A Constrained ECA Language Supporting Formal Verification of WSNs

Flavio Corradini,Rosario Culmone, Leonardo Mostarda,Luca Tesei
Computer Science Division
University of Camerino, Italy
Email: {flavio.corradini,rosario.culmone,leonardo.mostarda,luca.tesi}@unicam.it

Franco Raimondi
Computer Science Department
Middlesex University, UK
Email: {f.raimondi}@mdx.ac.uk

***Abstract*—Modern wireless sensor and actuator networks (WSANs) are composed of spatially distributed low cost nodes that can contain different sensors and actuators. Event condition action (ECA) based languages have been widely proposed in order to program WSANs. Implementing applications by using ECA rules is an error-prone process thus various formal methods have been proposed. In spite of this great variety, formal verification of ECA rules has not been tailored to the context of WSANs. In this paper we present IRON, an ECA language for programming WSANs. IRON allows the automatic verifications of ECA rules. These are used by the IRON run-time platform in order to implement the required behaviour.**

***Keywords*-Event condition actions, constraints programming, wireless sensor and actuator networks, model checking.**

I. Introduction

Recent advances in sensor technology, micro-processors, wireless communications and Micro Electro-Mechanical Systems (MEMS) have fostered a strong research interest in wireless sensor and actuator networks (WSANs). Modern WSANs are composed of spatially distributed low cost nodes that can contain different sensors and actuators with good computation and processing capabilities. WSANs have been used to build various applications (e.g., home automation and agriculture automation) and quite often event condition action (ECA) based languages [1], [2], [3] are used in order to program them. ECA rules are used to define responses to events and are specified in the form "on the occurrence of certain events, if some conditions are true, do these actions". Implementing reactive systems by means of ECA rules is an error-prone process [4] thus various formal methods have been suggested. For instance approaches based on Petri Net (PN) [4], SMV [5], SPIN [6] have been proposed. However, in spite of this range of approaches, the formal verification of ECA has not been tailored to the context of WSANs. WSANs requires macroprogramming [7], the ability of grouping sensors together in order to perform multicast and broadcast communications.

In this paper we introduce IRON (Integrated Rule ON data). IRON is a language that supports the organisation of sensors and actuators into sets [**?**], allows the definition of conditions and actions over sets thus abstracting multicast and broadcast communications. IRON programs are composed of two separate specifications that are static and dynamic. The static part is composed of variables declarations (these are sets, physical and logical devices) plus global constraints defined over them (i.e., first order formulae). Constraints can be used in order to specify rules that bind variables together. The dynamic part is composed of ECA rules that are defined by the programer. The separation of static and dynamic parties brings various advantages. The static model can be reused for different applications. The same rules, if they are defined over sets, can be reused when new devices are added and/or removed from these sets (effectively sets decouple the device instances and the rule definitions).

IRON supports the formal verification of event-condition-action rules. More precisely, it supports the verification of the following properties: (i) *termination*; (ii) *confluence*; (iii) *consistency*. These are verified by translating IRON programs into Alloy [8] specifications. This is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples.

Section II of the paper presents our case study. Section III details the IRON language and Section IV discusses all properties that can be verified. Section V discusses the implementation of IRON. Finally, section VI compares the presented work with existing related works in this area while section VIII provides a conclusion and outlines future work.

II. A home automation case study

Monitoring and automatic control of building environment is a case study considered quite often in the literature [9], [10]. Home automation can include the following functionalities: (i) heating, ventilation, and air conditioning (HVAC) systems; (ii) emergency control systems (fire alarms); (iii) centralised light control; and (iv) other systems to provide comfort, energy efficiency and security. In order to validate our approach we consider the fire alarm system and the automatic heating application.

The fire alarm system is composed of temperature sensors, smoke detectors and sprinkler actuators. When a temperature sensor reads a value that exceeds a specified threshold (e.g., 50 celsius degree) and a smoke sensor detects smoke all the sprinklers are activated.

```
1  program ≡ ( device | rule | var_decl )+
2
3  device  ≡ physicalDevice | logicalDevice | set
4
5  physicalDevice  ≡
6    physical (sensor|actuator) type id [= exp] node(id,id)
7    [in id (, id )* ] [where exp]
8
9  logicalDevice ≡
10   logical (sensor | actuator) type id = exp
11   [in id (, id )* ] [where exp]
12
13 set ≡ set (sensor | actuator) type id
14
15 rule ≡ rule id on (id)+ when exp then action
16
17 action  ≡ [id = exp ]+
18
19 exp ≡ exp op exp | (exp) | term
20
21 term ≡ id | int | set_op set id | true | false | function
22
23 type ≡ int | boolean
24
25 set_op ≡ all | any | one | no | lone
26
27 OP ≡ == | != | < | > | <= | >= |  + | - | * | / |
28   and | or | not
29
30 var_decl ≡ type id = exp [where exp]
```

Figure 1. The IRON extended BNF

The automatic heating application is composed of different temperature sensors and various heaters and works in the following way: (i) if there exist a temperature value that is greater than the maximum temperature T_{max} (e.g., 24 celsius degree), the central heating system turns off; (ii) if there exist a temperature value that is less than the minimum temperature T_{min} (e.g., 16 celsius degree), the central heating system turns on.

III. IRON: Integrated Rule ON data

Figure 1 describes the IRON grammar. In order to ease the presentation we use an extended BNF form that uses regular-expression-like operations. More precisely we use $(x)*$ in order to mean zero or more repetitions of x, $(x)+$ in order to mean one or more repetitions of x while we use $[x]$ in order to mean an optional occurrence of x. Boldface is used in order to denote terminals of the grammar.

An IRON program defines various devices, sets and variable declarations (line 1 of the grammar).

A. Devices

A device can be physical, logical or a set (line 3 of the grammar). A physical device defines a piece of hardware that is physically installed in the environment, it has a type (i.e., integer [1] or boolean) and can be either a sensor or an actuator. A physical device has a name and always specifies the keyword $node(id, id)$ where the first id is an identifier

[1]For the sake of presentation we only consider integers however IRON also supports floating points.

```
1  physical sensor int temperature node(1,1)
2    in TemperatureSet
3    where temperature > −80 and temperature < 60
4  physical sensor int smoke node(2,1)
5    in SmokeSet
6
7  physical actuator boolean sprinkler node(3,1) in
       SprinklerSet
8  physical actuator boolean heating = false node(4,1) in
       HeatSet
9
10 logical sensor boolean tempAlarm = false
11 logical sensor boolean smokeAlarm = false
12
13 set sensor int TemperatureSet
14 set sensor int SmokeSet
15 set actuator int SprinklerSet
```

Figure 2. Static model of the home automation case study

that uniquely identifies the physical node while the second id uniquely identifies a sensor/actuator that is installed on the node. The keywords in and $where$ can be added in order to specify a list of sets the physical device belongs to and a static constraint the device must satisfy, respectively. A constraint specifies a law that various variables, devices and sets must satisfy. The use of a constraint is twofold: it defines valid states of the system regardless of the rules that are defined. On the other hand it is used at run-time to verify whether or not any physical device is providing erroneous data.

Figure 2 shows some declarations that are related to our home automation case study. The first line declares a temperature physical sensor that is of the type integer. The temperature sensor has identifier 1 and is installed on a node with identifier 1 (this is specified by using the keyword node(1,1)). This sensor is added to the TemperatureSet set. A constraint is defined on the temperature sensor that is its value cannot exceed sixty degree and be under minus eighty degree celsius. The third line declares a smoke physical sensor that is of the type boolean.

IRON also supports the definition of logical devices. A logical device can be set according to the values observed over different sensors and actuators thus it produces information that would be impossible to get by considering a single physical device. A logical sensor/actuator does not specify any $node(id, id)$ keyword but must specify an initial value (line $10 - 13$ of the grammar). We emphasise that logical devices (when possible) should be always preferred to normal variable definitions since the produced code is more readable.

The line 10 and 11 of Figure 2 define a $tempAlarm$ and $smokeAlarm$ boolean logical sensors. As we are going to see in the next Section this logical devices relates smoke and temperature data in order to support the detection of fire.

B. Sets

Sets (line 3 of the grammar) are considered to be logical devices and are used to group together either sensors or

actuators of the same type (line 15 of the grammar). A programmer can assign values to a set that contains actuators. This assignment can be used in order to instruct all the actuators to perform a specified action. Effectively, a set assignment is an abstraction of a multicast communication primitive that can be used to communicate to actuators an action to be performed. A programmer can read the value of a set of sensors in order to define events and specify conditions. To this end various set operators have been defined (see Section III-C for details).

Line 11 and line 12 of Figure 2 define a TemperatureSet and SmokeSet set, respectively. Line 13 defines a set of actuators.

C. Event condition action rules

The monitoring and control actions are specified by using event condition action rules. A rule has a name and is composed of three different parts that are *on*, *when* and *then* (line 17 of the grammar). After the *on* keyword there is a list of devices or variables. Whenever one of them changes its value the boolean expression that follows the keyword *when* is evaluated. When this expression is evaluated to true the rule can be applied that is the actions listed after the keyword *action* can be executed. A boolean expression can include relational and logical operators, integers, devices, variables and functions. An action is a list of assignments to variables, physical actuators and logical devices. Special operators are used to support the definition of a boolean condition over a set that are *all*, *any*, *no*, *one* and *lone*. *All* is a universal operator that allows the definition of conditions that must be satisfied by all devices belonging to the set. *Any* is an existential operator that can be used in order to specify that at least one of the element of the set must satisfy the condition. *No* (*one*) is useful when we need to express that no (exactly one) element of the set must verify the specified condition. *lone* is useful when we need to express that at most one element of the set must verify the specified condition.

The rule of Figure 3 implements the fire alarm policy that has been presented in Section II. The *tempAlarmRule* rule is defined over the set temperatureSet. This rule has a condition $any\ temperatureSet > 50$ that is evaluated whenever one of the temperature sensors, that belongs to the temperatureSet set, changes its value. If the condition is true (one of the sensors has temperature greater than 50) the action $tempAlarm = true$ is applied. The tempAlarm variable is a logical sensor that is used in order to signal a high temperature.

IV. Formal verifications of IRON specifications

The application of formal verification techniques to ECA-based programs is essential to support the error-prone activity of defining ECA rules. IRON supports the analysis of dynamic behaviour of ECA rules by first translating

```
1  //Variables are declared in Figure 2
2
3  rule tempAlarmRule on temperatureSet
4    when any temperatureSet >50
5    then tempAlarm = true
6
7  rule smokeAlarmRule on smokeSet
8    when any smokeSet == true
9    then smokeAlarm = true
10
11 rule fireAlarm on tempAlarm, smokeAlarm
12   when tempAlarm and smokeAlarm
13   then sprinklerSet = true
```

Figure 3. Fire alarm system

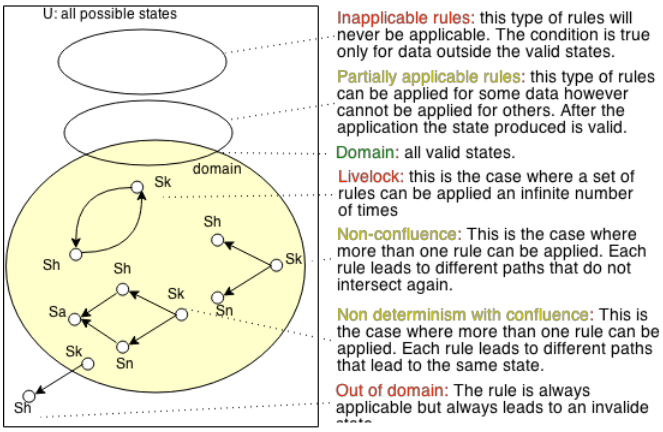

Figure 4. State space, rules and related problems

them into Alloy specifications, then studying the following correctness properties: (i) *termination*; (ii) *confluence*; (iii) *consistency*.

A. Termination

IRON specifications can respond to external and internal events with the application of one or more rules. While external events are generated by physical devices, internal events are generated by the application of actions (e.g., changing the value of a logical device or a variable). An incorrect specification can generate a situation of *livelock* (see Figure 4) where the rules can be applied an infinite number of times. Livelocks can be easily defined by introducing the concept of state, state space (in the following referred to as universe U), domain and stable state.

Definition 1: A state is a n-tuple $(v_1, \ldots, v_n)$ that contains the values of all variables, sensors and actuators. We denote with v_i the value of the variable var_i and with T_{var_i} its type. This can be either integer or boolean.

Definition 2: A universe U (see Figure 4) is the set of all possible states. This is the cartesian product $T_{var_1} \times \ldots \times T_{var_n}$ where T_{var_i} is the type of the variable var_i.

Definition 3: Let U be a universe. A domain D (with $D \subset U$) contains all states that verify the constraints that are defined in the static part.

```
//Variables are defined in Figure 2

rule heatingOn on temperatureSet
  when any temperatureSet < 16
  then heating = true

rule heatingOff on temperatureSet
  when any temperatureSet > 18
  then heating = false
```

Figure 5. Heating program with no livelock

```
//Variables are defined in Figure 2

rule heatingOn on temperatureSet
  when any temperatureSet < 16
  then heating = true

rule heatingOff on temperatureSet
  when any temperatureSet < 18
  then heating = off
```

Figure 6. Heating system with livelock

Definition 4: A state $s = (v_1, \ldots, v_n)$ is a stable state if no rules can be applied.

Definition 5: An IRON program satisfies the termination property when all stable states (that satisfy the conditions of some rules) always lead (with the application of a finite number of rules) to a new stable state.

Figure 5 shows a central heating system program with no livelock. The central heating is turned on when any sensor in $temperatureSet$ reads a temperature that is less than 16 degrees while the heating is turned off when a sensor reads a temperature that is greater than 18 degrees. IRON can be used in order to verify that this policy satisfies the termination property. This is done by translating the IRON program into Alloy specifications. Alloy generates all possible states and connects them with transitions that are labelled with rule names (see Figure 7). More precisely a rule r labels a transition exiting from a state s_1 and entering the state s_2 when from s_1 the application of r leads to the new state s_2 (with $s_1 \neq s_2$). This labelled graph is called labelled transition system and can be analysed in order to check for cycles. If one is found a livelock is reported to the user. The left-hand box of Figure 7 shows some states of the central heating system program with no livelock [2]. A state is a couple where the first element is the value of the temperature sensor and the second value is the state of the heating actuator. Transitions that are labelled with rules are used in order to connect states. For instance when the temperature changes to 15 and the heating is off (the state is $(15, false)$) the rule $heatingOn$ can be applied. The application of this rule changes the state to $(15, true)$. This is represented with a transition that is labelled with $heatingOn$, exits from the state $(15, F)$ and enters the state $(15, T)$. We remark that the program of Figure 5 satisfies the termination property since its labelled transition system has no cycles.

Figure 6 shows a central heating system policy that contains a livelock. The central heating is turned on when a sensor reads a temperature that is less than 16 degrees and the heating is turned off when a sensor reads a temperature that is less than 18 degrees. The right-hand box of Figure 7 shows some states of the central heating system program with livelock. This program does not satisfy the termination property since there is a cycle.

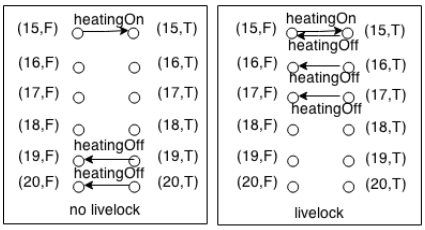

Figure 7. Labelled transition system for the Heating policies with and without livelock

B. Confluence

Confluence is an important property that ensures consistency in systems with concurrent behaviour. A program P, that verifies the termination property, satisfies confluence when, starting from the same state, any order of application of rules will lead to the same stable state.

Figure 8 shows a living room policy that switches both light and TV on when a presence is detected. This program satisfies the confluence property. No matter in which order the rules are applied both light and tv will be on. Figure 9 shows a living room policy with an error. This policy does not satisfy the confluence property. When the rule $LightOn$ is executed first the tv will be on while when the rule $tvOn$ is executed first the tv will be off.

Confluence property is verified in the same way of the termination property (see Section IV-A for details). IRON translate the program into Alloy specifications. Alloy generates a labelled transition system that contains all possible

```
define sensor presenceLiving
define actuator boolean tv=false
//some of the declarations are omitted

rule tvOn on presenceLiving
  when presenceLiving == true
  then tv = true

rule LightOn on presenceLiving
  when presenceLiving == true
  then light = true
```

Figure 8. Living room policy that satisfies the confluence property

[2]For the sake of simplicity we do not show all the transitions and all the states.

```
define sensor presenceLiving
define actuator boolean tv=false
//some of the declarations are omitted

rule tvOn on presenceLiving
  when presenceLiving == true
  then tv = true

rule LightOn on presenceLiving
  when presenceLiving == true
  then tv = false
```

Figure 9. Living room policy with a non-confluence error

```
//Variables are declared in Figure 2

rule tempAlarmRule on temperatureSet
  when any temperatureSet <30 and temperatureSet >30
  then tempAlarm = true

rule tempAlarmRule on temperatureSet
  when any temperatureSet <=1000
  then tempAlarm = true
```

Figure 10. Fire alarm system with unused rules

states and the related transitions. The transition system is checked in order to find non-confluence problems.

C. Consistency

IRON can perform various checks in order to detect rules that are *unused*, *incorrect*, *redundant* and *contradicting*.

1) Unused rules: These are rules that can never be applied. They can be further categorised into inapplicable rules and rules with contradictory premises. A rule that is inapplicable has a condition that is only true for states that are outside the domain (see Figure 7 for a graphical representation). A rule with a contradictory premise has some logical contradiction in its condition thus this can never be true.

Figure 10 shows a fire alarm policy with two rules that are inapplicable and contradictory. The first rule has a contradictory condition that can never be true (the temperature cannot be at the same time less and greater than thirty). The condition of the second rule is true only for values which are outside the domain (i.e., the set of valid states). In fact, we have constrained the temperature variable to be more than -80 (see the declaration of Figure 2 for details). Unused rules are detected by considering, for each rule r, all possible states of the domain. When there is no state that satisfy the condition of r the rule is reported as unused.

2) Incorrect rules: These are rules that can lead to a state that is outside the domain (i.e., an invalid state). They can be further categorised into out-of-domain rules and partially applicable rules. A rule is of the type out-of-domain when its application leads to a state that is outside the domain (i.e., an invalid state). A rule is partially applicable when the following two conditions hold: (i) if the rule is applied to a valid state the execution of the action will bring the system into a new valid state; (ii) the condition is also true for invalid states (i.e., states that are outside the domain). Incorrect rules are detected by considering, for each rule r, all possible states of the domain.

3) Redundant rules: Redundancy is the case where there are rules or chain of rules that are identical. The condition of these rules is always true for the same states and when applied lead always to the same state. A special case of redundancy are subsumed rules where one rule r_1 contains more prerequisites (conditions) of another rule r_2 that is whenever the rule r_1 is applicable then the rule r_2 is also applicable. Redundant rules are detected by considering, for each rule r, all possible states of the domain.

V. IRON IMPLEMENTATION

We have implemented an IRON prototype that is composed of a compiler, a graphic editor, a translator and a middleware.

The IRON compiler is able to perform some syntactic and semantic checks while the graphic editor allows the graphical visualisation/definition of programs. In Figure 11 we show the graphical user interface of IRON. Green and red circles represent boolean read-only devices (sensors) and writable ones (actuators) respectively. Green and red rectangles represent integer read-only devices and writable ones respectively. Transparent circles can be used to group together devices and variables in order to form sets. Grey squares represent rules. An arrow exits a device/variable and enters a rule when this device/variable is used in the condition of the rule. An arrow exits a rule and enters a device/variable when this is updated by the actions of the rule. These graphical interface has been proved very useful when ECA programs need to be written by non-expert programmers.

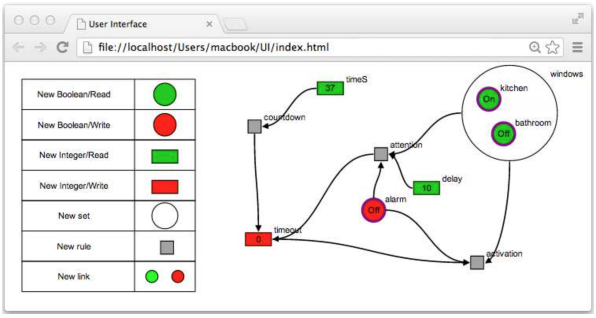

Figure 11. IRON GUI

The translator takes as an input IRON programs and produces Alloy specifications. Alloy is used to verify all properties that are termination, confluence and consistency (see Section IV for details). We have implemented a middleware layer that is capable of receiving sensor readings from different protocols such as ZigBee [11] and tinyOS

[12]. The middleware forwards all the sensor readings to the IRON run time support that applies all the rules. The interaction with actuators is performed via standard APIs that are implemented by using group communication. In the case of tinyOS we have used a clustering approach [13] in order to save energy. We have used xm1000 motes and zigbee hardware. The xm1000 is equipped with temperature, light and humidity sensors and runs tinyOS while smoke detectors and heating control actuators are zigbee devices.

VI. Related work

The event condition action programming paradigm has been extensively applied in wireless sensor networks and actuators. In [1] the authors present an efficient policy system that enables policy interpretation and enforcement on wireless sensors. Their approach support sensor level adaptation and fine-grained access control. In [2] the authors present a rule-based paradigm to allow sensor networks to be programmed at run time in order to support adaptation. The approach presented in [3] describes an event condition action based middleware for programming wireless sensor networks. While all the aforementioned approaches provide quite powerful tools for programming WSNs they do not provide any automatic means of translating programs into formal specifications that can be automatically verified.

Various approaches have been proposed in order to apply formal methods to event-condition-action rules. In [4] the authors translate a set of ECA rules into a Petri Net in order to verify termination and confluence. The approaches presented in [6] and [5] use the model checkers SPIN and SMV in order to verify termination. To the best of our knowledge although various approaches for verifying ECA rules have been proposed they have not been tailored to the context of WSANs. WSANs requires macroprogramming [7], the ability to group sensors in order to perform multicast and broadcast communications. IRON is a language that supports the categorisation of sensors into sets [14], allows the definition of properties over sets and support multicast and broadcast abstractions. IRON allows the automatic verification of ECA rules by translating them into Alloy specifications.

VII. Acknowledgements

This work has been partially supported by the MIUR PRIN project CINA (2010LHT4KM).

VIII. Conclusions

In this paper we present an ECA based language that is called IRON. IRON supports the organisation of sensors and actuators into sets, allows the definition of conditions and actions over sets thus abstracting multicast and broadcast communications. IRON programs are composed of two separate specifications that are static and dynamic. The static part allows the definition of physical and logical devices, sets and variables plus first order constraints. The dynamic part is composed of ECA rules. IRON supports the formal verification of *termination*, *confluence* and *consistency* properties. These are verified by translating IRON programs into Alloy [8] specifications.

References

[1] Y. Zhu, S. L. Keoh, M. Sloman, E. Lupu, N. Dulay, and N. Pryce, "An efficient policy system for body sensor networks," in *14th International Conference on Parallel and Distributed Systems (ICPADS 2008)*, 2008, pp. 383–390.

[2] X. Fei and E. H. Magill, "Reed: Flexible rule based programming of wireless sensor networks at runtime." *Computer Networks*, vol. 56, no. 14, pp. 3287–3299, 2012.

[3] G. Russello, L. Mostarda, and N. Dulay, "A policy-based publish/subscribe middleware for sense-and-react applications," *Journal of Systems and Software*, vol. 84, no. 4, pp. 638–654, 2011.

[4] X. Jin, Y. Lembachar, and G. Ciardo, "Symbolic verification of eca rules," in *International Workshop on Petri Nets and Software Engineering*, 2013.

[5] I. Ray and I. Ray, "Detecting termination of active database rules using symbolic model checking." in *ADBIS*, ser. Lecture Notes in Computer Science, A. Caplinskas and J. Eder, Eds., vol. 2151. Springer, 2001, pp. 266–279.

[6] E.-H. Choi, T. Tsuchiya, and T. Kikuno, "Model checking active database rules under various rule processing strategies." *IPSJ Digital Courier*, vol. 2, no. 13, pp. 826–839, 2006.

[7] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: A survey," *ACM Trans. Sen. Netw.*, vol. 4, no. 2, pp. 8:1–8:29, Apr. 2008.

[8] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2011.

[9] D.-M. Han and J.-H. Lim, "Smart home energy management system using ieee 802.15.4 and zigbee," *Consumer Electronics, IEEE Transactions on*, vol. 56, no. 3, pp. 1403 –1410, aug. 2010.

[10] K. Gill, S.-H. Yang, F. Yao, and X. Lu, "A zigbee-based home automation system," *Consumer Electronics, IEEE Transactions on*, vol. 55, no. 2, pp. 422 –430, may 2009.

[11] S. Farahani, *ZigBee Wireless Networks and Transceivers*. Newton, MA, USA: Newnes, 2008.

[12] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for sensor networks," in *in Ambient Intelligence*. Springer Verlag, 2004.

[13] E. Ever, R. Luchmun, L. Mostarda, A. Navarra, and P. Shah, "UHEED - an unequal clustering algorithm for wireless sensor networks," in *SENSORNETS 2012*, 2012, pp. 185–193.

[14] L. Mostarda, S. Marinovic, and N. Dulay, "Distributed orchestration of pervasive services," in *24th IEEE IAINA 2010, Perth, Australia, 20-13 April 2010*, 2010, pp. 166–173.