. The result of the expansion is a set of methods $\{m\sigma_1, \ldots, m\sigma_h\}$ where $m\sigma_i = \langle Id\sigma_i, Pr\sigma_i, I\sigma_i, O\sigma_i, B\sigma_i, Ch\sigma_i, \mathit{Iconn}\sigma_i, \mathit{Oconn}\sigma_i \rangle$. There can be several methods with the same *Id*. Their total expansion is then the union of their individual expansions. Note that all methods with the same *Id* must have the same I and O (although the elements in I and O can be connected differently.) Having several methods with the same Id is a way to have alternative ways to obtain the same O given the same I. For instance, an alternative method for Id `get-door-info` could use a laser instead of a camera to measure the door position. A functionality operator o is expanded in a similar way, yielding a functionality $o\sigma$.

We call *configuration domain* a pair $D = \langle F, M \rangle$ where F is a set of functionality operators, and M is a set of methods.

5.2 The algorithm

The configuration planner takes as input a configuration problem $\langle G, D, s \rangle$ where G is a robot goal stack with initially one unexpanded method instance (corresponding to an action), D is a domain, and s is a (dynamically acquired) state It maintains an initially empty configuration description C and an initially empty set of causal postconditions P. It works as follows:

(1) Take the top element $\alpha(a_1, a_2, \ldots)$ of G.
(2) If $\alpha(a_1, a_2, \ldots)$ matches the *Id* of an operator o, first check that its preconditions hold in s. (This is a backtrack point.) If they do, instantiate o, add the resulting functionality f to the configuration C, and add the clauses in Po to P. Calculate the cost of that functionality for that specific instantiation, first by querying the system state, and if that fails by taking the default value from the operator. (There is an extra start-up cost associated with inactive functionalities.) Go back to 1.
(3) If $\alpha(a_1, a_2, \ldots)$ matches a method in D, non-deterministically select an expansion m. (This is a backtrack point). If there is none, report failure and backtrack.
(4) Expand m as follows:
   (a) Add the channels Ch_m to the current configuration C.
   (b) Add the functionality and method *Ids* in B_m to the top of G.
   (c) Use the in and out connection fields $Iconn_m$ and $Oconn_m$ to reconnect any channels in C as described in the definition of Π (appendix A).
(5) If G is empty, return $\langle C, P \rangle$. Otherwise go back to 1.

To illustrate how the planner functions, let us assume it is expanding `l5: get--door-info(pippi, door4)` (step 1). It chooses to use the method in Fig. 4. First, it replaces `r` with `pippi` and `d` with `door4` everywhere in the method. It then looks in s to verify that `pippi` is a robot device and that is has a camera functionality (`f(r, cam, c)`; c needs to be bound to an actual camera functionality), and a measure-door functionality (`f(r, meas-door, m)`; m also needs to be bound). It also needs to find a binding for `room` that satisfies the conditions `in(pippi, room)` and `in(door4, room)` (step 3). New labels, say `l7` and `l8`, replace `f1` and `f2` in the body of the method. The channel is added to the current configuration (step 4a) and the two functionalities `l7: camera(pippi, c1)` and `l8: measure-door(pippi, md1, door4)` are added to the top of G (step 4b). Finally, we go through the channels already in the configuration, and any channel we find with label `l5` (`get-door--info(pippi, door4)`) for its out connection is reconnected to `l8` (`measure--door(pippi, md1, door4)`) (step 4c). We proceed to step 5, and then return to step 1. There we find `l7: camera(pippi, c1)` at the top of the stack, which matches a functionality operator and is added to the current configuration (step 2). And so on.

The planner returns a pair $\langle C, P\rangle$. The first value is a configuration description, which essentially consists of a set F of labeled functionalities, e.g., `l8: measure-door(pippi, md1, door4)`, and a set Ch of channels, e.g., `(l7, l8, image(pippi, door4))`. The second value is the set of postconditions P, which can be used to update the current state, which then can be used as input for generating the configuration following the current one (if any).

The above algorithm assumes that the system state is acquired before planning starts (step 4 in Fig. 3). For large network robot systems, this could be a time consuming process. However, the algorithm can be easily modified to acquire the state on demand for those parts of the system that are currently of interest. Note that a different state can result in a different set of admissible configurations. In the cross-door example, if one robot does not have a compass the third configuration in Fig. 2 above is not generated.

5.3 Formal properties

As expected with a centralized plan-based approach, the configuration planner has good formal properties. We state the following (proofs in Appendix A):

Theorem 1 (Soundness) *If D only contains information admissible methods (defined in Appendix A), then the configuration algorithm generates only admissible configurations.*

Theorem 2 (Completeness) *The configuration algorithm eventually generates any admissible configuration that is defined by the functionalities and methods in D.*

The configuration algorithm has been given in a non-deterministic form. To provide optimality, we must complement it with a search strategy. We use best first with branch-and-bound [33], using the cost of the partial configuration as a lower bound and pruning when this exceeds the lowest cost among the complete configurations generated so far. The search terminates when all partial configurations on the stack have a higher cost.

Theorem 3 (Optimality) *If the configuration algorithm uses the above search* *strategy, it returns the admissible configuration with the lowest cost that is defined by the functionalities and methods in D.*

6 Experiments

The proposed framework has been implemented in a specific instance of a network robot system, called a PEIS-Ecology. The experiments described here have been executed with this system in a home-like environment, using real robots and robotic devices with different functional, sensory, and actuation capabilities. All the experiments are based on a scenario originally presented by Broxvall et al. [8]. It should be emphasized that in the original work all the configurations were static, determined *a-priori*, and hand coded.

6.1 Goal of the experiments

The goal of the experiments reported here is twofold. First, to better illustrate the mechanisms of our self-configuration framework. Second, to validate the three claims made in the introduction, namely that our framework has the ability to: (1) automatically configure a network robot system to perform a given task, even when this task requires several sub-tasks to be accomplished (*autonomy*); (2) generate different configurations to cope with different tasks and conditions (*flexibility*); and (3) dynamically re-configure the system in response to failure of some of its components (*robustness*).

To achieve these goals, we have conducted three different series of experiments. The first series addressed autonomy: the robot was assigned a high-level task, and it had to autonomously generate a sequence of actions to perform this task, and for each action it had to configure the PEIS-Ecology in a suitable way. The second series was similar to the first one, but the task was assigned to a robot with different sensors and different initial conditions. This series was meant to demonstrate flexibility. The last series was meant to demonstrate robustness: a failure of one component in the configuration was forced during execution, and the system had to re-configure in order to complete the task.

6.2 The PEIS-*Ecology testbed*

The concept of PEIS-Ecology, originally proposed by Saffiotti and Broxvall [5], is one of the few existing concrete realizations of the notion of network robot system. The main constituent of a PEIS-Ecology is a *physically embedded*

```
(functionality
  name:    object-tracker(r, ot, obj)
  precond: f(obj-tra, r, ot), object(obj)
  input:   (image(r), image)
  output:  (pos(r, d), real-coordinates),
           (orient(r, d), radians)
  cost:    5)
```

```
(functionality
  name:    odor-classifier(p, oc, obj)
  precond: f(od-clr, p, oc), object(obj)
  input:   (smell(obj), smell),
           (rfid-reading(obj), tag)
  output:  (odor(obj), signature)
  cost:    10)
```

Fig. 5. Two functionalities for a PEIS-Ecology.

intelligent system, or PEIS. This is any computerized system interacting with the environment through sensors and/or actuators and including some degree of "intelligence". A PEIS generalizes the notion of robot, and it can be as simple as a toaster or as complex as a humanoid robot. A PEIS-Ecology consists of a number of PEIS embedded in the same physical environment, and endowed with a common communication and cooperation model. Communication relies on a shared tuple-space: PEIS exchange information by publishing tuples and subscribing to tuples. Cooperation relies on the notion of linking functional components: each PEIS can use functionalities from other PEIS in the ecology to complement its own. The PEIS-Ecology model has been implemented in an open-source middleware, called the PEIS-kernel. (See [34] for more details.)

In our experiments, we use our self-configuration framework to automatically configure a PEIS-Ecology. There are three elements in our framework that depend on the specific system being configured: the domains used by the planners (steps 2 and 5 in Fig. 3); the way to acquire the current state (steps 1 and 4); and the way to deploy and monitor a configuration (steps 6 and 7).

For the domains, since these are static they have been hand-coded. Fig. 5 shows two examples of functionality operators for the configuration planner in the PEIS-Ecology domain.

To acquire the current (system and world) state from the PEIS-Ecology, we use the mechanisms provided by the PEIS-kernel. To acquire the system state, the configuration process subscribes to the reserved key `name` to retrieve, for each functionality in the ecology, a tuple that contains its name, location (PEIS), status (on/off), and cost. On each PEIS there is a special functionality, called PEIS-init, which publishes this information in the distributed tuple-space.[4] For the world state, each functionality publishes a tuple describing its own part of the state, e.g., its current physical location or the state of an actuator attached to it. All these facts are put together into an overall world state, used by both the configuration planner and the action planner.

Deployment of a configuration description into a PEIS-Ecology is done in three steps. First, inactive functionalities are activated through the PEIS-init functionalities. Second, the needed channels are set-up by creating tuple subscrip-

[4] The cost of an active functionality is provided by the functionality itself; for an inactive functionality, PEIS-init provides an estimate of the cost of starting it.

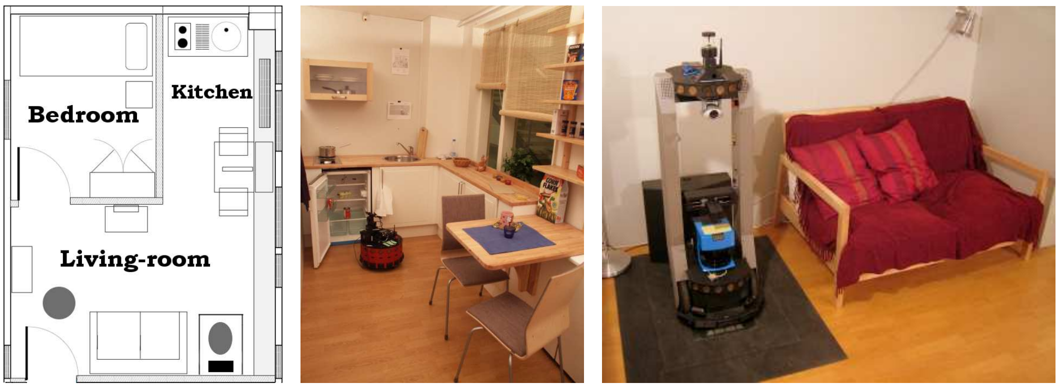


Fig. 6. Left: A sketch of the PEIS-Home. Middle: Pippi is about to smell inside the fridge. Right: Astrid at the charger.

tions between the involved functionalities. Finally, the configuration process subscribes to `done` tuples from terminating functionalities to be notified of completion; it also subscribes to `fail` tuples from all the functionalities in the configuration, to be notified on failures.

6.3 Experimental setup

We have run our experiments in a physical test-bed facility, called the PEIS-Home, which looks like a typical bachelor apartment of about $25m^2$ — see Fig. 6. It consists of a living-room, a bedroom and a small kitchen. The PEIS-Home is equipped with a communication and computation infrastructure, and with a number of PEIS. The following PEIS are of particular importance for our experiments.

Pippi: An iRobot Magellan Pro robot equipped with a color camera and with a Cyranose 320$^{\text{TM}}$ electronic nose used to classify odors. It runs the Thinking Cap [35] navigation controller, and several instances of Player [36] providing functionalities to interface with different sensors and actuators. It can also run the configuration process.

Astrid: A PeopleBot robot from ActivMedia Robotics equipped with a laser range finder, a Pan-Tilt-Zoom color camera, a panoramic camera, a gripper, a speech synthesis system, and (optionally) a Cyranose 320$^{\text{TM}}$. Astrid can run the Thinking Cap, Player, and the configuration process, as well as a scan-matching self-localization algorithm.

The PEIS-Fridge: A refrigerator with two simple Figaro gas sensors, a motorized door, an RFID tag reader that can read the tags on the items placed in the fridge, and an embedded computer.

The Home Security Monitor (HSM): A PC connected to a set of web-cameras mounted in the ceiling. It includes an object tracker and a deliberator that reacts to alarms published in the tuple-space.

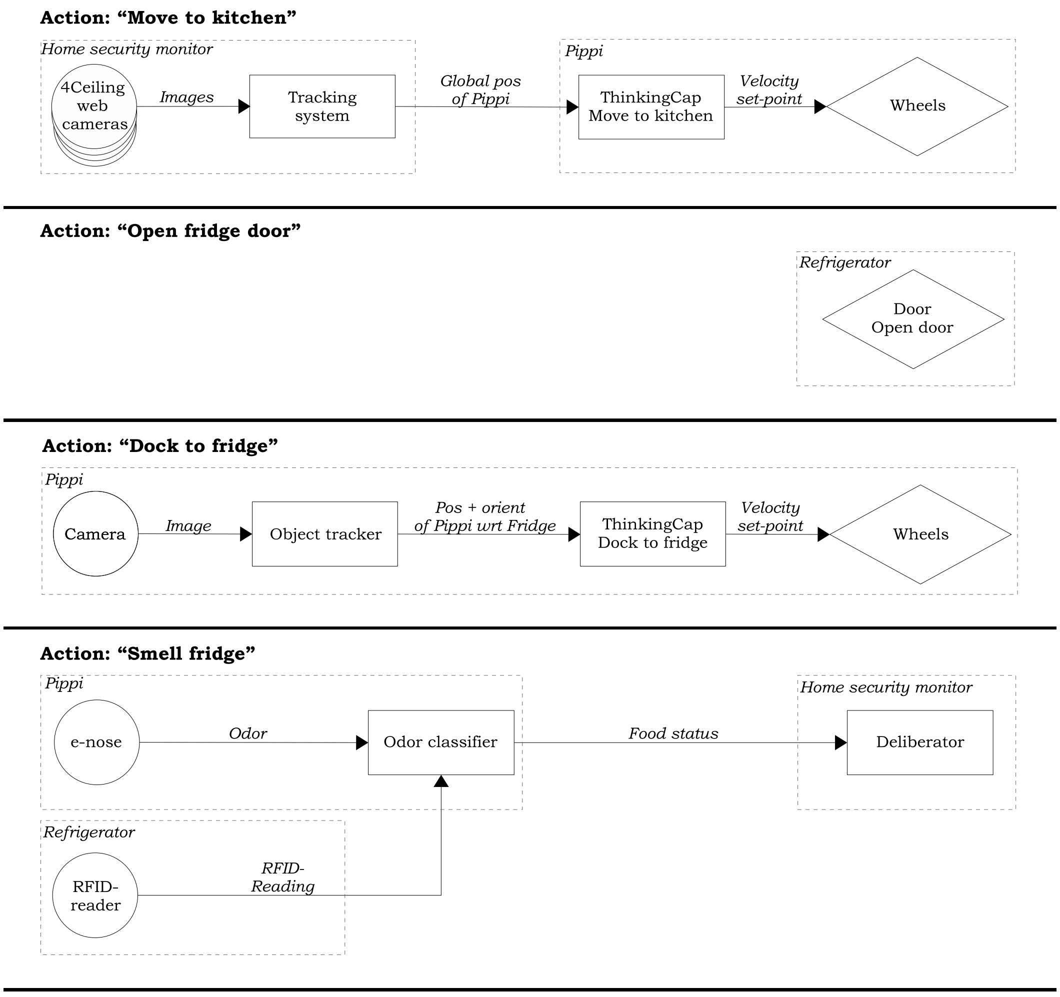


Fig. 7. The Peis-Ecology configurations (in simplified form) generated for the actions: "move to kitchen", "open fridge door", "dock to fridge", and "smell fridge".

At start-up, Pippi is located close to the entrance, and Astrid is located at the charging station in the living-room.

6.4 Experiment 1: Self-configuration

The first experiment illustrates the ability of our framework to automatically generate a sequence of configurations to perform a given task in the current context (state). The experiment unfolds as follows.

(a) The gas sensors in the fridge trigger an alarm, and post a tuple in the tuple-space. This is notified to the deliberator component in the HSM, which decides to send Pippi to make an olfactory inspection of the contents of the fridge. The deliberator sends a tuple with the high level goal `(smell-fridge)` to Pippi.

```
rd(Pippi), rd(Astrid), rd(HSM), rd(Fridge),          f(HSM, tr-sys, tr1), cost(HSM, tr1, 20, on),
f(Pippi, cam, c1), cost(Pippi, c1, 40, on),          f(Astrid, cam, c2), cost(Astrid, c2, 100, on),
f(Pippi, obj-tra, ot2), cost(Pippi, ot2, 130, on),   f(Astrid, laser, l1), cost(Astrid, l1, 100, off),
f(Pippi, odomet, od1), cost(Pippi, od1, 100, on),    f(Astrid, odomet, od1), cost(Astrid, od1, 10, on),
f(Pippi, TC, tc1), cost(Pippi, tc1, 20, on),         f(Astrid, TC, tc1), cost(Astrid, tc1, 20, on),
f(Pippi, e-nose , e1), cost(Pippi, e1, 50, on),      f(Astrid, obj-tra, ol3), cost(Astrid, ot3, 130, on),
f(Pippi, odor-cls , od1), cost(Pippi, od1, 40, on),  f(Astrid, self-loc, s1), cost(Astrid, s1, 20, on),
f(Pippi, wheels , w1), cost(Pippi, w1, 60, on),      f(Astrid, wheels , w1), cost(Astrid, w1, 55, on),
f(Fridge, door, dr1), cost(Fridge, dr1, 40, on),     termc(Pippi, tc1), termc(Astrid, tc1),
f(Fridge, rfid , rr2), cost(Fridge, rr2, 35, off),   termc(Pippi, od1), termc(Fridge, dr1)
f(HSM, cams, ca1), cost(HSM, ca1, 20, on),
```

Fig. 8. System state acquired in Experiment 1 (partial).

(b) The configuration process located at Pippi acquires the current state, and based on this generates an action plan (steps 1 and 2 of the process in Fig. 3). The plan consists of the actions "move to kitchen", "open fridge door", "dock to fridge", and "smell fridge".
(c) The first action "move to kitchen" is selected (step 3 in Fig. 3). Pippi acquires the current state to find out the functionalities available in the Peis-Ecology (step 4). Part of this state is visualized in Fig. 8. The configuration planner is then invoked (step 5). Given the state, there are three possible ways to configure the Peis-Ecology to allow Pippi to perform the "move" action. These ways differ in how the `ThinkingCap` component in Pippi obtains self-position information: (1) from the odometry on Pippi; (2) from the tracking system in the HSM, which can track the position of Pippi using the ceiling cameras; or (3) from Astrid, which can track the position of Pippi using its own camera. The configuration planner selects the second option since it has the lowest cost[5] (first row in Fig. 7).
(d) This configuration is deployed to the Peis-Ecology, and execution begins (step 6). While navigating, the `ThinkingCap` component in Pippi receives position updates from the HSM in the form of a stream of tuples.
(e) When the `ThinkingCap` establishes that Pippi has reached the kitchen, it posts a `done` tuple. This is notified to the configuration process (step 7), which then selects the next action "open fridge door" (step 3).
(f) The current state is again acquired (step 4), and a configuration to open the fridge door is generated (step 5). In the current implementation, the only configuration which achieves this is the one that consists of the single `open-fridge-door` component in the fridge, which does not require any information. (Second row in Fig. 7.) A smarter component could use information about free space in front of the fridge, obtained, for instance, from the ceiling cameras: accordingly, the configuration planner would include these components in the configuration.
(g) The new configuration is deployed (step 6) and executed (step 7).
(h) When the `open-fridge-door` component signals termination, the next action "dock to fridge" is selected and the current state is acquired (steps 3–4). To perform this action, Pippi needs to know the location of the

[5] In our implementation, costs are heuristic values which also include reliability. A better treatment of costs could be done, but it is beyond the scope of this paper.

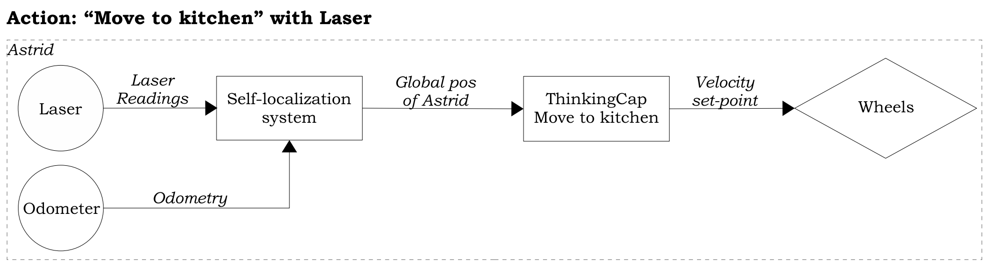


Fig. 9. Configuration for the action "move to kitchen" performed by Astrid, using a laser and a scan-matching algorithm.

fridge with respect to herself. This information can be obtained by the same methods considered for the "move to kitchen" action, or by using a camera on-board Pippi and a vision system to track the fridge edges. The configuration planner selects and deploys this last alternative (steps 5–6) since it has the lowest cost (third row in Fig. 7).

(i) When the action is completed, the next action "smell fridge" is selected. The `odor-classifier` on Pippi requires context information about the objects being smelled, which can be obtained from the RFID tags on the items in the fridge. The configuration planner thus generates the configuration in the last row in Fig. 7, and deploys it (steps 5–6).
(j) Finally, the `odor-classifier` sends the classification results to the deliberator in HSM and signals termination (step 7). Since there are no more actions, the execution of the top level task completes (step 3).

Based on the classification results, the deliberator may decide to start a new task, e.g., to ask Pippi to move to the bedroom and alert the occupants of the Peis-Home. This task would be done in a way similar to the one above.

6.5 Experiment 2: Different situation

The second experiment reproduces the first one, with the only difference that in step (a) the deliberator assigns the task (smell-fridge) to Astrid instead of Pippi, which was removed from the Peis-Ecology. Accordingly, the configuration process was executed on Astrid. Note that Astrid has a different sensor suite and a different starting position than Pippi. These facts are automatically acquired during the state acquisition phase: no manual adjustments were made between the first and the second experiment.

Astrid successfully completed the task. The execution unfolded as in Experiment 1 replacing "Pippi" by "Astrid", with the following two exceptions:

- The state acquired by Astrid was a subset of the previous one (see Fig. 8), in which all the literals associated to Pippi were not present.
- The `ThinkingCap` component on Astrid can obtain position information in

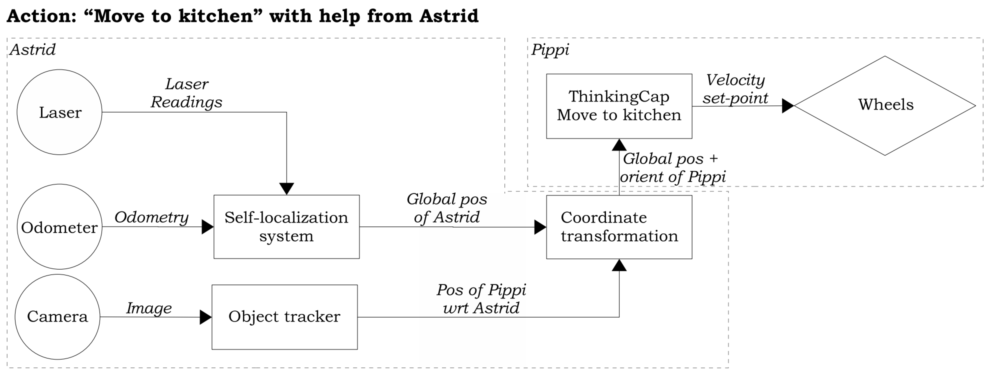


Fig. 10. Configuration for action "move to kitchen". Pippi gets help from Astrid.

two ways: (1) from the scan-matching self-localization system on Astrid, which takes laser data plus odometry as input; or (2) from the tracking system in the HSM. Correspondingly, the set of possible configurations to perform the action "move to kitchen" is different from the previous case. In our experiment, the configuration planner selected the first option, shown in Fig. 9, since this had the lowest cost.

6.6 Experiment 3: Respond to failures

The third experiment is a variant of the first one, in which a component fails during execution. Steps (a) to (d) proceed as in Experiment 1: Pippi is given the top-level task `(smell-fridge)`, generates an action plan, and generates a configuration for the first action "move to kitchen". However, during step (d) a failure occurs. Execution then proceeds as follows.

(d') The `object-tracker` component in HSM is unable to track Pippi because of poor lighting conditions, and it posts a `fail` tuple. This is received by the configuration process (step 7 in Fig. 3), which halts the execution of the current configuration.

(d") The current state of the system is acquired excluding the component which has failed, and a search for a new configuration is started (steps 4–5). This time the only available options to provide position information to the `ThinkingCap` in Pippi are: (1) from odometry on Pippi; or (2) from visual tracking on Astrid. The configuration planner selects the second option, shown in Fig. 10, based on its cost/reliability.

(d"') The new configuration is deployed, and it is executed cooperatively by Pippi and Astrid (step 6).

The rest of the experiment (steps e–j) proceeded as in Experiment 1.

6.7 Discussion

The above experiments were meant to demonstrate that our framework provides a network robot system with more autonomy, flexibility and robustness. Have these goals been achieved?

In the first experiment, a high-level task was decomposed into four actions, and the system self-configured to perform each action in turn. Some actions required tight coordination between robots: e.g., in the first action, the HSM was sending a continuous stream of position information to Pippi. No human intervention was required at any time. This experiment therefore successfully demonstrated the claim of *autonomy*. In the second experiment, the same task was successfully accomplished in a situation in which both the world state and the system state were different. The only manual interventions were: switch off the first robot (Pippi), and send the top-level task to Astrid instead of Pippi. This experiment therefore demonstrated the claim of *flexibility*. The last experiment replicated the first one, but a functionality failure was forced during execution. The system autonomously re-configured and continued the task until it was successfully accomplished. No manual intervention was required. This experiment therefore demonstrated the claim of *robustness*.

It should be noted that the above evaluation is only qualitative. This is adequate, since our aim was to verify the ability of our framework to generate the right configuration in each situation, not to measure the performance of the full system — which depends on the quality of the implemented functionalities. Several experiments have been performed in addition to the ones reported above, to test different situations; sample videos of these experiments are available on-line [34].

7 Conclusions

The approach presented in this paper gives a network robot system the ability to self-configure to perform a given task in the current situation. This approach combines functionalities, residing in different robots, into a functional configuration in which the robots cooperate to perform the task. Another way to look at our approach is that a virtual robot is built on the fly by combining software and hardware components residing on different robots, in order to collectively perform a given task in a given situation.

The ability to dynamically self-configure and re-configure is pivotal to the autonomy, flexibility and robustness of a network robot system. We have shown empirically that the approach presented here facilitates those properties; moreover, we have proved that this approach has the formal properties of soundness, completeness and optimality. The entire self-configuration process (Fig. 3), from state acquisition to a possible failure repair, is performed in real time without any manual intervention. We are not aware of other network robot systems that enjoy this same set of properties.

Our approach has a couple of important limitations. First, action planning and configuration planning are done almost independently. This entails that there is no guarantee, in general, that the planned actions will actually be executable: it may happen that no admissible configuration exists at the time when an action must be executed. In this case, the action planner is re-invoked to generate an alternative plan starting from the current situation. While this run-time backtracking approach is often effective, it may lead to sub-optimal executions in some cases. In order to guarantee global optimality of the whole sequence of configurations (not only of each individual configuration) action planning and configuration planning should be more tightly coupled. How to do this, however, is a rather complex research issue.

The second limitation is that we only consider the execution of a single top-level task. In general, several tasks might be performed concurrently, and new tasks might dynamically appear. A natural extension of the current framework would be to use task allocation techniques to assign different tasks to different configuration processes. With such an extension, issues such as resource handling, conflict resolution and deadlocks must also be considered. It is important to realize that both this problem and the previous one do not reflect inherent limitations of our approach, but rather restrictions in the scope of our current work. We plan to address both issues in our future research.

Finally, we emphasize that the self-configuration approach presented in this paper can potentially be applied to any distributed, component-based robotic system — in particular, it can be and has been applied beyond the Peis-Ecology framework used in our experiments. Examples involving cooperative transportation and cooperative perception tasks are reported by Lundh [37].

Acknowledgments

This work was supported by CUGS (the Swedish National Graduate School in Computer Science), and ETRI (Electronics and Telecommunications Research Institute, Korea) through the project “Embedded Component Technology and Standardization for URC (2004-2008)”. We thank Mathias Broxvall and Amy Loutfi for their help with the Peis-Ecology testbed, and Marco Gritti for many enlightening discussions.

References

[1] T. Akimoto, N. Hagita, Introduction to a network robot system, in: Proc of the Int Symp on Intelligent Signal Processing and Communication Systems, Tottori, Japan, 2006, pp. 91–94.

[2] J. Lee, H. Hashimoto, Intelligent space – concept and contents, Advanced Robotics 16 (3) (2002) 265–280.

[3] F. Dressler, Self-organization in autonomous sensor/actuator networks, in: Proc. of the 19th IEEE Int Conf on Architecture of Computing Systems, 2006.

[4] J. Kim, Y. Kim, K. Lee, The third generation of robotics: Ubiquitous robot, in: Proc of the 2nd Int Conf on Autonomous Robots and Agents (ICARA), Palmerston North, New Zealand, 2004.

[5] A. Saffiotti, M. Broxvall, PEIS ecologies: Ambient intelligence meets autonomous robotics, in: Proc of the Int Conf on Smart Objects and Ambient Intelligence (sOc-EUSAI), Grenoble, France, 2005, pp. 275–280.

[6] L. Chaimowicz, A. Cowley, V. Sabella, C. J. Taylor, ROCI: a distributed framework for multi-robot perception and control, in: Proc of the IEEE/RJS Int Conf on Intelligent Robots and Systems (IROS), Las Vegas, USA, 2003, pp. 266–271.

[7] J. Lee, K. Morioka, N. Ando, H. Hashimoto, Cooperation of distributed intelligent sensors in intelligent environment, IEEE/ASME Transactions on Mechatronics 9 (3) (2004) 535–543.

[8] M. Broxvall, S. Coradeschi, A. Loutfi, A. Saffiotti, An ecological approach to odour recognition in intelligent environments, in: Proc of the IEEE Int Conf on Robotics and Automation, Orlando, FL, 2006, pp. 2066–2071.

[9] R. Lundh, L. Karlsson, A. Saffiotti, Plan-based configuration of a group of robots, in: Proc of the 17th European Conf on Artificial Intelligence (ECAI), Riva del Garda, Italy, 2006, pp. 683–687.

[10] R. Lundh, L. Karlsson, A. Saffiotti, Plan-based configuration of an ecology of robots, in: Proc of the IEEE Int Conf on Robotics and Automation, Rome, Italy, 2007, pp. 64–70.

[11] R. Lundh, L. Karlsson, A. Saffiotti, Dynamic self-configuration of an ecology of robots, in: Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems, San Diego, CA, 2007, pp. 3403–3409.

[12] A. Kaminsky, Infrastructure for distributed applications in ad hoc networks of small mobile wireless devices, Tech. rep., Rochester Institute of Technology, IT Lab (may 2001).

[13] M. Hellenschmidt, T. Kirste, Self-organization for multi-component multi-media environments, in: Proc of the UniComp Workshop on Ubiquitous Display Environments, 2004.

[14] J. Rao, X. Su, A survey of automated web service composition methods, in: Proc of the Int Workshop on Semantic Web Services and Web Process Composition (SWSWPC), San Diego, CA, 2004.

[15] T. J. community, The project JXTA web site, www.jxta.org.

[16] G. Tesauro, D. Chess, W. Walsh, R. Das, A. Segal, I. Whalley, J. Kephart, S. White, A multi-agent systems approach to autonomic computing, in: Proc of the Int Conf on Autonomous Agents and Multiagent Systems, 2004, pp. 464–471.

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

[18] J. Bradbury, J. Cordy, J. Dingel, M. Wermelinger, A survey of self-management in dynamic software architechture specifications, in: Proc of the Int Workshop on Self-Managed Systems (WOSS), 2004.

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

[20] B. Morisset, G. Infante, M. Ghallab, F. Ingrand, Robel: Synthesizing and controlling complex robust robot behaviors, in: Proc of the Int Cognitive Robotics Workshop, (CogRob), 2004, pp. 18–23.

[21] D. Kim, S. Park, Y. Jin, H. Chang, Y.-S. Park, I.-Y. Ko, K. Lee, J. Lee, Y.-C. Park, S. Lee, SHAGE: a framework for self-managed robot software, in: Proc of the Int Workshop on self-adaptation and self-managing systems, 2006.

[22] Y. Ha, J. Sohn, Y. Cho, Automated integration of service robots into ubiquitous environments, in: Proc of the Int Conf on Ubiquitous Robots and Ambient Intelligence (URAI), 2006, pp. 177 – 182.

[23] D. Baker, G. McKee, P. Schenker, Network robotics, a framework for dynamic distributed architechtures, in: Proc of the IEEE/RSJ Int Conf on Intelligent Robots and Systems, Sendai, Japan, 2004, pp. 1768–1773.

[24] M. Gritti, M. Broxvall, A. Saffiotti, Reactive self-configuration of an ecology of robots, In: ICRA workshop on Network Robot Systems (2007).

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

[26] L. E. Parker, F. Tang, Building multi-robot coalitions through automated task solution synthesis, Proc of the IEEE, special issue on Multi-Robot Systems 94 (7) (2006) 1289–1305.

[27] F. Tang, L. Parker, A complete methodology for generating multi-robot task solutions using asymtre-d and market-based task allocation, in: Proc of the IEEE Int Conf on Robotics and Automation, Rome, Italy, 2007, pp. 3351–3358.

[28] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, i, Information and Computation 100 (1) (1992) 1–40.

[29] L. Karlsson, Conditional progressive planning under uncertainty, in: Proc of the Int Joint Conf on Artificial Intelligence (IJCAI), Seattle, USA, 2001, pp. 431–438.

[30] O. Pettersson, Execution monitoring in robotics: A survey, Robotics and Autonomous Systems 53 (2) (2005) 73–88.

[31] A. Bouguerra, L. Karlsson, Symbolic probabilistic-conditional plans execution by a mobile robot, in: IJCAI-05 Workshop: Reasoning with Uncertainty in Robotics (RUR), 2005.

[32] D. Nau, Y. Cao, A. Lothem, H. Munoz-Avila, SHOP: simple hierarchical ordered planner, in: Proc of the Int Joint Conf on Artificial Intelligence (IJCAI), Stockholm, Sweden, 1999, pp. 968–973.

[33] E. L. Lawler, D. E. Wood, Branch-and-bound methods: a survey, Operations Research 14 (1966) 699–719.

[34] The PEIS ecology project, Official web site, www.aass.oru.se/˜peis/.

[35] A. Saffiotti, K. Konolige, E. H. Ruspini, A multivalued-logic approach to integrating planning and control, Artificial Intelligence 76 (1-2) (1995) 481–526.

[36] Player/Stage Project, playerstage.sourceforge.net/.

[37] R. Lundh, Plan-based configuration of a group of robots, Licentiate Thesis. University of Örebro, Sweden (September 2006).

A Proofs of Formal Properties

In order to give proofs for Theorems 1-3 in Section 5.3, we need to give a more formal definition of the configuration problem and configuration solution. It is also important to define what it means for a method m to be well specified, i.e., information admissible. We say that an output $o \in O_m$ is bound if its data are produced inside the method, i.e., there is some $m' \in B$ such that $o \in O_{m'}$ and $\langle o, m' \rangle \in Oconn_m$. For each $m' \in B$ and each $i \in I_{m'}$ we say that i is bound if its data are produced in the method, i.e., there is either a channel $\langle m'', o, m', i \rangle$ such that $\mathrm{dom}(i) = \mathrm{dom}(o)$, $\mathrm{desc}(i) = \mathrm{desc}(o)$ and $m'' \in B$ and $o \in O_{m''}$, or supplied as input to the method, i.e., $i \in I_m$ and $\langle i, m' \rangle \in Iconn_m$. Then, we say that a method m is information admissible if: (1) all $o \in O$ are bound, and (2) for all $m' \in B$ and all $i \in I_{m'}$, i is bound.

A *configuration problem* is a tuple (G, D, Ch, s) where G a list of goals (i.e., functionality or method *Ids*), D is the domain specifying the functionalities and methods, Ch is the set of channels connected to the goal elements in G, and s is a state. If $P = (G, D, Ch, s)$, then $\Pi(P)$, the set of all configurations for G in s for D, is defined recursively as follows.

(1) If G is empty, then $\Pi(G, D, Ch, s)$ contains exactly one configuration, namely the empty configuration $\langle \emptyset, \emptyset \rangle$. Otherwise, let g be the first goal in G and G' be the remaining goals, and continue as follows.
(2) If g is a functionality *Id* which expands to f and its preconditions hold in s, then

$$\Pi(G, D, Ch, s) = \{\langle \{f\} \cup F', Ch' \rangle | \langle F', Ch' \rangle \in \Pi(G', D, Ch, s)\}$$

(3) If g is a functionality *Id* and the preconditions of its expansion f do not hold in s, then $\Pi(G, D, Ch, s) = \emptyset$.
(4) If g is a method *Id* which expands to the set M according to the domain D, then

$$\begin{aligned}\Pi(G, D, Ch, s) = \{&\Pi(append(B_{m'}, G', s), D, Ch')| \\ &m' \in M \text{ and} Ch' = (Ch \text{ with channels to/from } l \\ &\text{reconnected according to } Iconn_m / Oconn_m) \cup Ch_{m'}\}\end{aligned}$$

The reconnection of channels in Ch for a given m' works as follows. If $\langle m', o, m'', i \rangle \in Ch$ where $\langle o, m_1 \rangle \in Oconn_{m'}$, then this is changed to $\langle m_1, o, m'', i \rangle$. If $\langle m'', o, m', i \rangle \in Ch$ where $\langle i, m_1 \rangle \in Oconn_{m'}$, then this is changed to $\langle m'', o, m_1, i\rangle$.

Lemma 1 *All the configurations in* $\Pi(G, D, Ch, s)$ *are information admissible if: the domain* D *only contains information admissible methods,* $G = (m)$ *(a single method), and* $I_m = \emptyset$*, that is* m *does not need any external input.*

Proof. From the definition of information admissibility in Eq. 1, we have that a configuration is information admissible if and only if each input of each functionality is connected via an adequate channel ($ch \in Ch = (f_{send}, o, f, i)$) to an output of another functionality f_{send} with a compatible specification ($\text{desc}(o) = \text{desc}(i)$ and $\text{dom}(o) = \text{dom}(i)$).

From the definition of information admissible methods we have that each input i of a functionality/method is provided either from a method m_{send} in B_m via an internal channel (m_{send}, o, f, i) or from I_m. In the first case, if m_{send} is a functionality, the condition is satisfied. If m_{send} is a method, then it must also be information admissible and its output o must be connected to a submethod or functionality in the body of m_{send}. Going down through methods, eventually we will reach a functionality f_{send}.

In the second case, that is the input i comes from I_m, then m must be enclosed in some other method m'. m' must also be an information admissible method, with either some method $m_{send} \in B_{m'}$ connected to $i \in I_m$ (case 1) or with i connected to some $i \in I_{m'}$ (case 2). As there is a finite sequence of methods enclosing f, if the second case keeps repeating itself the top method G will eventually be reached, and in G the input of m_{send} will be connected to a method in B_G. $\square$

Lemma 2 *All the configurations in* $\Pi(G, D, Ch, s)$ *are causally admissible.*

Proof. Assume the contrary, that there exist a configuration in $\Pi(G, D, Ch, s)$ that is causally inadmissible. We argue that this case can never occur.

From the definition of causal admissibility in Eq. 2, we have that a configuration is causally admissible if and only if all functionalities in the configuration are applicable in the world state. To have a configuration causally inadmissible, it must be possible to add functionalities f to configurations for which the preconditions $Pr_f(s)$ do not hold. Since the preconditions of all functionalities are known, and since the preconditions of a functionality is verified with the state, before it is added to a configuration (step 1 and 2), there cannot exist any causally inadmissible configurations in $\Pi(G, D, Ch, s)$. Thus, all the configurations in $\Pi(G, D, Ch, s)$ are causally admissible. $\square$

We are now in a position to prove soundness, completeness and optimality for our configuration generation algorithm. We assume that the domain D is closed and finite, and G is a single method Id with no input. For convenience, we repeat the theorems given in Section 5.3.

Theorem 1 (Soundness) *If D only contains information admissible methods, then the configuration algorithm generates only admissible configurations.*

Proof. For a configuration to be admissible, it must be both information and causally admissible. For any given configuration problem P, Lemma 1 and Lemma 2 guarantee that all configurations in $\Pi(P)$ are both information and causally admissible if D only contains information admissible methods. Since the configuration algorithm is a straightforward implementation of the definition of $\Pi(P)$, it follows that it generates only admissible configurations. $\square$

Theorem 2 (Completeness) *The configuration algorithm eventually generates any admissible configuration that is defined by the functionalities and methods in D.*

Proof. By definition, the set of all configurations for a given configuration problem P is given by $\Pi(P)$. The configuration algorithm is a straightforward implementation of the definition of $\Pi(P)$, and it performs a full search since D is finite. Hence, it eventually generates any configuration in $\Pi(P)$. By Lemma 1, this configuration is admissible. Since this it true for any P, we have the thesis. $\square$

Theorem 3 (Optimality) *If the configuration algorithm uses best-first search with branch-and-bound, it returns the admissible configuration with the lowest cost that is defined by the functionalities and methods in D.*

Proof. Follows directly from the completeness of the algorithm in its nondeterministic form, and from the use of branch-and-bound. $\square$