Reasoning on Data Streams: An Approach to Adaptation in Pervasive Systems

Nicola Bicocchi[1], Emil Vassev[2], Franco Zambonelli[1], and Mike Hinchey[2](✉)

[1] Universita di Modena e Reggio Emilia, Modena, Italia
{nicola.bicocchi,franco.zambonelli}@unimore.it
[2] Lero–the Irish Software Engineering Research Centre, University of Limerick, Limerick, Ireland
{emil.vassev,mike.hinchey}@lero.ie

Abstract. Urban environments are increasingly invaded by devices that acquire sensor information and pave the way for innovative forms of context awareness. Collecting knowledge from loosely-structured data streams and reasoning about changes are two key elements of the process. This paper illustrates a possible way to combine these two elements in a coordinated way. We make use of a recently-developed framework for classifying data streams with service-oriented, reconfigurable components. Furthermore, we embed the KnowLang Reasoner, allowing logical and statistical reasoning on the acquired knowledge aiming to achieve self-adaptation.

1 Introduction

The widespread adoption of sensor networks, actuators and computational resources capable of interacting with people is transforming urban environments as well as domestic spaces [1,3,6]. However, the design of such systems presents challenges for current approaches. Designing with a top-down approach means that all the requirements of a software architecture have to be taken into account *a priori*. Systems engineered in this way have a predictable and measurable behaviour but are not well suited to cope with dynamic execution contexts. On the other hand, bottom-up design delivers robust systems that can eventually be used in pervasive environments. However, modelling system behaviour of such pervasive systems is not a trivial task and potential urban scenarios call for a balanced trade-off between the two approaches.

Situational awareness appears to be one of the key drivers that guide this trade-off. In fact, it can be used to provide systems with adaptation capabilities — essential in dynamic, interconnected, and yet, heterogeneous environments — without compromising predictable behaviours. For example, it would be possible to envision a system capable of continuously inferring its operating context and executing actions accordingly. Increasing both the number of inferred contexts and possible actions leads to seemingly-adaptive systems [2].

This paper illustrates how situational awareness and reasoning can be put to work together in order to implement adaptive behaviours. For awareness collection, we have made use of a recently-developed framework [4], centered around

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2015
P.C. Vinh et al. (Eds.): ICTCC 2014, LNICST 144, pp. 23–32, 2015.
DOI: 10.1007/978-3-319-15392-6_3

the concept of dynamic service reconfiguration, which is able to gather data from a number of different sources, and classify them using general-purpose algorithms. For reasoning, instead, we used the KnowLang framework [10]. The combination of these tools allows systems to collect data on situational awareness, reason about it, and select the most proper chain of actions accordingly in a closed loop fashion (see Figure 1). As an example, we discuss a case study describing the implementation of a self-driving, self-adapting drone. Situational awareness data is collected via various sensors, classified and provided to the KnowLang Reasoner to select the appropriate actions.

The rest of this paper is organised as follows. Section 2 presents our approach to data collection and classification. Section 3 provides a brief introduction to KnowLang along with a short discussion on how it can cope with the collected data. Then, in Section 4 we present a small proof-of-concept case study. Finally, Section 5 provides concluding remarks and a summary of our future goals.

2 Knowledge Collection and Understanding

In this section the framework for knowledge collection is described. It is based on reconfigurable components and its goal is to provide a starting point for many diverse applications. Developers are only required to select the required modules, and define their topology and reconfiguration strategies as depicted in Figure 1. It has been conceived around the following requirements [12] [7].

- *Adaptation.* The framework should be the key source of the applications' adaptive capabilities. It has to provide the mechanisms and tools necessary for knowledge processing. Applications relying on the framework must receive compact and structured information about the environment and use this as triggers for adaptation.
- *Self-Adaptation.* Given a specific classification task and a situation, the framework should select the most appropriate components. For example, it is possible to roughly recognise the vehicle used by a user with either GPS or an accelerometer or microphone. An energy constrained system could constantly monitor its residual energy and select the most appropriate trade-off between consumption and classification accuracy. Service-oriented and dynamically reconfigurable components have been recently proposed. They allow us to select among different components (i.e., sensors, classifiers) depending on the situation. Furthermore, reconfigurable components can transparently modify their internal parameters. For example, classifiers can analyse temporal windows of different sizes considering the availability of computational resources or energy boundaries.
- *Extensibility.* The framework should be extensible in several ways, without the need to restart it. First, it should be possible to deploy, modify, and remove context services. Second, the infrastructure should support the evolution of supported types of contexts by dynamic load context definitions, functionality, and acquisition mechanisms.

- *Modelling abstractions.* The type of situational information that is relevant for modelling and handling varies across application settings. For example, in a hospital, items like beds, pill-containers, and medicines are important information for the work of clinicians, but this is specific to hospitals. Hence the application programmer should be able to model and handle context data specific to various settings.
- *Software Engineering.* Organising the framework around the idea of reconfigurable components (i.e., sensors, classifiers) leads to modularity and composability of the software ecosystem. Developers are allowed to deploy components that are either: *(i)* already included in the framework or *(ii)* developed by third parties.
- *Event-based.* The core quality of situation-aware applications is their ability to react to changes in their environment. Hence, applications should be able to subscribe to relevant events and be notified when such events occur.

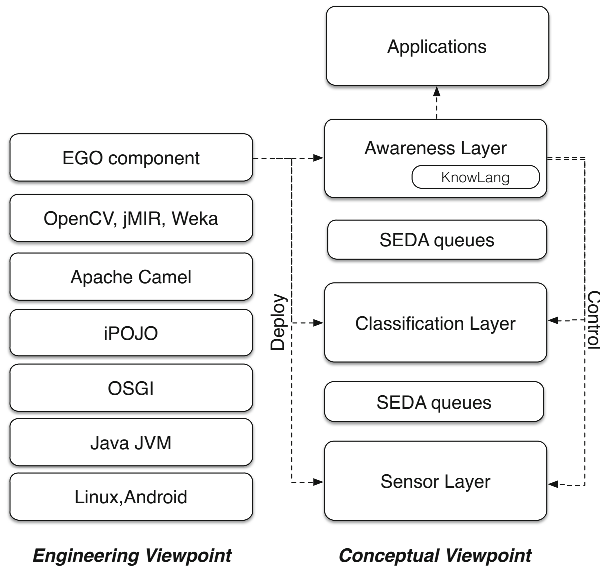

Fig. 1. The framework architecture is structured around three layers, namely *sensor*, *classifier* and *awareness* layers

The architecture is structured around three layers, namely the *sensor*, *classifier* and *awareness* layers. Each layer can host multiple modules connected to each other via application-definable topologies. The data flow from sensors through the whole architecture is by means of in-memory queues enabling modules decoupling and many-to-many asynchronous communications. Each layer can host multiple modules.

The sensor layer hosts modules that are responsible for retrieving raw data from physical sensors and pre-processing them. An example could be a module

acquiring images from a camera and cropping and resizing them. At the time of writing, we have already implemented modules for reading data from Android devices.

The classification layer hosts modules that consume data coming from the sensor layer and classify them (i.e., generate semantically richer information). An example could be a module able to classify the activity performed by a user by processing accelerometer data. At the time of writing, we have implemented modules for classifying user activity, location, speed, and vehicle used on the basis of common smartphone sensors. It is worth noting that our goal is to build a general-purpose awareness framework that can be used as a common basis for both research and application development, not to solve every possible classification problem. Specific applications will need their own modules to be developed.

The awareness layer hosts modules consuming labels produced in the classification layer and feeding external applications with situational information. These modules might have different goals depending on the application. However, they can be divided into two main classes. The former comprises modules delegated to sensor fusion processes. These modules receive labels, eventually conflicting, coming from multiple classification modules and apply algorithms to achieve higher semantic levels. For example, common-sense knowledge has been recently proposed [5] and could be integrated at this level. The latter, instead, is related with the capability of the framework of monitoring and controlling itself. In a sense, the awareness layer could be the key to building a *self-aware* awareness module. For example, it would be possible within this level to integrate modules observing the internal status of the framework and activating different classifiers and sensors depending on operating conditions. This capability could be used to achieve both improved classification accuracies and reduced power consumption levels by continuously selecting the most suitable classifiers and sensors. In this work, we embedded the KnowLang reasoner within this layer. It receives labels from the classification layer and selects the arriving actions.

From an engineering viewpoint, the architecture is implemented on top of industrial-level Java technologies. Each module is actually an OSGi component able to meet the requirements mentioned above. On top of OSGi, we have an iPOJO layer. iPOJO is a container-based framework handling the lifecycle of *Plain Old Java Objects (POJOs)* and supporting management facilities such as dynamic dependency handling, component reconfiguration, component factory, and introspection. Moreover, the iPOJO container is easily extensible and allows pluggable handlers, typically for the management of non-functional aspects. On top of the iPOJO framework we build the support for the staged and layered architecture by making use of Apache Camel. This framework provides components with the capability of asynchronously processing data streams and communicating through in-memory queues. These queues allow modules belonging to different layers to continuously communicate with each other with minimum hardware requirements. Considering that pattern recognition has a central role in situation awareness, we wrapped well-known data manipulation libraries within the framework such as Weka, OpenCV, and jMIR.

3 KnowLang

KnowLang [10] is a formal language dedicated to knowledge representation for self-adaptive systems. The language implies a multi-tier specification model allowing for integration of ontologies together with rules and Bayesian networks [8]. The language aims at efficient and comprehensive knowledge structuring and awareness based on logical and statistical reasoning coping with the non-deterministic behaviour of self-adaptive systems by handling uncertain knowledge via additive probabilities used to represent degrees of belief. With KnowLang, we build a knowledge base (KB) with a variety of knowledge structures such as *ontologies*, *facts*, *rules* and *constraints*. The KnowLang ontologies are composed of hierarchically organised *concepts* and *objects*. Moreover, concepts and objects may be additionally related via *relations*. Relations are binary, i.e., connect two concepts, two objects, or an object with a concept, and may have probability-distribution attributes (e.g., over time, over situations, etc.). The relations can be expressed graphically as *concept maps* (see Figure 2). Probability distribution is provided to support probabilistic reasoning and by specifying relations with probability distributions, we actually specify Bayesian networks connecting the concepts and objects of an ontology.

Figure 2 shows a KnowLang specification sample demonstrating both the language syntax [9] and its visual counterpart — a concept map based on interrelations with no probability distributions.

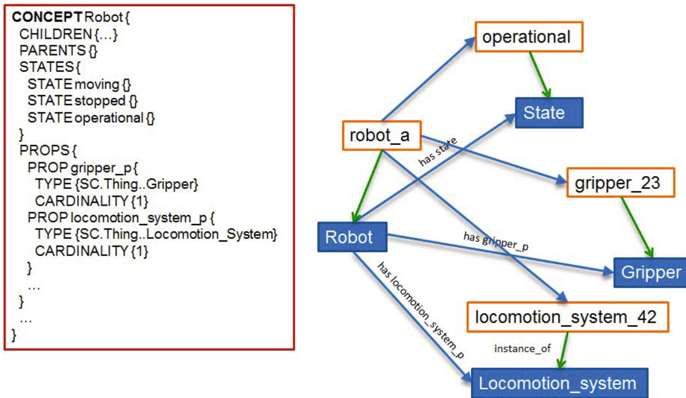

Fig. 2. KnowLang Specification Sample

For reasoning purposes, every concept specified with KnowLang, has an intrinsic $STATES$ attribute (see Figure 2) that may be associated with a set of possible states that a concept's instances may be in. In general, a concept may occupy a new state when *concept properties* have been changed or some events or actions have occurred in the system or environment. KnowLang employs special knowledge structures and a reasoning mechanism for modelling self-adaptive behaviour [11]. Such a behaviour can be expressed via special *policies*, *events*,

actions, *situations*, and *relations* between policies and situations. Policies (Π) are at the core of self-adaptive behavior. A policy π has a *goal* (g), *policy situations* (Si_π), *policy-situation relations* (R_π), and *policy conditions* (N_π) mapped to *policy actions* (A_π) where the evaluation of N_π may eventually (with some degree of probability) imply the evaluation of actions (formally denoted with $N_\pi \xrightarrow{[Z]} A_\pi$). A condition is a Boolean expression over the ontology, e.g., the occurrence of a certain event. A goal g is a desirable transition to a state, or from a specific state, to another state (formally denoted with $s \Rightarrow s'$). A state s is a Boolean expression over ontology ($be(O)$). Ideally, KnowLang policies are specified to handle specific situations, which may trigger the application of policies. A policy exhibits a behaviour via actions generated in the environment or in the system itself. Specific conditions determine which specific actions (among the actions associated with that policy) will be executed. When a policy is applied, it checks what particular conditions are met and performs the mapped actions. An optional probability distribution may additionally restrict the action execution. Although initially specified, the probability distribution is recomputed after the execution of any involved action. The re-computation is based on the consequences of the action execution, which allows for *reinforcement learning*.

4 Case Study

To demonstrate our approach, we describe an example of a self-aware surveillance drone designed for detecting people within specific areas of interest (see Figure 3). The system has the goal to collect sensor data, classify the data, and

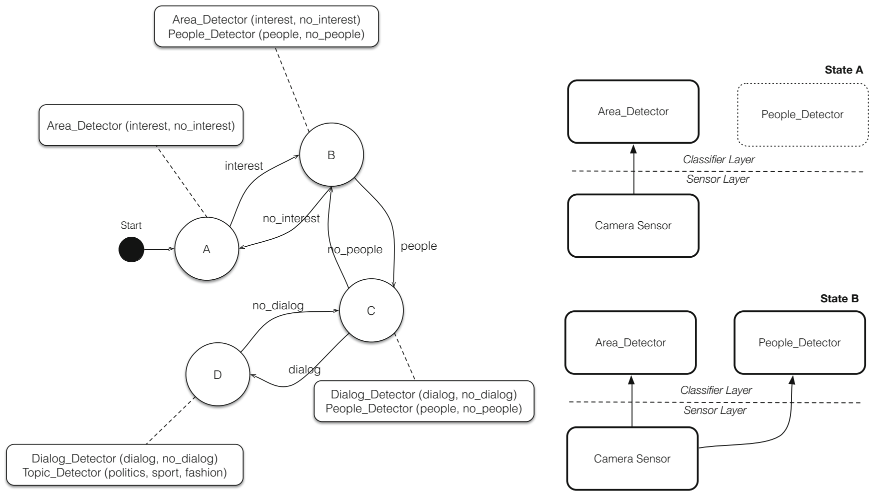

Fig. 3. A self-aware surveillance drone designed for detecting people within specific areas of interest

define the situation the data are immersed in. However, instead of using a single complex classifier, we provide developers a means of using a number of simpler and more specific classifiers. These modules can be enabled, disabled, wired and rewired in a dynamic way by making use of their output to navigate the state automata. Each status has a name and is associated with a set of classifiers — and associated sensors — that have to be active and a set of possible transitions. Each time the output of an active classifier changes, a reconfiguration is applied. Needed modules are deployed and inactive ones are automatically removed to reach the new status. In this way, the overall problem of situation recognition is modularised in a way similar to the way we believe our brain works. Each node embeds the knowledge acquired by the former and activates more specific classifiers to collect further details.

Figure 3, drives the reconfiguration of a surveillance drone. State A is activated as soon as the drone takes off and tries to detect areas of interests. As soon as an area of interest is spotted, state B is activated and eventual people are detected. State C, activated only when people are detected in an area of interest analyses audio signals to detect dialogs. Finally, state D, refines state C by inferring the general topic of the conversation using common sense knowledge and speech recognition techniques.

It is worth noting that this example shows the internal logic of the awareness module of two different applications. However, despite the fact that the logic used to collect situational awareness has to be linked with the application logic, these automata are agnostic about *how* the situational knowledge is actually collected. In fact one could completely change sensors and classifiers used in each and every state without altering the application logic. We think this feature could both: *(i)* sensibly speed up the prototyping of pervasive applications and *(ii)* help in the development of pattern recognition modules. If fact, one could quickly assess different algorithms, libraries, and approaches without altering anything within the actual application. For the purpose of this case study, we used KnowLang to support the behaviour outlined above. The first step was to specify a simple knowledge base (KB) representing the domain outlined in the case study, e.g., the drone itself and the drone's operational environment with entities such as areas of interest, people, drone base, etc. Recall that this domain is described via a domain ontology expressed through domain-relevant concepts and objects (concept instances) related through relations (see Section 3). To handle explicit concepts such as situations, goals, and policies, we gave some of the domain concepts explicit state expressions. The following is a partial specification of the Drone concept. As shown, the Drone has properties, functionalities, and states (Boolean expressions validating states).

```
CONCEPT Drone {
  PARENTS {srvllnceDrone.drones.CONCEPT_TREES.System}
  CHILDREN { }
  PROPS {
    PROP dFlyCapacity {TYPE{srvllnceDrone.drones.CONCEPT_TREES.FlyingCapacity} CARDINALITY{1}}
    PROP dPlanner {TYPE{srvllnceDrone.drones.CONCEPT_TREES.Planner} CARDINALITY{1}}
    PROP dCommunicationSys {TYPE{srvllnceDrone.drones.CONCEPT_TREES.CommunicationSys} CARDINALITY{1}}
  }
  FUNCS {
    FUNC plan {TYPE {srvllnceDrone.drones.CONCEPT_TREES.Plan}}
    FUNC lineExplore {TYPE {srvllnceDrone.drones.CONCEPT_TREES.LineExplore}}
    FUNC spiralExplore {TYPE {srvllnceDrone.drones.CONCEPT_TREES.SpiralExplore}}
    FUNC takeoff {TYPE {srvllnceDrone.drones.CONCEPT_TREES.TakeOff}}
    FUNC flyTowardsBase {TYPE {srvllnceDrone.drones.CONCEPT_TREES.FlyTowardsBase}}
    FUNC lookForPeople {TYPE {srvllnceDrone.drones.CONCEPT_TREES.LookForPeople}}
  }
  STATES {
    STATE IsUp { PERFORMED{srvllnceDrone.drones.CONCEPT_TREES.Drone.FUNCS.takeoff} }
    STATE IsPlaning { IS_PERFORMING{srvllnceDrone.drones.CONCEPT_TREES.Drone.FUNCS.plan} }
    STATE IsExploring { IS_PERFORMING{srvllnceDrone.drones.CONCEPT_TREES.Drone.FUNCS.lineExplore} OR
          IS_PERFORMING{srvllnceDrone.drones.CONCEPT_TREES.Drone.FUNCS.spiralExplore} }
    STATE InLowFlayCapacity {
          srvllnceDrone.drones.CONCEPT_TREES.Drone.PROPS.dFlyCapacity.STATES.smallFlyingTime}
    STATE FoundAreaOfInterest { srvllnceDrone.drones.CONCEPT_TREES.Drone.STATES.IsExploring AND
          srvllnceDrone.drones.CONCEPT_TREES.SpottedAreasOfInterest >= 1 }
    STATE FlayingOverAreaOfInterest { }
    STATE IsExploringAreaOfInterest { srvllnceDrone.drones.CONCEPT_TREES.Drone.STATES.IsExploring AND
          srvllnceDrone.drones.CONCEPT_TREES.Drone.STATES.FlayingOverAreaOfInterest }
    STATE FoundPeopleOfInterest {
          srvllnceDrone.drones.CONCEPT_TREES.Drone.STATES.IsExploringAreaOfInterest AND
          srvllnceDrone.drones.CONCEPT_TREES.SpottedPeople >= 1 }
} }
```

To specify the drone's behaviour with KnowLang, we used goals, policies, and situations (see Section 3). The following is a specification sample showing a drone's policy called $GoFindAreaOfInterest$. As shown, the policy is specified to handle the goal $FindAreaOfInterest$ and is triggered by the situation $DroneIsOnAndAreaNot\,Found$. Further, the policy triggers via its $MAPPING$ sections conditionally (e.g., there is a $CONDITONS$ directive that requires the drone's flying capacity be higher than the estimated time needed to get back to the base) the execution of a sequence of actions. When the conditions were the same, we specified a probability distribution among the $MAPPING$ sections involving same conditions (e.g., $PROBABILITY\,0.6$), which represents our initial belief in action choice.

```
CONCEPT_POLICY GoFindAreaOfInterest {
  CHILDREN {}
    PARENTS { srvllnceDrone.drones.CONCEPT_TREES.Policy}
    SPEC {
      POLICY_GOAL { srvllnceDrone.drones.CONCEPT_TREES.FindAreaOfInterest }
      POLICY_SITUATIONS { srvllnceDrone.drones.CONCEPT_TREES.DroneIsOnAndAreaNotFound }
      ....
      POLICY_MAPPINGS {
        MAPPING {
         CONDITIONS { srvllnceDrone.drones.CONCEPT_TREES.TimeToDroneBase <
           srvllnceDrone.drones.CONCEPT_TREES.drone.PROPS.dFlyCapacity }
         DO_ACTIONS { srvllnceDrone.drones.CONCEPT_TREES.drone.FUNCS.lineExplore }
         PROBABILITY {0.6}
      }
      MAPPING {
        CONDITIONS { srvllnceDrone.drones.CONCEPT_TREES.TimeToDroneBase <
          srvllnceDrone.drones.CONCEPT_TREES.drone.PROPS.dFlyCapacity }
        DO_ACTIONS { srvllnceDrone.drones.CONCEPT_TREES.drone.FUNCS.spiralExplore }
        PROBABILITY {0.4}
      }
      ....
} } }
```

As specified, the probability distribution gives the designer's initial preference about what actions should be executed if the system ends up running the $GoFindAreaOf\ Interest$ policy. Note that at runtime, the KnowLang Reasoner maintains a record of all the action executions and re-computes the probability rates every time when a policy has been applied, and subsequently when

actions have been executed. Thus, although initially the system will execute the function *lineExplore* (it has the higher probability rate of 0.6), if that policy cannot achieve its goal with this action, then the probability distribution will be shifted in favour of the function *spiralExplore*, which might be executed next time when the system will try to apply the same policy. Therefore, probabilities are recomputed after every action execution, and thus, the behaviour changes accordingly.

As mentioned above, policies are triggered by situations. Therefore, while specifying policies handling the drone's objectives (e.g., $FindAreaOfInterest$), we need to think of important situations that may trigger those policies. Note that these situations will eventually be outlined by scenarios providing alternative behaviours or execution paths out of that situation. The following code represents the specification of the situation $DroneIsOnAndAreaNotFound$, used for the specification of the $GoFindAreaOfInterest$ policy.

```
CONCEPT_SITUATION DroneIsOnAndAreaNotFound {
....
  SPEC {
    SITUATION_STATES {srvllnceDrone.drones.CONCEPT_TREES.drone.STATES.IsUp,
      srvllnceDrone.drones.CONCEPT_TREES.drone.STATES.IsExploring}
    SITUATION_ACTIONS {srvllnceDrone.drones.CONCEPT_TREES.LineExplore,
      srvllnceDrone.drones.CONCEPT_TREES.FlyTowardsBase}
}}
```

As shown, the situation is specified with $SITATION_STATES$ (e.g., the drone's states $IsUp$ and $IsExploring$) and $SITUATION_ACTIONS$ (e.g., $LineExplore$, $SpiralExplore$, and $FlyTowardsBase$). To consider a situation effective (i.e., the system is currently in that situation), the situation states must be respectively effective (evaluated as true). For example, the $DroneIsOnAndAreaNotFound$ situation is effective if the Drone's state $IsExploring$ is effective (is on hold). The possible actions define what actions can be undertaken once the system fails in a particular situation.

Note that the specification presented is a part of the KB that is operated by the KnowLang Reasoner. The reasoner encapsulates that KB and acts as a module in the awareness layer of the *framework for knowledge collection* (see Figure 1). The reasoner is "fed" with classified sensory data (labels), produced by the classification layer, and returns situational information and proposed behaviour upon request. The consumed labels help the reasoner update the KB, which results in re-evaluation of the specified concept states (recall that states are specified as a Boolean expression over the ontology, i.e., a state expression may include any element in the KB). Subsequently, the evaluation of the specified states helps the reasoner determine at runtime whether the system is in a particular situation or if a particular goal has been achieved. Moreover, it can deduce an appropriate policy that may help the drone "go out" of a particular situation.

5 Conclusion and Future Work

In this paper we proposed a combination of two innovative frameworks for context-awareness and reasoning aimed at self-aware systems. The former has been designed

around the concept of reconfiguration and built using well-established Java technologies. It is able to collect data from a number of different sources and classify them using general-purpose algorithms. The latter has been designed for reasoning and action selection using both logical and statistical techniques. To test their mutual synergies, a case study describing a self-aware surveillance drone has been described in greater detail.

We are planning to challenge this approach in more complex scenarios to better understand how the framework self-* structure could simplify the engineering of pervasive applications, particularly on mobile platforms.

Acknowledgments. This work was supported by the European Union FP7 Integrated Project Autonomic Service-Component Ensembles (ASCENS) and by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre.

References

1. Smart cities Ranking of European medium-sized cities, Vienna, Austria (2007). http://tinyurl.com/bqh83np
2. Abeywickrama, D.B., Bicocchi, N., Zambonelli, F.: Sota: towards a general model for self-adaptive systems. In: Reddy, S., Drira, K. (eds.) WETICE, pp. 48–53. IEEE Computer Society (2012)
3. Bicocchi, N., Cecaj, A., Fontana, D., Mamei, M., Sassi, A., Zambonelli, F.: Collective awareness for human-ict collaboration in smart cities. In: WETICE, pp. 3–8 (2013)
4. Bicocchi, N., Fontana, D., Zambonelli, F.: A self-aware, reconfigurable architecture for context awareness. In: IEEE Symposium on Computers and Communications, Madeira, Portugal (2014)
5. Bicocchi, N., Lasagni, M., Zambonelli, F.: Bridging vision and commonsense for multimodal situation recognition in pervasive systems. In: International Conference on Pervasive Computing and Communications, Lugano, Switzerland (2012)
6. Kehoe, M., et al.: Understanding ibm smart cities. Redbook Series (2011)
7. Khan, W.Z., Xiang, Y., Aalsalem, M.Y., Arshad, Q.: Mobile phone sensing systems: A survey. IEEE Communication Survey and Tutorials **15**, 402–427 (2013)
8. Neapolitan, R.: Learning Bayesian Networks. Prentice Hall (2003)
9. Vassev, E.: KnowLang Grammar in BNF. Tech. Rep. Lero-TR-2012-04, Lero, University of Limerick, Ireland (2012)
10. Vassev, E., Hinchey, M.: Knowledge representation for cognitive robotic systems. In: Proceedings of the 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISCORCW 2012), pp. 156–163. IEEE Computer Society (2012)
11. Vassev, E., Hinchey, M., Gaudin, B.: Knowledge representation for self-adaptive behavior. In: Proceedings of C* Conference on Computer Science and Software Engineering (C3S2E 2012), pp. 113–117. ACM (2012)
12. Ye, J., Dobson, S., McKeever, S.: Situation identification techniques in pervasive computing: A review. Pervasive and Mobile Computing **8**, 33–66 (2012)