Bringing COP to embedded programming

Mikhail Afanasov

May 16, 2013

Abstract. Wireless Sensor Networks (WSNs) are widely applied in different areas, such as health monitoring, wildlife tracking, smart home etc. These days WSNs are used in highly changing environments and the software for WSNs needs to be adaptable but still simple.

Context-oriented programming (COP) is a programming technique which makes it possible to create an adaptive software. Such a software could adapt to changes in the environment and to evolve according to this changes. The term "evolve" means that software can change behavior and this behavior depends on the context in which it is executed. This technique could simplify the complex source-code for WSNs.

In this work we review existing applications of WSN and discuss how COP could be applied to the different scenarios. We also propose methods like context classification, context events and context rules to simplify programming for WSNs.

1 Introduction

WSNs are often deployed in a highly changing environment. It means that a programmer should always care about changes in data acquired by sensors. It leads to increasingly complex source-code. This problem has more significance in mobile WSNs, where we should be aware of location changes. Thus it is hard to implement and maintain such a software. Bringing Context-Oriented Programming in programming for WSN could solve the problem.

Context-Oriented Programming (COP) [**?**] is an approach which provides an opportunity for a programmer to develop a software which can dynamically change the behavior depending on context conditions. Context can be defined as "any information that can be used to characterize the situation of an entity, where an entity can be a person, place or physical object" [**?**]. Context-awareness can be defined as detecting current internal or external state of WSNs.

The context structure for WSNs could be very complex and have nontrivial interaction. But contexts can be divided into groups and considered as classes of contexts. Such classification brings COP on a new level of abstraction and makes it possible to operate complex structures of contexts as a superposition of smaller substructures. It also simplify the integration between different platforms, since another platform could be considered as a new class.

In this work context transition in WSNs are also considered. WSNs should be aware not only of the context, but also of the transition between contexts. In other words, system should perform particular actions in the same time when context is changed. It could be done by providing *context transition behavior*. Thus programmer can specify particular behavior of WSN when context transition occurred. Programmer should also be able to define restrictions and dependencies in context transitions. During the execution there are contexts, which can not be activated while another context is active. If context has dependencies, then it can not be activated while another contexts is deactivated. Context restrictions and context dependency could clarify complex interactions between contexts.

In this work we propose possible solutions of the problems stated above. We review several scenarios of using WSNs and propose how COP can be used in these scenarios. For every scenario we show a context structure and classification, and describe behavior of the WSN. At the end of the paper we discuss possible ways to manage context transitions and restrictions. We also show an implementation of COP using one of the languages for programming for WSNs.

2 Scenarios

WSNs are systems in which numerous tiny sensing devices are distributed over the environment to study it. They are proposed in many engineering, scientific, civil and military applications and widely used to monitor different entities in areas ranging from wildlife [**?**] [**?**] [**?**] to human health-care [**?**] [**?**]. Some of them are discussed in this paper.

2.1 Wildlife tracking.

In biology research there is a necessity to track wildlife. For example, WSN devices can be attached to wildlife species (e.g. zebras [**?**], badgers [**?**] or turtles [**?**]) to study their behavior. Human intervention is highly undesirable in this kind of research [**?**], that is why WSNs are very well suited there.

Community class. Usually the monitored area extends beyond the communication range of nodes. Thus, the network is characterized by intermittent connectivity among nodes [**?**]. But animals are social beings, so it is possible to explore recognizable patterns of movement and community interaction [**?**]. We can detect a group of nodes which is representing a separate community and assume that *community* is a context. An event when node leaving the group or joining to the group could be considered as a change of the context.

Base station class. In the most of WSNs the main goal is to acquire the data from the nodes and to deliver it to the base station. Because of the wide area of monitoring, nodes are not always in the range of the base station. So, it is possible to extract another context *base station*. It is activated when the base station is in the communication range and deactivated when it's not.

Activity class. Species' activity can be characterized by several states such as: sleeping, rest and movement [**?**]. Thereby, we can emphasize *sleeping* (specie isn't moving), *rest* (specie is moving for a short distances) and *movement* (specie is moving for a long distances) contexts.

Battery level class Another aspect of WSNs is power consumption. During the execution we can track the battery level. Within the *low battery level* context node could switch off sensors with low priority.

Contexts' diagram for wildlife tracking scenario is displayed on the figure **??**. Contexts described above can be separated into different classes: *Community class*, *Base station class*, *Activity class* and *Battery level class*. *Community class* indicates whether the node is belonged to the particular community or not. *Base station class* consist of two contexts *Within the BS range* and *Out of BS range*. *Activity class* contains possible activities of specie. *Battery level class* includes *Low level*, *Normal level* and *High level*. Within the separate class contexts transactions can be performed independently. However, there are could be restrictions for contexts' transactions. For example, we could not expect dramatic changes in network topology,

while *sleep* context is activated. Thus, within *Community* and *Base station* classes there are transition condition: if *sleep* context is deactivated, then it is possible to perform transaction within classes mentioned above. Another restriction is a battery level. Context *Within the base station* could be activated only when the node has sufficient battery level, i.e. *High level* or *Normal level* are activated.

While particular transactions between contexts are performed, a node should switch some sensors on or off. For example, if *Within BS range* context is activated, then the node should switch on high range transmitter, but switch it off if *Out BS range* is activated. While *sleep* context is activated, GPS sensor should be switched off. While *low level* is activated, system have to disable sensors with low priority. Thus we should be able to define particular behavior for transactions between context.

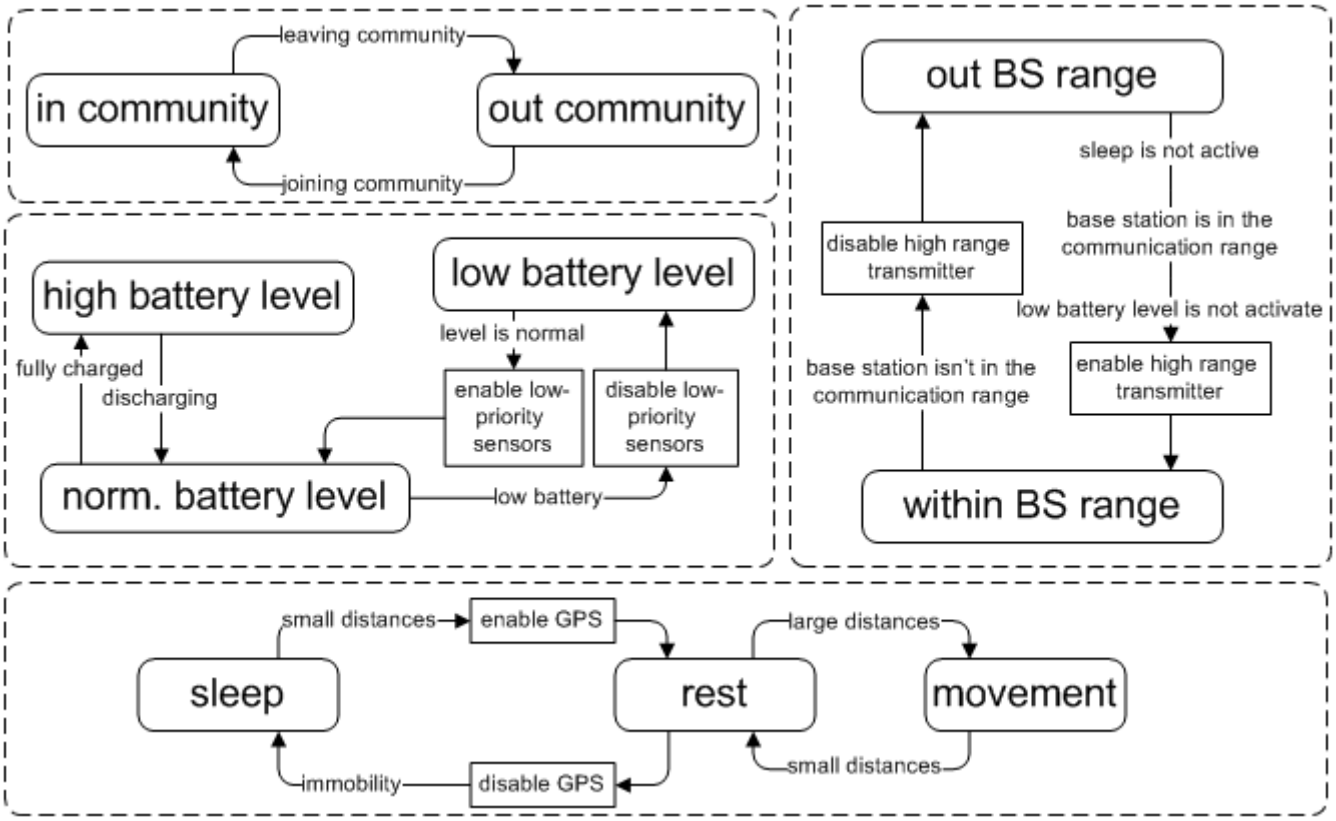

Figure 1: Diagram of the contexts for wildlife tracking.

2.2 Healthcare

Another scenario of WSNs application is healthcare industries. Medical applications of WSNs would improve the existing healthcare and especially for children, elderly and ill. This kind of WSNs should not only provide sensed data, but also identify current condition of the person, in other worlds, identify the context. The context information helps in understanding current condition and taking the situation under control. Context-awareness, defined as "providing relevant information and/or services to the user, where relevancy depends on user's task [**?**]", is a main issue for the health monitoring [**?**]. Depends on tasks, such as: activities of daily living monitoring, fall and movement detection, local tracking, medication intake monitoring, and medical status monitoring [**?**], we can extract possible contexts.

Emergency class. *Emergency* class implies, but not limited to, such events as: falling, heart attack, critical health condition etc. In case of emergency software activates additional sensors

to get more precise information about an event. For example, in case of falling or heart attack, GPS sensor could be activated and emergency message could be sent to appropriate services.

Activity class. This group of contexts consists of such activities as: walking, sitting, laying etc. Usually, if a person is sitting or laying, then sudden falling is not expected. And if *walking* context is activated, software can switch on GPS tracker and try to detect sudden falling by an accelerometer sensor. Software could also track unusual activities. For example, if a person is *sleeping* too long, then *emergency* context could be activated.

Location class. It includes *indoor* and *outdoor* contexts. While GPS works well outdoor and could be used to help people with cognitive disabilities or to identify person's location in emergency situation, it does not works properly indoor [**?**]. Activation of *indoor* context will force software to use another ways to detect person's location. For example, by using Radio Frequency Identification (RFID) markers [**?**].

Figure **??** displays the diagram for possible healthcare contexts. Here *Emergency*, *Activity* and *Location* contexts classes can be highlighted. *Location* class contains *indoor* and *outdoor* contexts. We have to define conditions of contexts transition and behavior during these transitions. Thus, while *indoor* context is active, GPS should be switched off. *Outdoor* context can not be activated while *sleep* context is active. But, if it is activated, then we should enable GPS. *Emergency* class consist of *normal* context, *critical condition*, *falling* and *heart attack*. *Normal* context represents a normal state, but if sensors detects emergency situation like *critical condition*, *falling* or *heart attack*, corresponding context is activated. But *falling* context can not be activated, while *sleeping* context is active. *Activity* class includes contexts *idle*, *moving*, *sitting* and *sleeping*. There are also restrictions: system can not perform a transition from *sleeping* contexts to the *moving* context directly. *Sleeping* context can not be activated while *outdoor* context is active. But if *sleeping* context is activated, system should disable accelerometer sensor, since we can not expect sudden falling. While *moving outdoor*, GPS sensor should be activated for precise location tracking.

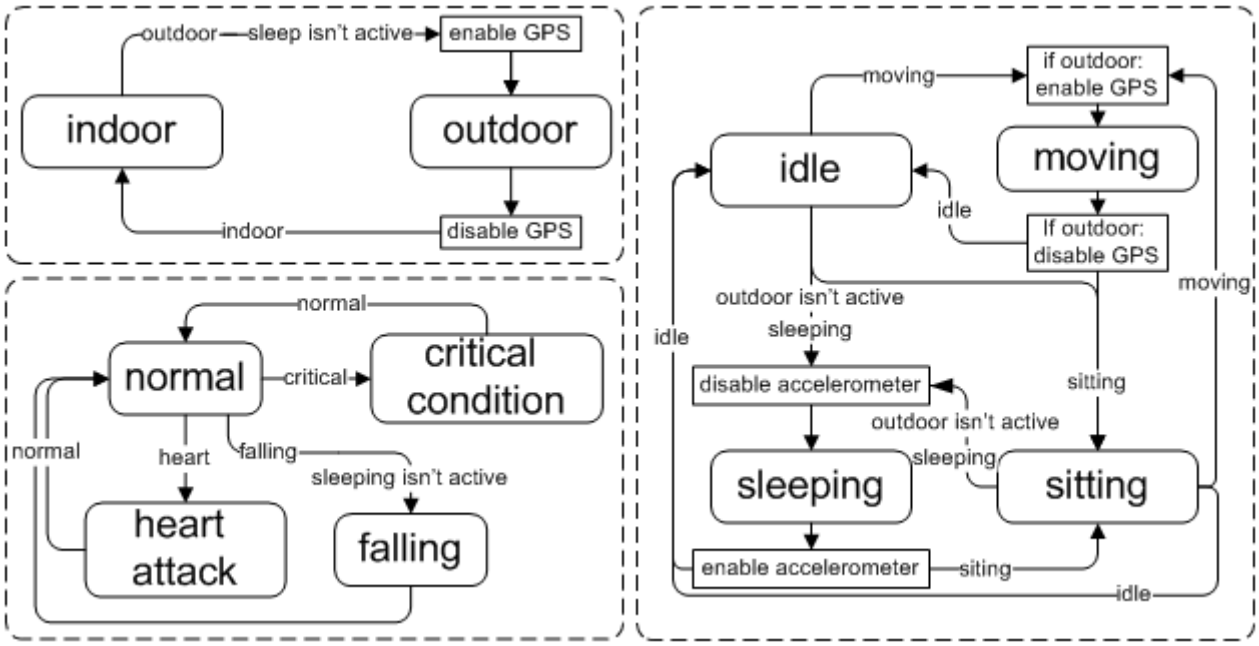

Figure 2: State machine for healthcare contexts.

2.3 Smart home

Humans' life quality could be dramatically increased by using WSNs to build a *Smart Home*. In this scenario WSNs is embedded to the house or a flat to detect current conditions of householders and the environment.

Emergency class. This class is very similar to the *emergency* class discussed in **??**. If WSN detects fire or housebreaking it can send an emergency call. If integrated with *Healthcare system*, it can also activate emergency depends on *Healthcare system* context. For example, if a householder has an injury or heart attack, the WSN can also activate a context from *emergency* class.

Householder's location. WSN can also track a householder's location. For example, if a householder moves from a kitchen to a dining room, the system switches off the light in the kitchen and switches on the light in the dining room. In this context system is also able to detect housebreaking. For example, if an unrecognized person appears in the house, while a householder is sleeping, then WSN activates the *emergency* context. If a householder leaves the house, then *Smart Home* could just lock the door. If a householder is nearby, then it could open the door.

Health monitoring. WSN can be programmed to detect specific conditions and activate a specific context. For example, if the temperature in the room is lower (or higher) than normal, system can enable air heating (or conditioning) to return the temperature to the normal state. If *Healthcare System* is integrated to *Smart Home*, so the system can notify the householder about his current condition and recommend activities to improve householder's state of health.

Figure **??** shows a state diagram of possible contexts of *Smart Home*. In this application three classes could be found: *Emergency* class, *Location* class and *Climate* class. Contexts of *Emergency* class are activated if system detects intruders, fire or health emergency. There is a necessity to perform particular actions during context transition. For example, to call the police, the fire or an ambulance. *Location* class includes *outdoor* and rooms in house. During context transitions within *location* class, system have to turn on/off lights and lock/unlock door. *Climate* class could contains temperature contexts. If system detects high or low temperature, then it enables conditioner or heater.

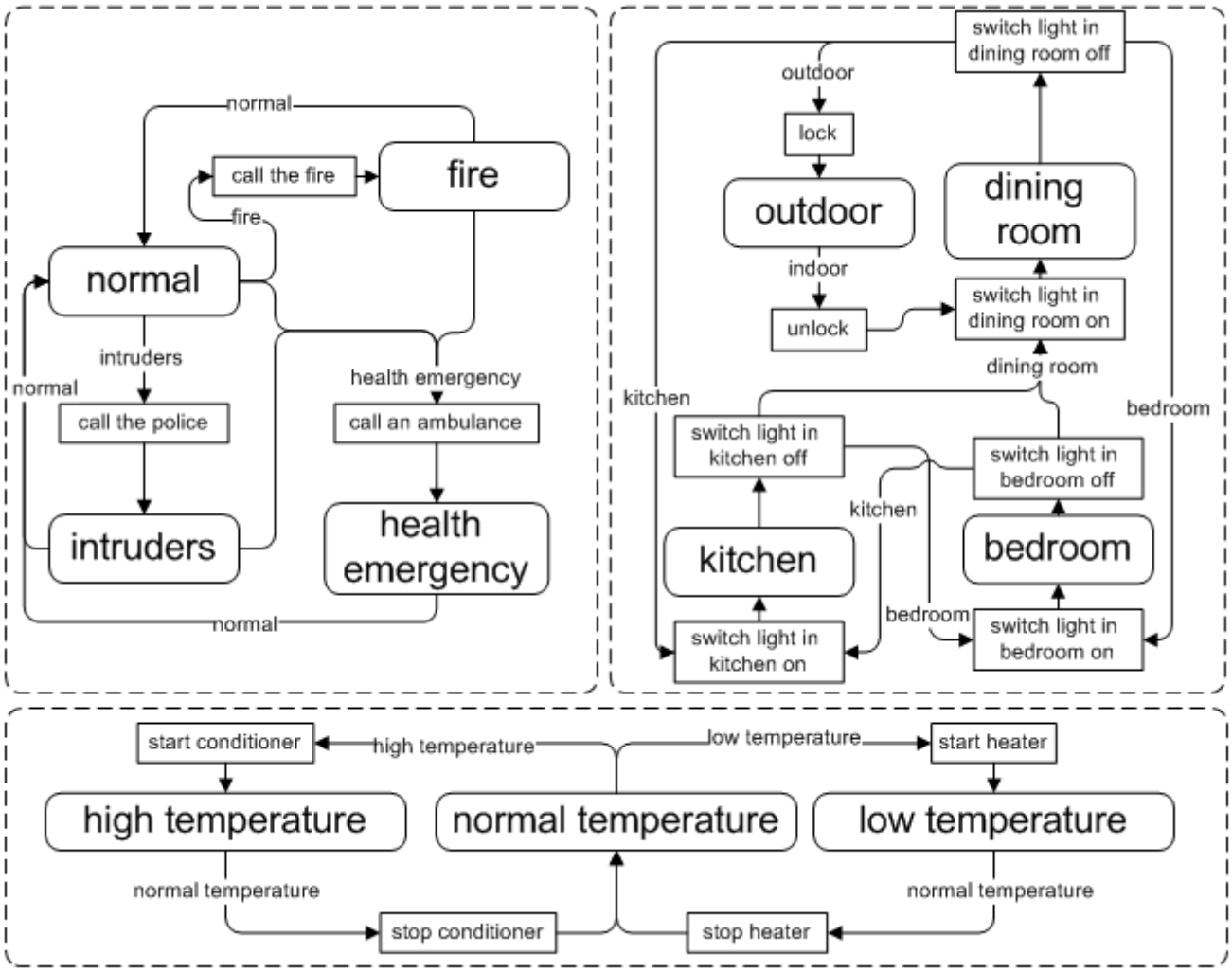

Figure 3: A state diagram for smarthome.

3 Context Recognition

Regardless the scenario, we can emphasize several context detection mechanisms. The first step of context recognition is acquiring the low-level sensor readings. Considering these readings, we can then extract the context [**?**].

Context-Aware Packets method allows single sensor to share sensed data in order to detect meaningful context model from manifold inputs [**?**]. This mechanism assumes that every node can send a packet with data acquired by node to the base station. Base station then analyzes the data and recognizes the context.

Artificial Neural Networks (ANNs) method is used as a solid clustering algorithm for context-awareness in sensor networks [**?**]. The low-level sensors produce a noise while acquiring the data. While ANNs still perform well despite of nose, they also add a new context to the system without intervention from a user.

Bayesian Networks (BNs) method makes it possible to estimate current context statistically [**?**]. It relies on the intensity of the sensing data. The higher the intensity, the more significant the data is.

4 Discussion

Most of the implementations of WSNs consider that context is managed by human or by program on a remote computer or a base station, but there are cases when wireless nodes should manage the context by themselves. For example, in wildlife tracking nodes could be out the range of the base station for a long time, but it is still necessary to manage the context.

While global contexts are described, we need to create tools to manage the context on a low hardware level, like wireless sensor node. We didn't consider decision making process, but focused on managing the behavior within the given context. In a trivial case, programmer activates context manually within the program.

As was considered in a previous sections, different context and context classes (groups) can be extracted in different scenarios. There are also actions should be performed before activation and after deactivation of the context. Each context also has transition rules, i.e. transition from one context to another can be performed if it satisfies the particular conditions.

According to scenarios described in the section **??**, four types of constructs should be defined: *context event*, *context transition*, *check* and *trigger*. *Context event* occurs if sensed data satisfies one of the conditions. *Context transition* is the fact of the transitions between contexts and treated as an *event*. *Trigger* is a condition which triggers the transition between contexts, *check* verifies if the transition is possible or not. In other words, we have two types of events and two types of transitions rules. In the figure **??** different combinations of rules and events are shown. Thus, *context event* (or sensed data) as well as *context transition* can *trigger* the transition between contexts. But we also need to verify validity of such a transition. So we can use *context event* and *context transition* as *checks*. In other words, transition can be performed only if another *context transition* has been performed or if another *context event* has been occurred.

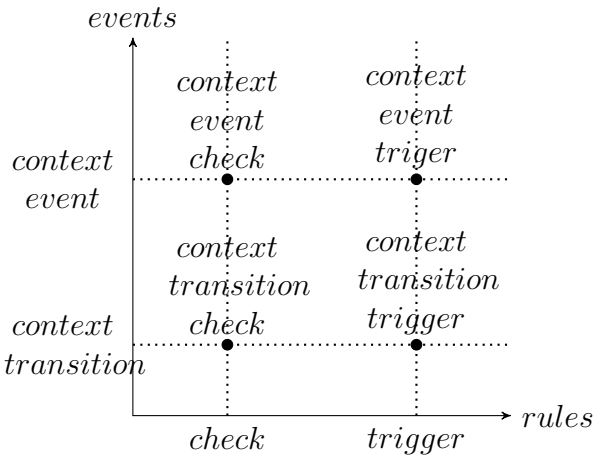

Figure 4: Events and rules.

Context class can be considered as a state machine, where context is a state. In each scenario could be several state machines. Context transitions can be considered as changes of a state of the state machine. Context transitions have rules and constraints, thus state can not be changed if the conditions are not satisfied. Context transitions are initiated by triggers. There are also actions which should be performed before activation and after deactivation of the context. Thus, we can emphasize main requirements of the context-oriented language for embedded systems. Language should provide tools to:

- Define context

- Define context-depended functions.
- Define context configuration
- Define actions before activation and after deactivation
- Define rules of context activation
- Define triggers

5 Implementation

We are using TinyOS and nesC language for demonstration purposes. In order to use context-oriented paradigm, we added new key-words in the language. We are also extended components by *context* and *context configuration*. *Context* is an analog of *module* and *context configuration* is an analog of *configuration*. These two components can be used as other components in nesC language.

Context components is used to define context, possible transitions, actions before activation and after deactivation, layered functions, and to check transition conditions. In the rest it can be used like a *module*.

```
1  context High {
2    transitions Normal;
3    uses interface Leds;
4  }
5  implementation {
6    event void activated() {
7    }
8    event void deactivated() {
9    }
10   bool check() {
11     return TRUE;
12   }
13   layered void toggle_leds() {
14     call Leds.set(1);
15   }
16 }
```

Figure 5: Context component.

On the figure **??** context definition is displayed. Key word *transitions* on the line 2 define legal transitions from context *High*. It means, that only *Normal* can be activated as the next context after *High*. This definition is not mandatory. If *transitions* are not defined, any transitions are legal. Line 3 represents regular nesC key word and means, that we are using *Leds* component. Event *activated* (line 6) is fired just before context activation, while event *deactivated* (line 8) is fired just after deactivation. These events are also not mandatory. Method *check()* represents *context event check*. This method is called to verify conditions of context activation. If *check()* returns FALSE, context will not be activated. If method is not defined, conditions will not be verified. Key-word *layered* (line 13) indicates implementation of the function, which is specific for the current context. *Layered* functions implementation is mandatory if they are defined in context configuration.

In the *context* component *context transition check* and *context transition trigger* can also be specified. For example, on the line 2 (figure **??**) key-word *while* declares, that transition from

Indoor to *Outdoor* context can be activated only if *Activity.Moving* context is active, where *Activity* is a context class.

```
1 context Indoor {
2   transitions Outdoor while Activity.Moving;
3 }
4 implementation {
5 }
```

Figure 6: Context transition rule example.

Programmer can also specify *Context transition trigger* within the *context* declaration. For example, if we have integration between two WSNs *Smart Home* and *Health Care*, programmer may want to activate *SamrtHome.Emergency* context if *HealthCare.Emergency* context was activated. It can be done as it shown on the line 2 of the figure **??**.

```
1 context Emergency {
2   triggers SmartHome.Emergency;
3 }
4 implementation {
5 }
```

Figure 7: Context transition trigger example.

Context configuration component is used to define layered functions and context configuration. In the rest it can be used like a regular *configuration*.

```
1  context configuration Temperature {
2    layered void toggle_leds();
3  }
4  implementation {
5    contexts High,
6      Normal is default,
7      Low;
8    components LedsC;
9    High.Leds -> LedsC;
10   Normal.Leds -> LedsC;
11   Low.Leds -> LedsC;
12   Error.Leds -> LedsC;
13 }
```

Figure 8: Context configuration component.

In the listing **??** we assume, that contexts *High*, *Normal* and *Low* are already defined. *Error* context is default and generated by compiler, however it can be overridden by programmer in a standard context definition way. On the line 2 we declare a layered function, which should be implemented in each context. Lines 5, 6 and 7 show contexts belonging to the *Temperature* class. It also indicates, that all declared contexts should implement layered functions. Line 6 also indicates that *Normal* context will be active after initialization. The rest of the listing is a regular declaration of components and wires.

Context and *context configuration* has nesC-like structure and utilize the same logic. Both components can be used in native nesC way as a regular components.

```
1  module DemoC {
2    uses context configuration Temperature;
3    uses interface Boot;
4  }
5  implemetation {
6    event vois Boot.booted() {
7      activate Temperature.Low;
8      call Temperature.toggle_leds();
9    }
10   event void Temperature.contextChanged(context_t con) {
11   }
12 }
```

Figure 9: Components usage example.

Figure **??** displays a simple example. Here we declare, that we are using *context configuration Temperature* (line 2). Key-word *activate* (line 7) is used to activate a particular context in a particular context class. Here we activate context *Low* of the context class *Temperature*. Then we call a layered function (line 8). Behavior of the layered function depends on the activated context. Event *contextChanged()* (line 10) is fired if the context within the particular context class has been changed.

```
1  Configuration DemoAppC {
2  }
3  implementation {
4    components
5      Temperature,
6      MainC,
7      DemoC;
8    DemoC.Temperature -> Temperature;
9    DemoC.Boot -> MainC;
10 }
```

Figure 10: Components usage example (configuration).

On the figure **??** main configuration is displayed. Here (lines 5 and 8) we use context configuration *Temperature* as a regular component.

6 Future Work

In the previous section we introduced new components and key words in the nesC language, but modifications are very slight and very well fitted. In future work we will focus on the source-to-source compiler. Nevertheless, it is possible to mention main ideas of transformation from the *context-oriented nesC* to the native nesC.

Following listings are provided to compare context-oriented nesC sources and generated native nesC sources. Lines with red background will be removed in generated source. Green color indicates, that highlighted lines will be added to the generated source. Blue highlighted lines will be slightly modified.

```
1  context High {
2   transitions Normal;
3     uses interface Leds;
4  }
5  implementation {
6   event void activated() {
7     }
8   event void deactivated() {
9     }
10  bool check() {
11       return TRUE;
12    }
13  layered void toggle_leds() {
14       call Leds.set(1);
15    }
16 }
```

```
1  module HighTemperatureContext {
2   provides interface ContextCommands as Command;
3   provides interface TemperatureLayer as Layer;
4   uses interface ContextEvents as Event;
5     uses interface Leds;
6  }
7  implementation {
8   event void Event.activated() {
9     }
10  event void Event.deactivated() {
11    }
12  command bool Command.check() {
13       return TRUE;
14    }
15  command void Layer.toggle_leds() {
16       call Leds.set(1);
17    }
18  command void Command.activate() {
19  signal Event.activated();
20  }
21  command void Command.deactivate() {
22  signal Event.deactivated();
23  }
24 }
```

Figure 11: Context nesC source (left) and native nesC (right).

```
1  module DemoC {
2   uses context configuration Temperature;
3     uses interface Boot;
4  }
5  implemetation {
6     event vois Boot.booted() {
7   activate Temperature.Low;
8   call Temperature.toggle_leds();
9     }
10  event void Temperature
    .contextChanged(context_t con) {
11    }
12 }
```

```
1  #include "Contexts.h"
2  module DemoC {
3   uses interface ContextGroup as TemperatureGroup;
4   uses interface TemperatureLayer;
5     uses interface Boot;
6  }
7  implementation {
8     event Boot.booted() {
9   call TemperatureGroup.activate(LOW);
10  call TemperatureLayer.toggle_leds();
11    }
12  event void TemperatureGroup
    .contextChanged(context_t con) {
13    }
14 }
```

Figure 12: Context nesC source (left) and native nesC (right).

```
1  Configuration DemoAppC {
2  }
3  implementation {
4    components
5  Temperature,
6       MainC,
7       DemoC;
8  DemoC.Temperature → Temperature;
9    DemoC.Boot -> MainC;
10 }
```

```
1  configuration DemoAppC {
2  }
3  implementation {
4    components
5  TemperatureConfiguration,
6       MainC,
7       DemoC;
8  DemoC.TemperatureGroup → TemperatureConfiguration;
9  DemoC.TemperatureLayer → TemperatureConfiguration;
10   DemoC.Boot -> MainC;
11 }
```

Figure 13: Context nesC source (left) and native nesC (right).

We don't provide listing of translation *context configuration Temperature* because of big size of the generated source. Using *context configuration Temperature* compiler generates *TemperatureLayer* interface, which includes layered functions as commands; *TemperatureGroup* module, which provides TemperatureLayer interface and manages contexts (i.e. activates and deactivates); *TemperatureConfiguration* configuration, which wires context components between each other and third party components; *Contexts.h* file, which contains information about contexts and definition of the type *context_t*.

As we can notice, we made very slight changes in native nesC, but extended capabilities of the language. We also can notice strong rules in translation, which can be used to implement source-to-source compiler.

7 Related Work

The techniques for context-oriented programming were reviewed in details by Slavaschi et al. [?]. Some of the techniques were utilized in the implementation of the COP in NesC.

In NesC there are no tools for dynamic loading and for object-oriented programming, so it is not possible to use *in-laguage approach*, thus we are using *source-to-source compiler* as it was proposed in [?]. We are also using an idea of *Layer in class*[?]. However, there are no classes in NesC, so the idea was slightly chaged to fit the language restrictions. In context-oriented NesC each *context* in *context group* provides different implementation of layered functions. Also, several *context groups* could provide the different sets of layers. Thus we define a set of layers within the one *context group* to provide a behavioral variation.

In context NesC a modification of *per-object* activaton method is used. This method also was reviewed in [?]. But instead of objects there are *components* in NesC. Thus activation occurs within the one component, which represents a *context group*. This type of activation allows to keep a behavioral consistency, while adding behavioral variation.

This work was also partially inspired by [?], where event-based context activation and context transitions were proposed. But there are also main differences, which makes this work original in the area.

While in [?] and [?] authors were focused on Java-based platforms and object-oriented languages, we are focused on embedded programming and on mobile WSNs particularly. In this work we consider embedded software platforms, which are usually characterized by restricted capabilities and limited resources. Thus, specific for WSNs optimization and adaptation should be performed. We also should to take into account the sensed data.

In [?] *event declaration* and *layer transition rules* were proposed. Considering *layer* as a *context*, we can ephasize another significant difference. While in [?] rules are only considered

as activators and deactivators of the contexts, this work divide the term *rule* into *check* and *trigger*. We also consider not only *context transition* but also *context event*, in other words, *events* are expanded to cover the impact of the sensed data in WSNs. Thus we extend constructs proposed in [**?**].

8 Conclusion

In this paper we discussed several WSNs applications and proposed possible contexts for different scenarios. Software for mobile WSNs becomes more complex, which makes it difficult to implement and to maintain. COP is intended to make a software for WSNs more adaptable and less complex. We extracted different contexts within different scenarios and showed that COP could be used in applications for WSNs. We also considered context detection and possible behavior of the software within the particular context. This is the first step to bringing COP to the programming for WSNs.

But even after extracting contexts, we can see a very complex structure and interactions between contexts. Complicity could be reduced by contexts classification. Moreover, classification makes it possible to provide integration between platforms easier: we can consider other platform as a separate class of contexts. For example, *Smart home* could extract *Healthcare* contexts to adapt home environment or to enable emergency. We also showed a necessity of defining a behavior not only within particular context, but also when transition between contexts occurs. There are also restrictions in context transitions, which also should be defined. Thus, COP for embedded programming should provide mechanisms of defining context classification, transitions and restrictions.

While transitions and restrictions are already proposed in [**?**] and can be implemented by event-based COP, context classification is not yet proposed. Classification, context transitions and restrictions could make platform integration and programming for WSNs easier and more effective.

We are also proposed a language extension for nesC, which introduces new key-words and types of components. As shown in this paper, our approach not only extend language capabilities, but also simplify developing process by generating the source code.

References

[1] Alemdar H., Ersoy C. "Wireless sensor networks for healthcare: A survey", 2010

[2] Chang Y., Chen C., Chou L., Wang T., "A novel indoor wayfinding system based on passive RFID for individuals with cognitive impairments", Second International Conference on Pervasive Computing Technologies for Healthcare, 2008, pp. 108–111.

[3] Dey A. K., Abowd G. D. "Towards a better understanding of context and context-awareness", 1st International Symposium on Handheld and Ubiquitous Computing, 1999.

[4] Dey A. K., Salber D., Abowd G. D., Futakawa M., "The Conference Assistant: Combining Context-Awareness With Wearable Computing',' Proceedings of the 3rd In-ternational Symposium on Wearable Computers, San Francisco, 1999, pp. 21-28.

[5] Dyo V., Ellwood S. A., Macdonald D. W., Markham A., Mascolo C., Pasztor B., Trigoni N., Wohlers R. "Poster Abstract: Wildlife and Environmental Monitoring using RFID and WSN Technology.", InProc. of the Int. Conf. on Embedded Networked Sensor Systems (Sen-Sys), 2009.

[6] Gorlick A. "Turtles to test wireless network", July 2007.

[7] Hirschfeld R., Costanza P., Nierstrasz O. "Context-oriented programming.", Journal of Object Technology, 2008

[8] Juang P., Oki, Y. Wang H., Martonosi M., Peh L. S., Rubenstein D. "Energy-Efficient Computing forWildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet." In Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), 2002.

[9] Kamina T., Aotani T., Masuhara H. "Designing event-based context transi-tion in context-oriented programming.", Proceedings of the 2nd International Workshop on Context-Oriented Programming, COP'10, ACM, New York, NY, USA, pp. 2:1–2:6, 2010.

[10] Kamina T., Aotani T., Masuhara H. "EventCJ: a context-oriented programming language with declarative event-based context transition.", Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD '11, ACM, New York, NY, USA, pp. 253–264, 2011

[11] Korel B.T., Koo S. G. M. "A Survey on Context-Aware Sensing for Body Sensor Networks" 2010

[12] Lindgren A., Mascolo C., Lonegan M., McConnell B. "Seal2Seal: A Delay-Tolerant Protocol for Contact Logging in Wildlife Monitoring Sensor Networks." InProc. of Int. Conf. on Mobile Ad-hoc and Sensor Systems (MASS), 2008.

[13] Lorincz K., Chen B.-R. , G. Challen W. G., Chowdhury A. R., Patel S., Bonato P., Welsh M. "Mercury: A Wearable Sensor Network Platform for High-fidelity motion Analy-sis." InProc. of the Int. Conf. on Embedded Networked Sensor Systems (SenSys), 2009.

[14] Lukac M., Girod L., Estrin D. "Disruption Tolerant Shell." InProc. of the SIGCOMM Wkshp. on Challenged Networks (CHANTS), 2006.

[15] Michahelles F., Samulowitz M., Schiele B. "Detecting Context in Distributed Sensor Networks by Using Smart Context-Aware Packet" 2002.

[16] Ng J.W.P., Lo B.P.L., Wells O., Sloman M., Peters N., Darzi A., Toumazou C., Yang G. "Ubiquitous monitoring environment for wearable and implantable sensors (UbiMon)", in: 6th International Conference on Ubiquitous Computing, 2004.

[17] Pasztor B., Mottola L., Mascolo C., Picco G. P., Ellwood S., Macdonald D. "Selective Reprogramming of Mobile Sensor Networks through Social Community Detection" 2008

[18] Salvaneschi G., Ghezzi C., Pradella M. "Context-oriented programming: A software engineering perspective", The Journal of Systems and Software, p.1801–1817, 2012.