A Self-Configuration Mechanism for Software Components Distributed in an Ecology of Robots

Marco GRITTI [a], Alessandro SAFFIOTTI [a]
[a] *AASS Mobile Robotics Laboratory, University of Örebro, Sweden*

Abstract.
Distributed heterogeneous robotic systems are often organized in component-based software architectures. The strong added value of these systems comes from their potential ability to dynamically self-configure the interactions of their components, in order to adapt to new tasks and unforeseen situations. However, no satisfactory solutions exist to the problem of automatic self-configuration. We propose a self-configuration mechanism where a special component generates, establishes and monitors the system configurations. We illustrate our approach on a distributed robotic system, and show an experiment in which the configuration component dynamically changes the configuration in response to a component failure.

Introduction

Robotic technologies begin to be employed in relatively inexpensive devices and consumer products, and they slowly start to appear in our homes, offices, and workshops. Many observers believe that these devices will be networked and will be able to share their information and coordinate their actions. This vision is becoming rather popular in robotics, and it has been spelled out under different names, like network robot systems [1], intelligent spaces [8], artificial ecosystem [16], and PEIS-Ecology [14]. In this paper, we generally refer to this vision as *ecology of robots*.

The software controlling these distributed robotic systems is usually constituted by components spread over the various devices. The strong added value of this approach comes from the potential of these systems to dynamically *self-configure* the component interactions, both within and across devices. This dramatically increases flexibility and adaptability, since the collaboration patterns are not fixed at design time but can change according to the task that should be performed, or to the current operating conditions. Much work has been done recently on the principles and techniques for automatic self-configuration in autonomous robotics [9,12] as well as in other areas of computer science like ambient intelligence [7], web services [13], or autonomic computing [17]. Despite this, no satisfactory solution exist to the problem of automatic self-configuration.

In this paper we propose a reactive approach to self-configuration, in which a special component dynamically creates, monitors and modifies configurations of a robot ecology. This component accepts task specifications in the form of *configuration templates*. This approach is inspired by ideas from the field of semantic web services and by the heritage of reactive architectures in robotics. The key elements of our approach are:

- *formal descriptions* of the components in the ecology;
- *configuration templates* expressing tasks in the form of desired components and desired component interactions; and
- mechanisms for *discovery* and *matching* of formal descriptions that allow to automatically find the components required by a given template.

In order to illustrate our approach, we apply it to a specific ecology of robots: the PEIS-Ecology. The approach, however, should be applicable to other distributed robotic platforms as well. The rest of this paper is organized as follows. In the next section we state the system models and requirements of our self-configuration mechanism. In Section 2 we give an overview of the constituents of this mechanism. In Section 3 we explain in detail our template-based configuration component. In Section 4 we show an experiment in which our component generates a configuration for a task of cooperative navigation and dynamically changes such configuration in response to a failure.

1. Prerequisites of the System

The self-configuration mechanism proposed here ignores the internal logics of the software components of the system. In other words, all components are internally abstracted as *black boxes*. On the other hand, their external behavior must comply to the model of *asynchronous computation*: components accept the data that are set into their input ports, process them, and export the results by making them available on their output ports.

A uniform model of component interaction is also fixed: components are allowed to interact only through *connections* of input ports to output ports. Such input-output connections must be dynamic, in the sense that they are not to be decided at component design time, but it must be possible to establish or release them at runtime. According to this model, components exchange information in the form of *flows of data* on their input-output connections. This model of interaction is ubiquitous within robotic systems, e.g., in a sensor processing pipeline, or in a hierarchy of controllers. Such model also appears in other domains, e.g. multi-media content streaming through the Internet.

A convenient way to make all system components comply to the same model of interaction is to build them on top of a common *middleware*. In the present work, the PEIS-kernel [3] was adopted as middleware, but the achieved results do not depend on it. For the purposes of our self-configuration mechanism, it is sufficient that the middleware has an API that supports (1) asynchronous access to the component functions, and (2) dynamic connections of input ports to output ports. These two requirements are generally met by most of the existing middleware, being usually implemented through mechanisms of subscription to asynchronous events and of event notification. The self-configuration mechanism described below applies to any distributed system that satisfies the above prerequisites. The PEIS-Ecology [14] belongs to this class of systems.

2. Self-Configuration

The overall behavior of a distributed system is determined by the concurrent execution of the various components, which combine together their functionalities through specific interactions. We intend such concurrent executions organized in *system configurations*. A *system configuration* for a task t is a pair $<B, S>$, where:

- B is the set of running components whose functionalities contribute to t;
- S is the set of all the connections established between the components in B.

2.1. An Architecture Pattern for Self-Configuration

The problem of configuring a robot ecology consists in determining what components are eligible to execute in a common system-level task, and what are their suitable connections. In the terms of the above definition, this means to determine the two sets B and S opportune to solve a given task t. This computation can be done automatically, according to the architecture and interaction schemes of Figure 1 and Figure 2.

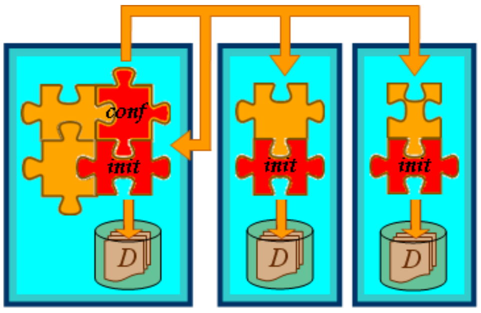

Figure 1. System architecture for self-configuration.

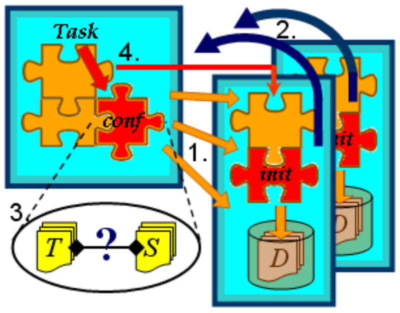

Figure 2. The self-configuration process.

As illustrated, repositories of component descriptions D should be deployed in the system. Special *init*-components should run on the same sites of the repositories, which fetch and propagate the descriptions. Other special *conf*-components (or *configurators*) should exist, capable of computing the configurations opportune to achieve the various system-level tasks using the information stored in the component descriptions.

2.2. Formal Component Descriptions

The formal component descriptions adopted here are called component *profiles*. These describe the components in both their *functional* and *non-functional* aspects.

The functional part of a profile lists the signals accepted in input and the signals produced in output. These are in one-to-one correspondence respectively with the component input and output ports. As an example, consider the advertisements in Figure 3 and in Figure 4. These publish the profiles of a navigation control component for mobile robots and of a motor drive component. The navigation control component has three input signals (named `sonar`, `position`, and `localize`) and produces one output signal (named `set.velocity`). The motor drive component accepts just one input signal and has no output signals. All signals are typed.

The non-functional part of a component profile includes the component type, and the list of component properties. Component types classify components according to the functionality they provide. Component properties declare additional features of the components, which allow to further specialize their description. For example, the type of the navigation component is `NavigationControl`; the type of the motor drive is `MotorDrive`. The motor drive has also a property, with type `Support` and value `ASTRID`. This property specifies the real object that carries the actuator to which such component is a bare interface. In this case such object is Astrid, a mobile robot.

Component profiles are stored into manifest files referred to as *advertisements*. We deploy the advertisements on the same sites of the component instances. This is a typical solution for allowing automatic component discovery in peer-to-peer frameworks.

```
<component>NavigationControl</component>
<compName>"thinkingcap"</compName>
<componentDescription>
  "A fuzzy logic behavior-based mobile
   robot control architecture."
</componentDescription>
<parameter>
  <paramName>"contr.b-goal"</paramName>
  <DataType>B-Goal</DataType>
</parameter>
<signalInput>
  <sigInName>"sonar"</sigInName>
  <DataType>DistanceArray</DataType>
</signalInput>
<signalInput>
  <sigInName>"position"</sigInName>
  <DataType>EncoderReading</DataType>
</signalInput>
<signalInput>
  <sigInName>"localize"</sigInName>
  <DataType>PlanarPose2D</DataType>
</signalInput>
<signalOutput>
  <sigOutName>"set.velocity"</sigOutName>
  <DataType>PlanarVel</DataType>
</signalOutput>
```

Figure 3. Advertisement of a navigation component.

```
<component>MotorDrive</component>
<compName>"driveastrid"</compName>
<componentDescription>
  "The drive controller of Astrid."
</componentDescription>
<property>
  <PropType>Support</PropType>
  <propValue>"ASTRID"</propValue>
</property>
<signalInput>
  <sigInName>"setpoint"</sigInName>
  <DataType>PlanarVel</DataType>
</signalInput>
```

Figure 4. Advertisement of the drive on Astrid.

```
<component>PositionSensor</component>
<compName>"persontracker"</compName>
<componentDescription>
  "A people detection and tracking
   system using stereo vision."
</componentDescription>
<signalOutput>
  <sigOutName>"person.pos"</sigOutName>
  <DataType>PlanarPose2D</DataType>
</signalOutput>
```

Figure 5. Advertisement of a tracker component.

2.3. Use of Component Descriptions

In our mechanism for automatic self-configuration, the component profiles are processed by the special *conf*-components for (1) selection of the components eligible to enter a configuration, and (2) check of mutual compatibility between components supposed to interact. These computations are done in compliance with the following rules:

Component selection rule: a component of profile p, is *eligible* to enter a system configuration if and only if this configuration needs a component b such that (i) the type of p matches the type of b, and (ii) for all properties r_i^p of p, if b has a property r_j^b such that the type of r_i^p matches the type of r_j^b, then their values are equal. In such case, it is also said that p is *compatible* with b.

Component mutual compatibility rule: given a set of components with profiles P, of which one, named p^{si}, represents a *sink* of data, and all the others, grouped in the set $P^{so} = P \setminus \{p^{si}\}$, represent *sources* of data, the sink profile is *mutually compatible* against all of its sources if and only if for each input i^{si} of the sink there is at least one output o^{so} among the outputs of the sources such that o^{so} matches i^{si}.

All the above matchings are purely *syntactic*: if the types of the compared quantities are equal, then the quantities match, otherwise they do not match.

2.4. The Four Acts of Self-Configuration

As depicted in Figure 2, having received a system-level task specification, the *conf*-component broadcasts queries (Act 1) that ask for components suitable for such task; then it collects relevant component descriptions received in reply from the various *init*-components (Act 2). These two acts constitute the *discovery* phase of the self-configuration process. After a query timeout, the *conf*-component evaluates the feasibility of the desired task given the descriptions it has received (Act 3). This is the *composition* phase.

In this phase the *conf*-component *searches* among all possible configurations featuring (some of) the discovered components for one that can solve the assigned task. If such a configuration is found, the *conf*-component deploys it into the system (Act 4). This is done by (i) activating all the components that have been selected, and (ii) establishing all the connections that have been computed.

3. The Reactive Configuration Component

The *conf*-components are the key elements of our self-configuration mechanism. We now describe the implementation of one sort of *conf*-component: the *reactive configurator*. Upon receiving in input a system-level task specification, the reactive configurator is able to configure the system in a way appropriate to achieve such task, and to re-configure it whenever a failure is declared by some component of the deployed configuration.

3.1. Template Configurations as System-Level Task Specifications

We specify a task by listing what *types* of components interact, what are the *properties* that those components must fulfill to be able to correctly perform their subtask, and finally what are the *interactions* according to which it is meaningful for these components to exchange data. Such kind of specification is a *template configuration*.

For example, consider the task of *playing the radio* in a given room. A template configuration for it could list as types of interacting components a radio tuner, and (at least) one speaker. The speaker component must comply with the property to have *place* in the desired room. The desired interaction is that the tuner should feed the speaker, and not *vice versa*. As another example, consider the task of *navigating a robot* in the free space. Its template configuration could list a position sensor (possibly a virtual sensor), a drive for the motors of the robot, and a navigation control component. It is necessary that the motor drive has the robot that should move as its *support*. If the position sensor is proprioceptive, it must have the same robot as its *support*. The desired interactions are the following: the position sensor should feed the navigation control component; the navigation control component, in turn, should feed the motor drive.

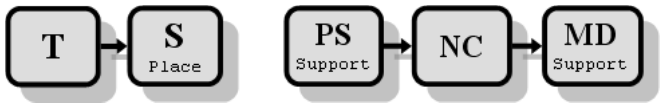

Figure 6. The template configurations for *playing the radio* (left) and for *navigating a robot* (right).

Figure 6 shows two possible template configurations for those tasks. The properties of the components become task-dependent *parameters* of the template configurations. For instance, the template of Figure 6 (right) can be parameterized by setting "*support* = `ASTRID`". Fully parameterized template configurations t are the input of the reactive configurator, whose functioning is illustrated in the following.

3.2. The Algorithm for Configuration Generation

The reactive configurator performs component discovery in a trivial way. Having received a system-level task specification in the form of a parameterized template configuration t, it broadcasts a generic query for discovering all the components of the system. The set P of all received component profiles identifies the set of all components currently available. At the beginning of the composition phase, the reactive configurator determines the set $F \subseteq P$ of profiles of the components which are eligible to enter a configuration for t. This is done applying the component selection rule of Section 2.3 to all the profiles in P, against all the components B^t listed in the template t. More precisely, the configurator computes a partition $F^1 \dots F^n$ of F such that for all $b_i \in B^t$, F^i contains all and only those profiles that are compatible with b_i.

Consider for example the task of *navigating* `ASTRID`. If the configurator receives the advertisement of a motor drive with "*support* = `PIPPI`", this will be immediately discarded applying the selection rule, because this component has a support property different from the one that is required. The selection rule will also make the configurator discard all the advertisements of the components whose types are not `PositionSensor`, `MotorDrive`, or `NavigationControl`. Discarding components because of type or property mismatch is a preliminary heuristic cut of the configuration search space.

Algorithm 1 CONSEARCH$(t, F^1, \dots, F^n)$

```
Require: Template $t$, and $n$ profile sets.
Ensure: A configuration $\tilde{c}$ or FAIL.
  $\tilde{c}_0 \leftarrow \emptyset, C \leftarrow \{\tilde{c}_0\}$
  while $C \neq \emptyset$ do
    $\tilde{c} \leftarrow first(C)$
    if $|P^{\tilde{c}}| \equiv |B^t|$ then
      return $\tilde{c}$
    else
      $X \leftarrow$ EXPAND$(\tilde{c}, t, F^1, \dots, F^n)$
      $C \leftarrow rest(C) \cup X$
    end if
  end while
  return FAIL
```

Function 2 EXPAND$(\tilde{c}, t, F^1, \dots, F^n)$

```
Require: Config. $\tilde{c}$; $t$, and $F^i$ as before.
Ensure: All one-level expansions of $\tilde{c}$.
  $X \leftarrow \emptyset$
  $U \leftarrow \{b_i \in B^t \mid \forall p \in F^i$ is $p \notin P^{\tilde{c}}\}$
  for all $b_i \in U$ do
    for all $p \in F^i$ do
      if $local\text{-}compatibility(p, \tilde{c}, t)$ then
        $X \leftarrow X \cup \{\tilde{c} \oplus p\}$
      end if
    end for
  end for
  return $X$
```

Given the profiles of the eligible components, partitioned in $F^1, \dots, F^n$, the configurator *searches* in the space of all their possible configurations for one that complies to the assigned template t. A *state* in this search problem is an internal representation $\tilde{c}$ of a *partial configuration*, where just some of the components in B^t are associated to some profiles of F, and just some of the interactions listed in t are associated to groups of input-output connections. The only available *action* of this search problem is to add to a partial configuration $\tilde{c}$ a new association between a profile $p \in F$ and a template component $b \in B^t$ that is compatible with it. This is denoted by the operation $\oplus$, defined among internal representations of configurations and component profiles. Algorithm 1 is the search algorithm implemented in the reactive configurator. It accepts in input a parameterized template configuration t, and the various $F^1, \dots, F^n$. Algorithm 1 implements a sort of uninformed search, where exploration is done expanding internal representations $\tilde{c}$ of partial configurations, grouped in a working set C.

Function 2 computes the set X of expansions of a partial configuration $\tilde{c}$. The function computes the set U of components of t for which no compatible profile is already in the set $P^{\tilde{c}}$ of the profiles already in $\tilde{c}$. For all the components $b_i \in U$, this function tries to expand $\tilde{c}$ with any of the profiles $p \in F^i$, i.e. the ones compatible with b_i. Each expanded configuration is obtained adding to $\tilde{c}$ a profile p through the operator $\oplus$. The expansion step is done in compliance with the following rule:

Local compatibility rule: a profile $p \in F^i$ can be associated to a component $b_i \in U$ if and only if (i) if all the components $b^f \in B^t$ which are *feeders of* b_i are already associated in $P^{\tilde{c}}$, then p must be mutually compatible against all the profiles that are associated to all the b^f; and (ii) for all the components $b^F \in B^t$ which are *fed by* b_i, if some b^F has all its other feeders already associated in $P^{\tilde{c}}$, then the profile associated to b^F must be mutually compatible against the set of profiles constituted by p plus the profiles associated to all the other feeders of b^F.

Mutual compatibility between component profiles is always tested according to the mutual compatibility rule of Section 2.3. When the mutual compatibility test fails, the local compatibility fails too, and the new profile is not added to the partial configuration. Otherwise, all the input-output connections between the tested profiles are added to the partial configuration. The information contained in the template configurations about what components have direct interactions with what other components permits another important heuristic cut of the search space. Without such information, the local compatibility test would explode in complexity, because at every expansion step the search algorithm would have to check the compatibility of the newly introduced profile against all the subsets of $P^{\tilde{c}}$, in all their possible interaction patterns.

3.3. Monitoring and Reconfiguration

Once a configuration has been generated and deployed, the configurator component starts to *monitor* the configuration execution. A configurator can monitor clean failures of the deployed components by connecting to a special output `FAIL` that all the components are supposed to export. Whenever a failure event is signaled by a component, the configurator erases its profile from the set of the discovered profiles and re-triggers the search algorithm to look for another system configuration that can solve the original task. This monitoring phase is intrinsically reactive.

4. Experimental Run

The experiment that follows was performed on a real PEIS-Ecology platform [3,14].

4.1. Setup of the Experiment

The experiment takes place in the PEIS-Home: a reconstruction of a small apartment of about 25 m^2. The living room of the PEIS-Home is under the field of view of a 3D stereo camera. A PeopleBot named Astrid (Figure 8) is present in the environment.

The assigned task is *moving* `Astrid` *to the* `bedroom`. This is specified through the parameterized template configuration of Figure 7. Five components are listed in the template. Their types are: `SonarArray`, `Encoder`, `PositionSensor`, `NavigationControl`, and `MotorDrive`. For each component, the template lists its actual parameters[1], the required properties, and the interactions in which it takes part. The navigation control component is given one parameter of type `B-Goal`. For all the

[1]In our complete framework, besides the input signals, components can also accept *operational parameters*, which are for setting or tuning their running mode. This increases the components flexibility of use.

```
                  *** components ***

("Comp;#SonarArray"       () (("Objects;#Support" "ASTRID"))     (0))
("Comp;#Encoder"          () (("Objects;#Support" "ASTRID"))     (1))
("Comp;#PositionSensor"   () (("Objects;#Support" "ASTRID"))     (2))
("Comp;#NavigationControl" (("Cpt;#B-Goal" "(AT BEDROOM)")) () (0 1 2 3))
("Comp;#MotorDrive"       () (("Objects;#Support" "ASTRID"))     (3))

                 *** interactions ***
(0 3) (1 3)    ; sonar->control    / encoder->control
(2 3) (3 4)    ; position->control / control->drive
```

Figure 7. Parameterized template configuration for the navigation task of the experiment.

other components the `Support` property is specified. If present in their advertisements, this property must be equal to `ASTRID`. The interactions meaningful in the task are listed at the end of the template: component number 3 (the `NavigationControl`) should be fed by all other components, and it should feed component number 4 (the `MotorDrive`). This template is an extension of the one in Figure 6 (right).

The sonar array, encoder, and drive are part of the equipment of Astrid. They are all grounded on the same software. This is based on Player [6]. The advertisement of the motor drive was shown in Figure 4. The other two are similar: all expose a `Support` property with value `ASTRID`. The advertisement of the navigation component was shown in Figure 3. The corresponding robot navigation software, called Thinking Cap [15], accepts high-level navigation goals in the form of logical propositions, called *behavioral goals* (or B-Goals). Two distinct localization systems are available for the role of position sensor. One of them is the person tracker advertised in Figure 5. The person tracker is a people detection and tracking system [10] that uses the 3D camera of the PEIS-Home. This system is also able to track Astrid. The other position sensor is a factitious component that just transforms the value of the odometry from the Player on Astrid into the absolute coordinates of the PEIS-Home. From the functional point of view the two position sensors are equivalent: they both issue a single output of type `PlanarPose2D`.

4.2. Execution of the Experiment

Configuration: After receiving the template of Figure 7, the reactive configurator queries the ecology, and receives the advertisements of all the components available at the moment, among which are the six components discussed above. These six components are the only eligible to enter the configuration. The robot sonars, encoder, and drive match the types and properties required by the template. The Thinking Cap is advertised as a `NavigationControl` component, precisely as needed. The person tracker and the absolute odometry are `PositionSensor`. They do not expose a `Support` property, hence this is not checked for them. During configuration search, all template components but the position sensor have just one compatible profile. In the mutual compatibility tests following the various expansions of the search algorithm, the drive component is verified to be mutually compatible with the Thinking Cap. In turn, the Thinking Cap is verified to be mutually compatible with the set of components constituted by the sonars, the encoders, and the person tracker. The generated configuration features the person tracker, which was the first position sensor to be tested.

Execution: The generated configuration is deployed. The snapshot of Figure 9 was taken with a system inspection tool just after the deployment was complete. Apart from

Figure 8. Astrid.

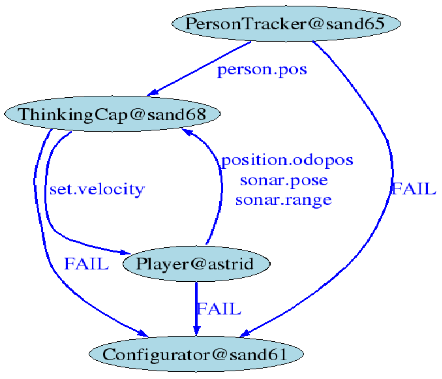

Figure 9. The deployed configuration.

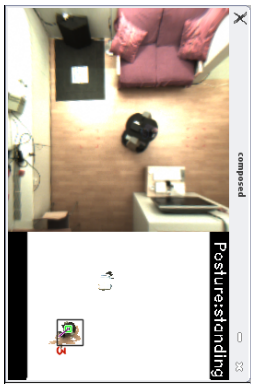

Figure 10. Tracker view.

the connections between the task-related components, the snapshot shows also that the reactive configurator has connected itself to all the needed failure signals. After activating all the components and establishing all the connections, the configurator injects the B-Goal into the Thinking Cap. The Thinking Cap starts sending velocity setpoints to the drive of Astrid. Astrid starts moving. The person tracker extracts the visual signature of Astrid from the images received from the stereo camera (Figure 10), and transmits to the Thinking Cap the absolute position of Astrid in global coordinates. The navigation of Astrid proceeds smoothly until Astrid exits the field of view of the stereo camera. When that happens, the person tracker looses the robot, and raises its `FAIL` signal.

Reconfiguration: Upon receiving the failure signal, the configurator turns off the person tracker and removes its profile from the set of discovered components. Then it dismantles the deployed configuration. Astrid stops, because its drive is not receiving any more the velocity setpoint updates. The configurator starts from scratch another search for a new configuration. During the new configuration search, the configurator builds up the same associations as before, except for the position sensor, which is now associated to the absolute odometry component. As soon as the new subscriptions are deployed, Astrid resumes its course toward the bedroom.

5. Conclusions

The main contribution of this paper is to propose a task-driven, reactive approach to self-configuration for the software of distributed robot systems. We have shown the applicability of this approach to the PEIS-Ecology, but this approach should be applicable to any type of distributed robot system. The most notable benefit of our approach is to make the system more flexible and adaptive, by giving it the ability to automatically self-configure for a given task, and to automatically re-configure in case of failures.

Although other approaches have been proposed in literature for formal description and discovery of software components in distributed robotic systems [2,4,5,11], very few works exist that deal with the automatic, run-time composition of these components for a given task [9,12]. Contrary to the latter works, the approach presented in this paper is reactive. Two aspects of reactivity of our approach are the locality of configuration search in the neighborhood defined by the template, and the immediate return of the first solution found. As consequence, our approach can scale well with the size of the robot ecology, but it cannot guarantee optimality of the solution. Global planning approaches can overcome this problem, but have difficulties to scale-up. In the future, we intend to explore a hybrid global/local approach taking inspiration from distributed planning.

Our approach is inspired by ideas coming from the field of semantic web services, e.g. OWL-S. It should be emphasized, however, that one could not directly abstract components of a robot ecology as services of OWL-S. Web Services are similar to method invocations: they accept input parameters and issue a return value after a certain time. By contrast, typical robotic components are continuous processes that process and produce continuous flows of data. An important contribution of our work is to have devised a novel framework of formal descriptions tailored for components typical of robot ecologies and of distributed robotic systems of similar nature.

Acknowledgments

This work has been supported by the Swedish Research Council (Vetenskapsrådet), and by ETRI (Electronics and Telecommunications Research Institute, Korea). Our thanks to Mathias Broxvall for valuable discussions and technical help.

References

[1] T. Akimoto and N. Hagita. Introduction to a network robot system. In *Proceedings of the 2006 International Symposium on Intelligent Signal Processing and Communications*, Tottori, Japan, 2006.

[2] F. Amigoni and M. Arrigoni Neri. An application of ontology technologies to robotic agents. In *Proceedings of the International Conference on Intelligent Agent Technology*, Compiègne, France, 2005.

[3] M. Broxvall. A middleware for ecologies of robotic devices. In *Proceedings of the First International Conference on Robot Communication and Coordination*, Athens, Greece, 2007.

[4] L. Chaimowicz, A. Cowley, V. Sabella, and C. J. Taylor. ROCI: A distributed framework for multi-robot perception and control. In *Proceedings of IROS 2003*, Las Vegas, Nevada, USA, 2003.

[5] J. Fritsch, M. Kleinehagenbrock, A. Haasch, S. Wrede, and G. Sagerer. A flexible infrastructure for the development of a robot companion with extensible HRI-capabilities. In *ICRA'05*, Barcelona, 2005.

[6] B. Gerkey, R. Vaughan, K. Sty, A. Howard, G. Sukhatme, and M. Mataric. Most valuable player: A robot device server for distributed control. In *Proceedings of IROS 2001*, Wailea, Hawaii, USA, 2001.

[7] A. Kameas, I. Bellis, I. Mavrommati, K. Delaney, M. Colley, and A. Pounds-Cornish. An architecture that treats everyday objects as communicating tangible components. In *Proc. of PerCom'03*, 2003.

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

[9] R. Lundh, L. Karlsson, and A. Saffiotti. Plan-based configuration of an ecology of robots. In *Proceedings of the 2007 IEEE International Conference on Robotics and Automation*, Roma, Italy, 2007.

[10] R. Muñoz-Salinas, E. Aguirre, and M. García-Silvente. People detection and tracking using stereo vision and color. *Image and Vision Computing*, 25(6):995–1007, 2007.

[11] H. Noguchi, T. Mori, and T. Sato. Automatic generation and connection of program components based on rdf sensor description in network middleware. In *Proceedings of IROS 2006*, Beijing, China, 2006.

[12] L. Parker and F. Tang. Building multi-robot coalitions through automated task solution synthesis. *Proceedings of the IEEE*, 94(7):1289–1305, 2006.

[13] J. Rao and X. Su. A survey of automated web service composition methods. In *Proceedings of the ICWS'2004 International Workshop on SWSWPC*, San Diego, California, USA, 2004.

[14] A. Saffiotti and M. Broxvall. PEIS Ecologies: Ambient Intelligence meets Autonomous Robotics. In *Proceedings of Smart Objects & Ambient Intelligence (sOc-EUSAI 2005)*, Grenoble, France, 2005.

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

[16] A. Sgorbissa and R. Zaccaria. The artificial ecosystem: a distributed approach to service robotics. In *Proceedings of ICRA'04*, New Orleans, Louisiana, USA, 2004.

[17] G. Tesauro, D. Chess, W. Walsh, R. Das, A. Segal, I. Whalley, J. Kephart, and S. White. A multi-agent systems approach to autonomic computing. In *Proceedings of AAMAS'04*, New York City, USA, 2004.