EAI Endorsed Transactions
on Cognitive Communications
Research Article ICST.ORG

High-level Programming and Symbolic Reasoning on IoT Resource Constrained Devices

Salvatore Gaglio [1,2], Giuseppe Lo Re [2], Gloria Martorella [2], Daniele Peri [2,*]

[1] ICAR CNR, Viale delle Scienze, Ed. 11, 90128 Palermo, Italy
[2] DICGIM University of Palermo,Viale delle Scienze, Ed. 6, 90128 Palermo, Italy

Abstract

While the vision of Internet of Things (IoT) is rather inspiring, its practical implementation remains challenging. Conventional programming approaches prove unsuitable to provide IoT resource constrained devices with the distributed processing capabilities required to implement intelligent, autonomic, and self-organizing behaviors. In our previous work, we had already proposed an alternative programming methodology for such systems that is characterized by high-level programming and symbolic expressions evaluation, and developed a lightweight middleware to support it. Our approach allows for interactive programming of deployed nodes, and it is based on the simple but e ective paradigm of executable code exchange among nodes. In this paper, we show how our methodology can be used to provide IoT resource constrained devices with reasoning abilities by implementing a Fuzzy Logic symbolic extension on deployed nodes at runtime.

Keywords: High-level programming, Resource constrained devices, Knowledge Representation, Fuzzy Logic.

Received on 22 December 2014; accepted on 10 March 2015; published on 28 May 2015

doi:10.4108/cogcom.1.2.e6

1. Introduction

According to the Internet of Thing (IoT) vision [1], all kinds of devices, although computationally limited, might be used to interact with people or to manage information concerning the individuals themselves [2]. Besides reactive responses on input changes, the whole network may exhibit more advanced behaviors resulting from reasoning processes carried out on the individual nodes or emerging from local interactions. However, nodes' constraints leave the system designers many challenges to face, especially when distributed applications are considered [3]. Conventional programming methodologies often prove inappropriate on resource constrained IoT devices, especially when knowledge must be treated with a high level representation or changes of the application goals may be required after the network has been deployed [4]. Moreover, the implementation of intelligent mechanisms, as well as symbolic reasoning, through rigid layered architectures, reveals impracticable on resource constrained devices such as those commonly used in Wireless Sensor Networks (WSNs). Often this issue is faced through the adoption of an intelligent centralized system that uses WSNs as static sensory tools [5]. Indeed, integration of WSN devices in the IoT seems quite natural and desirable, provided that the aforementioned issues be addressed. In our previous work [6, 7], we introduced an alternative programming methodology, along with a lightweight middleware, based on high-level programming and executable code exchange among WSN nodes. The contribution of this paper consists in the extension of the methodology to include symbolic reasoning even on IoT resource constrained devices. The remainder of the paper is organized as follows. In Section 2 we describe the key concepts of our methodology and the symbolic model we adopted. In Section 3, we extend the symbolic approach characterizing our programming environment with Fuzzy Logic, and in Section 4 we show an application to make the nodes reason about their position with respect to thermal zones of the deployment area. Finally, Section 5 discuss the adopted

* Corresponding author. Email: daniele.peri@unipa.it

solution in terms of efficiency, while 6 reports our conclusions.

2. Key Concepts of the Development Methodology

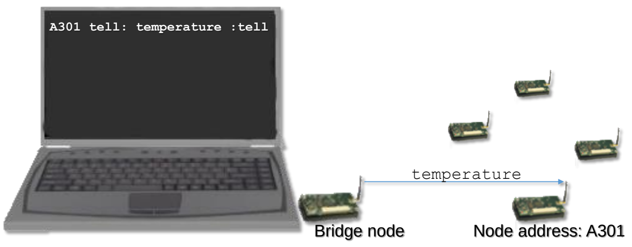
Mainstream praxis to program embedded devices consists in cross-compilation of specialized application code together with a general purpose operating system. The resulting object code is then uploaded to the on-board permanent storage. Instead, our methodology is based on high-level executable code exchange between nodes. This mechanism, while abstracted, is implemented at a very low level avoiding the burden of a complex and thick software layer between the hardware and the application code. Indeed, a Forth environment runs on the hardware providing the core functionalities of an operating system, including a command line interface (CLI). This also allows for interactive development, which is a peculiar feature of our methodology that can be used even to reprogram deployed nodes. This way, nodes can be made expand their capabilities by exchanging pieces of code among themselves in real time. The CLI is accessible through either a microcontroller's Universal Asynchronous Receiver-Transmitter (UART) or the on-board radio [6]. The Forth environment is inherently provided with an interpreter and a compiler. Both can be easily extended by definin new *words* stored in the *dictionary*. Being Forth a stack-based language, words use the stack for parameters passing. A command, or an entire program, is thus just a sequence words.

The acquisition of sensory data is already supported as we have previously extended the dictionary with the words to manage the sensor-MicroController Unit (MCU) interface, to enable the Analog to Digital Converter (ADC) and to leave the sensory reading on the stack. For instance, the program to measure the temperature is just the word `temperature`, whereas sensing the ambient light is achieved by executing the single word `luminosity`. Although the code is concise but expressive, the execution of these words involves the reading from the ADC and the return of the raw data on the stack. The description of a task in natural language and its implementation can be thus made very similar.

Our programming environment is composed of some nodes wirelessly deployed and a wired node that behaves as a bridge to send user inputs to the network. In previous work [7], we introduced the syntactic construct that implements executable code exchange among nodes:

$$\texttt{tell:}\ \langle code \rangle\ \texttt{:tell}$$

in which ⟨*code*⟩ is a sequence of words, sent as character strings, to be remotely interpreted by the receiver node. The address of the destination node is left on the top of the stack. A numeric as well as a string value, can be taken at runtime from the top of the stack and inserted in the outgoing packet when special markers, such as ~ for numbers and ~s for strings are encountered.



```
A301 tell: temperature :tell
```

Figure 1. Executable code exchange to make a node sense the temperature quantity. To tell a node to locally perform temperature measurements, the user can interact with the bridge node interpreter by typing on its shell the sequence of words to exchange symbolic programs. The destination node address –the hexadecimal value A301 in the example– is expressed as a 16-bit value according to the IEEE802.15.4 short addressing mode. The word `temperature` is the symbolic program that is copied in the outgoing frame payload, sent as character string to be remotely interpreted.

Therefore, the exchange of code promotes distributed computations since a node that is not equipped with temperature and light sensors can tell another to measure the temperature or ambient light just by executing the construct `tell:` ⟨*code*⟩ `:tell` and including the symbolic program for the measurement as follows:

```
tell: temperature :tell
```

or it can be typed at the CLI of the bridge node, as in Figure 1.

Due to their internal limitations, sensor nodes are mostly confine to perceive environmental conditions in WSN applications. This is not expected to change in the IoT context, yet in the following section we show how using a suitable programming approach even small sensors can be provided with symbolic reasoning abilities.

3. Distributed Processing and Symbolic Reasoning

In our programming environment, purely reactive behaviors can be easily implemented on the remote nodes by sending them the sequence of words to be executed if certain conditions are met. Let us consider the following command given through the CLI of the bridge node:

```
bcst tell: close-to-window? [if]
red led on [then] :tell
```

This command broadcasts –the word bcst leaves the reserved address for the purpose on the stack– the code between the `tell:` and `:tell` words. Once received, each node executes the word `close-to-window?` to evaluate if it is close to the window and, if so turns the red LED on. The word `close-to-window?`, already in the dictionary, performs temperature and luminosity measurements and checks if both sensory readings are above a predefine threshold. As it can be noticed, the code is quite understandable, although all the words operate just above the hardware level by setting ports or enabling the ADCs to read temperature and light exposure. This code, as well as those in the rest of the paper, has been used on Iris Mote nodes equipped with the MTS400 sensor board to acquire data about temperature and light exposure. For the sake of showing how it is possible to incorporate in our middleware new abstractions to support intelligent applications here we introduce a Fuzzy Logic extension. Fuzzy Logic has the peculiarity to be appropriate to implement approximate reasoning in several contexts as well as for machine learning purposes [8]. We adopted a classic Forth Fuzzy Logic implementation [9] that we modifie to make it run on the Harvard architecture AVR microcontroller used in the Iris Mote platform. Finally, we also enriched the original implementation with the possibility to exchange fuzzy definition and evaluation among nodes.

The wordset to enable high-level fuzzy reasoning on IoT resource constrained devices is provided in Table 1 and allows for the creation of fuzzy input/output variables, for the definitio of the related membership functions, for fuzzific tion, for rule evaluation and for defuzzific tion processes.

Differently from [9], to create a new fuzzy variable we included the word `fvar` to be used according to the following syntax:

<min_val> *<max_val>* `fvar` *<name>*

where *<min_val>* and *<max_val>* represent the defini tion domain of the fuzzy variable and *<name>* is the name associated with the new variable. Differently from *<min_val>* and *<max_val>* values that are expected to be on the stack, the variable name is provided at runtime. When this construct is executed by the node interpreter, a new entry named *<name>* is created in the dictionary, which is located in Flash memory, while a fi e cells structure is allocated in RAM. As illustrated in Figure 2, a fuzzy variable can be thought of as a sequence of fie ds. The Forth code to create this structure is self-explanatory:

```
begin-structure fv
  field: fv.crisp
  field: fv.link
  field: fv.low
  field: fv.high
  field: xt
end-structure
```

Table 1. Words defined in the dictionary to implement fuzzy reasoning according to [9].

Word	Description
`slope`	Compute the slope given two points of a side
`set-slope`	Set the left and right slope in the appropriate membership fields
`&`	Fuzzy AND
`|`	Fuzzy OR
`~`	Fuzzy NOT
`=>`	Fuzzy implication
`fuzzify`	Given a crisp value and a membership, assign a membership value for it
`apply`	Apply the crisp input to the specified fuzzy input variable
`output`	Create an output fuzzy variable
`singleton`	Define a singleton output function
`rules`	Evaluate rules
`conclude`	Defuzzify and leave the crisp output on top of the stack

Once the word `fvar` is executed, a generic `fv` structure is instantiated and *<min_val>* and *<max_val>* values are stored in the `fv.low` and `fv.high` fie ds. The firs fie d stores the crisp input value, and it is followed by a link fie d, i.e. the membership function list associated with that fuzzy variable. The following two fie ds contain the validity range, i.e. the minimum value and the maximum value allowed for the crisp input. Finally, as we want to allow the nodes to reason about sensory data, the last fie d contains the address of the word to perform the measurement of the physical quantity associated with that variable.

Let us defin two fuzzy variables, `temp` and `lightexp`. The last fie d of `temp` stores the address of the word `temperature`, while the address of the word `luminosity` is the last fie d of `lightexp`. The words `luminosity` and `temperature` have been already introduced in the previous section.

Similarly, to create a membership function, the word `member` expects on the stack four control points which determine the shape of the membership function and its name is provided at runtime according to the following syntax:

<bottom-left> *<top-left>* *<top-right>* *<bottom-right>* `member` *<name>*

As a fuzzy variable, also a membership function is a generic structure composed of several fie ds:

```
begin-structure membership
  field: fval
```

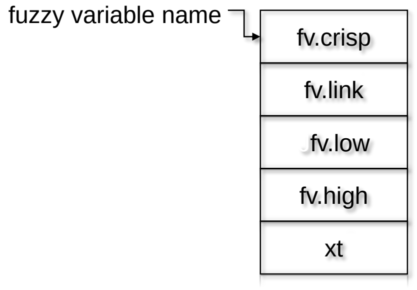
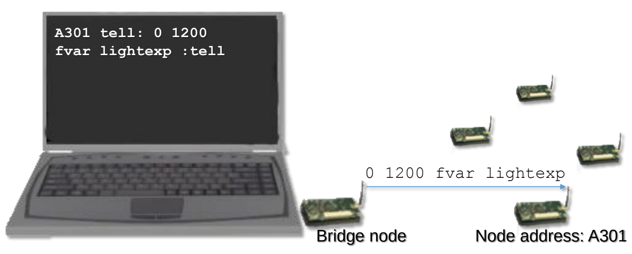


Figure 2. The definition of a fuzzy variable include a new entry in the Flash memory word dictionary and allocates five contiguous cells in RAM as a sequence of fields. The first cell stores the crisp input value, while the link field contains the address of the first defined membership function related to the fuzzy variable. The following two cells store the validity range, while the last one stores the execution token (*xt*), i.e. the address of the word t sense the physical quantity associated with the fuzzy variabl Once the fuzzy variable name is used, the address of the fir field is fetched on top of stack.

```
                  field: link
                  field: lm
                  field: lt
                  field: rt
                  field: rm
                  field: ls
                  field: rs
          end-structure
```

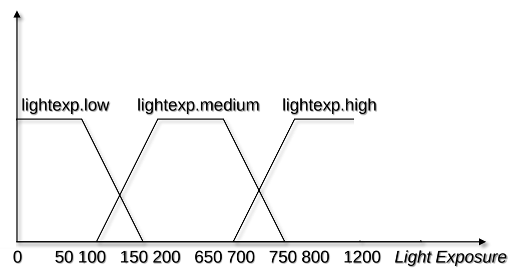
The firs fie d contains the truth value resultin after the fuzzific tion process, while the second fie stores the address of the next membership function Essentially, a fuzzy variable and its membership functions are implemented as linked list. Membership functions are trapezoidal and therefore four contro points are stored in the appropriate four successive fie ds, left-most (*lm*), left-top (*lt*), right-top (*rt*), and right-most (*rm*). Finally, two further memory cells are required to store the left slope (*ls*) and the right slope (*rs*) of both sides. When the firs membership function is defined the fuzzy variable link fie d stores the address of the newly created membership function. As the word `member` is executed, the four control points on top of stack are stored in the appropriate fie ds of the `membership` structure along with the left and right slopes. Figure 4 shows the code to defin the fuzzy variable related to light exposure named `lightexp` and the related membership functions according to the words described previously.

Moving on with the initial example in which a node evaluates its proximity to a window, in place of two crisp variables, the fuzzy variables `temp` and `lightexp`



```
A301 tell: 0 1200
fvar lightexp :tell
```

Figure 3. Similarly to Figure 1, the executable code exchange mechanism allows to define fuzzy variables and their related membership functions on already deployed nodes. To remotely define the fuzzy variable `lightexp`, the code to be remotely executed must be enclosed between `tell:` and `:tell` and typed at the CLI of the bridge node that sends the executable code to the destination node. The remote node receives the sequence of



```
0       1200    fvar    lightexp
0       0        50     150      member   lightexp.low
100     200     650     750      member   lightexp.medium
700     800     1200    1200     member   lightexp.high
```

Figure 4. Fuzzy sets associated with the fuzzy variable `lightexp`. On the right side, the code to define the fuzzy variable `lightexp` and its membership functions. The definition domain, corresponding to the raw readings values interval [0,1200], is given before the word `fvar`, while the word `member` defines each of the three trapezoidal membership functions by using four control points (bottom-left, top-left, top-right, and bottom-right).

can be easily define on deployed nodes provided that the symbolic program is placed between `tell:` and `:tell` as indicated in Figure 3.

The representation of a fuzzy variable and its membership functions in memory is provided in Figure 5.

A node can be made measure light exposure, and fuzzify it with the code:

```
lightexp measure apply
```

The word `measure` fetches the xt fie d of the fuzzy variable that precedes it and executes the associated

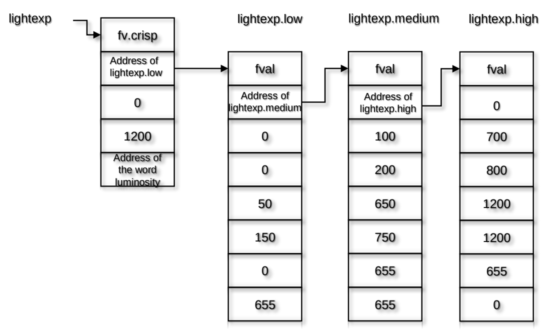


Figure 5. Memory representation of the fuzzy variable `lightexp` and its related membership functions after the code shown in Figure 4 is executed. The implementation refers to linked structures. Each link field stores the address of the next defined membership function. A link field that is equal to zero indicates the last membership function concerning that variable. It is worth noticing that the slope values are "scaled" to 65535 since this is the maximum number that can be expressed with 16-bits.

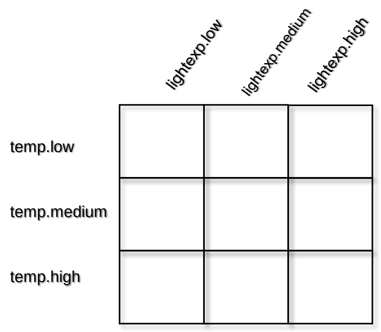


Figure 6. The execution of `temp lightexp 2 classification thermal-zone` creates the word `thermal-zone` that is bound to the two fuzzy variables. A 9 cells sized memory area is allocated as `temp` and `lightexp` have both three linguistic variables associated. In essence, each of these cells identifies a thermal-zone, a membership class according to which the node classifies itself. This area stores all the possible combinations for the rule evaluation process and aggregation.

code. In detail, when the word `measure` is interpreted, the word address, which is stored in the xt fie d, is executed. Then, the word `luminosity` is executed and the sensory reading is left on top of the stack. This value is treated as crisp input by the word `apply`. As its name suggests, the word `apply` applies the crisp input to all the membership functions referring to `lightexp` and stores the fuzzy truth value in the corresponden t `fval` fie d. Basicall y this word scans the linked list and fuzzifie the sensory reading for each membership function.

To access the truth value resulting from the fuzzific tion process the code:

```
lightexp.low @
```

pushes onto the stack the truth value by using the built-in word @ (*fetch*). Rather than through a thresholding process, a device can establish if it is close to the window through the evaluation of fuzzy rules in the form:

```
temp.high @ lightexp.high @ & => close-to-window
```

where `temp.high` and `lightexp.high` are membership functions of the fuzzy input variable `temp` and `lightexp` respectively, and `close-to-window` is one of the linguistic labels associated to the output variable. Similarly to the case of the thresholding process, if both the temperature and the light exposure levels are high a node can infer to be under sunlight , and thus close to the window .

4. Inferring the Node Distribution according to Thermal Zones

Let us suppose we intend to make the deployed nodes able to discover their distribution with respect to thermal zones of an environment lighted by some windows exposed to direct sunlight , and lamps. Each node assesses in turn the thermal zone it belongs to, and makes the others aware of this information. We define the syntactic construct `classification` to make the nodes able to classify according to an arbitrary number of fuzzy variables. With the previousl y define input variables `temp` and `lightexp` the code:

```
temp lightexp 2 classification thermal-zone
```

creates the new word `thermal-zone`, which is bound to the two fuzzy variables `temp` and `lightexp` as illustrated in Figure 6.

When the new word `thermal-zone` is executed, it measures the temperature and luminosity , fuzzifie the crisp inputs and evaluates the rules by storing the firin strength for each rule, indicating the degree to which the rule matches the inputs. The rule generation process considers all the possible combinations of all the membership functions, -i.e. in this case, the set of all ordered pairs `(a,b)` where `a` and `b` are linguistic terms associated respectively with `temp` and `lightexp`. When handling few variables, this does not cause excessive memory occupation. It offers instead the advantage of considering a fine-g ained classific tion based on all the n-tuples, that in this case, are all valid. However, optimization methods for the reduction

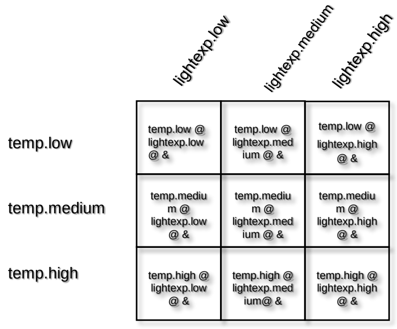

lightexp.low
lightexp.medium
lightexp.high
temp.low
temp.medium
temp.high
temp.low @ lightexp.low @ &
temp.low @ lightexp.medium @ &
temp.low @ lightexp.high @ &
temp.medium @ lightexp.low @ &
temp.medium @ lightexp.medium @ &
temp.medium @ lightexp.high @ &
temp.high @ lightexp.low @ &
temp.high @ lightexp.medium@ &
temp.high @ lightexp.high @ &

Figure 7. The rule generation involves the evaluation of all the possible combinations of the truth values of each membership function. Finally, the rule aggregation process consists in scanning the table to return the cell index storing the rule with the maximum strength. This index represents the class the node belongs to.

of a larg e scale rule base may be required in real-time fuzzy systems [10–12]. When needed, the table is traversed to compute the membership grade of the output by aggregating all rules. The rule with the maxim um strength is taken as the output membership class (Figure 7). This way, each node is able to classify itself into one of the thermal zones. To support more sophistica ted beha viors, it is possible to expl oit the mechanism of code exchang e among nodes to trig ger the process of neighbor discov ery in order to keep track of their classific tion into thermal-zones.

For this purpose, it is necessary to defin the table `nodes-distribution` to contain the number of nodes for each thermal zone (Figure řeffig:thermal classific tion). To trig ger the whole classific tion process, the word `classification-start` can be sent to already depl oyed nodes through the executable code exchang e par adigm. For instance, each device starts the timer and can transmit once, after waiting (word `on-timer`) for a time that is function of its unique ID. When its time is elapsed, the word `classification-spread` is executed, the node classifie itself into a thermal zone and then broadcasts the class it bel ongs to, together with the code to make the others upda te the whole distribution. The Forth code required for the entire process is the foll owing:

```
: local-update
nodes-distribution update ;

: spread
dup local-update
bcst [tell:] ~ local-update [:tell] ;

: classification-spread
thermal-zone spread ;
```

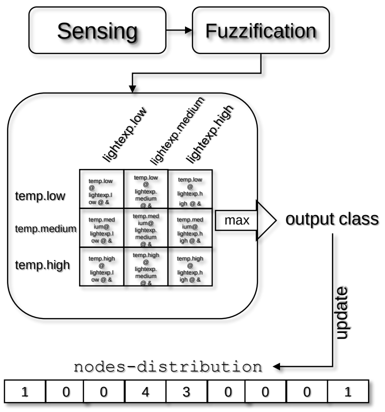


Figure 8. The word "thermal-zones" operates by sensing the temperature and light. Both sensory readings are treated as crisp input and fuzzified according to the membership functions of `lightexp` and `temp`. The rule generation considers all the pairs of truth values related to linguistic variables bound to different fuzzy variables. This is justified by the fact that each combination represents a different thermal zone identified by distinct temperature and light conditions. Indeed, the index of the cell storing the maximum value represents the thermal zone the node belongs to. The cell correspondent to the output class is incremented in nodes-distribution in order to allow each node to assess the distribution of the others.

```
: classification-start
start-timer
on-timer ['] classification-spread ;
```

in which the word `spread` crea tes a messag e with the code to make the other devices upda te locall y the `nodes-distribution`. At the end of the `update` process, each node hol ds the curren t nodes distribution in terms of thermal zones, as such:

```
Class 1 2 3 4 5 6 7 8 9
    # 5 1 0 0 0 0 0 1 1
```

Five nodes bel ongs to class 1, one node to class 2 and so on. Each node knows the number of nodes in the netw ork and their position, without any cen tralized computa tion. Once some nodes are mov ed from their position to another , and the process is trig gered ag ain, each node is able to detect the new distribution.

Moreov er, the anal ysis of the nodes distribution may lead a node to classify itself as an outlier , to trig ger self-diagnosis oper ations, and even to take specifi actions, by reasoning about the whole netw ork configu ation and its membership thermal zone. The inter activity gran ted by our approach permits the progr ammer to comm unica te with the netw ork through the serial shell of the bridg e node. For instance, the progr ammer can tell the nodes bel onging to class 8 to turn their red LED on:

```
bcst tell: thermal-zone 8 class? [if]
   red led on [then] :tell ;
```

5. Experimental Results

Because of the limitations in terms of available resources, the implementation of symbolic reasoning on resource constrained devices must be particularly efficient. Our approach makes IoT applications to be developed on real devices provided with an environment running at close contact with the hardware. This prevents the presence of further intermediate layers between the hardware and software applications, increasing efficiency. Moreover, as already widely discussed, although running on the hardware, the symbolic computation allows to treat knowledge with a high degree of expressiveness. Differently from mainstream approaches, distributed computation is made inexpensive due to the fact that both high and low level executable code can be exchanged. The inclusion of reasoning mechanisms on resource constrained devices is particularly efficient as it occupies only 6 bytes of RAM and 863 bytes of Flash memory. The fuzzy wordset consists of 31 words. The application allowing the classific tion into thermal zones is quite compact since it consists of only 20 words and occupies 560 bytes of RAM and 825 bytes of Flash memory.

6. Conclusions

In this paper, we showed how distributed symbolic reasoning can be implemented on resource constrained IoT devices by exploiting executable code exchange. Our contribution aims to fil the lack in the absence of programming paradigms enabling a vast adoption of IoT in everyday life. The possibility to exchange executable code makes the system adaptive and autonomous, since each node can evolve on the basis of realtime inputs, in terms of both data and executable code, from other nodes and from the user. We showed how abstractions and symbolic expression evaluation can be efficiently incorporated into a programming model for such networks by exploiting both interpretation and compilation of code. As an example, we described the syntactic constructs that can be define to make the nodes aware of their position with respect to a subdivision of the environment into thermal zones. Our methodology reveals suitable for implementing more advanced behaviors on IoT devices since symbolic reasoning is performed even on inexpensive, and resource constrained microcontrollers.

References

[1] Atzori, L., Iera, A. and Morabito, G. (2010) The Internet of Things: A Survey. *Computer Networks* **54**(15): 2787 – 2805. doi: http://dx.doi.org/10.1016/j.comnet.2010.05.010 .

[2] Guo, B., Zhang, D., Yu, Z., Liang, Y., Wang, Z. and Zhou, X. (2013) From the Internet of Things to Embedded Intelligence. *World Wide Web* **16**(4): 399–420. doi: 10.1007/s11280-012-0188- y.

[3] Martorella, G., Peri, D. and Toscano, E. (2014) Hardware and Software Platforms for Distributed Computing on Resource Constrained Devices. In Gaglio, S. and Lo Re, G. [eds.] *Advances onto the Internet of Things* (Springer International Publishing), *Advances in Intelligent Systems and Computing* **260**, 121–133. doi: 10.1007/978-3-319-03992-3_9 .

[4] Kortuem, G., Kawsar, F., Fitton, D. and Sundramoorthy, V. (2010) Smart Objects as Building Blocks for the Internet of Things. *Internet Computing, IEEE* **14**(1): 44–51. doi: 10.1109/MIC.2009.143 .

[5] De Paola, A., Ortolani, M., Lo Re, G., Anastasi, G. and Das, S.K. (2014) Intelligent Management Systems for Energy Efficiency in Buildings: A Survey. *ACM Comput. Surv.* **47**(1): 13:1–13:38.

[6] Gaglio, S., Lo Re, G., Martorella, G. and Peri, D. (2014) A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*: 1–8. doi: 10.1109/ETF A.2014.7005179 .

[7] Gaglio, S., Re, G.L., Martorella, G. and Peri, D. (2014) A Lightweight Middleware Platform for Distributed Computing on Wireless Sensor Networks. *Procedia Computer Science* **32**(0): 908 – 913. doi: http://dx.doi.org/10.1016/j.procs.2014.05.510 , URL `http://www.sciencedirect.com/science/article/pii/S1877050914007108`. The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014).

[8] Navara, M. and Peri, D. (2004) Automatic Generation of Fuzzy Rules and its Applications in Medical Diagnosis. In *Proc. 10th Int. Conf. Information Processing and Management of Uncertainty, Perugia, Italy*, **1**: 657–663.

[9] VanNorman, R. (1997) Fuzzy Forth. *Forth Dimensions* **18**: 6–13.

[10] De Paola, A., Lo Re, G. and Pellegrino, A. (2014) A Fuzzy Adaptive Controller for an Ambient Intelligence Scenario. In Gaglio, S. and Lo Re, G. [eds.] *Advances onto the Internet of Things* (Springer International Publishing), *Advances in Intelligent Systems and Computing* **260**, 47–59.

[11] Jin, Y. (2000) Fuzzy Modeling of High-dimensional Systems: Complexity Reduction and Interpretability Improvement. *Fuzzy Systems, IEEE Transactions on* **8**(2): 212–221. doi: 10.1109/91.842154 .

[12] Yam, Y., Baranyi, P. and Yang, C.T. (1999) Reduction of Fuzzy Rule Base via Singular Value Decomposition. *Fuzzy Systems, IEEE Transactions on* **7**(2): 120–132. doi: 10.1109/91.755394 .