Bringing Context-Oriented Programming into Embedded Systems

First Author
AuthorCo, Inc.
123 Author Ave.
Authortown, PA 54321 USA
author1@anotherco.com

Fifth Author
AuthorCo, Inc.
123 Author Ave.
Authortown, PA 54321 USA
author5@anotherco.com

Copyright is held by the author/owner(s).
UbiComp '13 Adjunct, Sept 8-12, 2013, Zurich, Switzerland.
ACM 978-1-4503-2139-6/13/09...$15.00.

Abstract

Programming for embedded systems makes a significant part in the modern software development. Despite the rapid growth of the high-level languages there are still a necessity to use embedded systems. The prime example of this are Wireless Sensor Networks (WSNs). Most of the platforms for them are embedded.

The main task of WSNs is to monitor the environment. Thus, they are widely applied in different areas, such as health monitoring, wildlife tracking, smart home etc. But the environment is changing very fast, so it is necessary to react to these changes. In order to make WSN to be aware of changes we have to use a representation of the state of the environment within the WSN. In other words, we have to use *Context* in programming for WSNs.

Context could be defined as an entity, which represents a state of the environment. It can be used as a simplified model of the environment and could be formed according to sensing data. It is possible to develop more autonomous and more flexible system by using *Contexts* in programming for WSNs.

Context-oriented programming (COP) is a programming technique which makes it possible to create an adaptive software by using *Context* as an information about the state. 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 behaviour and this behaviour depends on the context in which it is executed.

In this work we show how COP could be used for programming for embedded platforms and for WSNs in particular. We overview an example of the possible application of the WSN, then we extract the structure of the context for this application to highlight the specific aspects of the context-oriented approach in embedded systems. In the end we propose a small extension of the existing language for WSNs.

Introduction

Embedded systems are widely used to build different kind of devices and composition of devices. For example, in WSNs embedded platforms are mostly used. WSNs are inherently environment dependent. It means that a programmer should always care about changes in data acquired by sensors. It leads to increasingly complex source-code. Even by using a representation of the state of the environment, it is hard to avoid complexity without elaborated tool. Thus it is hard to implement and maintain such a software.

In most cases WSNs are used as a passive components to gather data and to send it to the base station, where it is precessed. But there are also cases when WSN could be out of the range of the base station for a long period of time. In that case WSN should handle the changes in the external state (like changes in environment) and in the internal state (like exceptions within the program). In other words, WSN should be more autonomous and adaptive.

Context-Oriented Programming (COP) [**?**] is an approach which provides an opportunity for a programmer to develop a software which can dynamically change the behaviour 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.

Context-Oriented Programming could be used as an effective way to use a *Context* as a representation of the environment. It also will help to avoid the increasing complexity of a program. But bringing COP into embedded systems is accompanied by following challenges.

- While COP could provide a convenient way to use an environment representation in WSNs, the context structure for WSNs could be very complex and have non-trivial interaction. Thus, initial Context-Oriented paradigm could not be applied as is.
- Embedded platforms are mostly static, there are no tools for object-oriented or dynamic programming. Therefore, COP approaches for high-level languages (like Java, C++ or Python) are not applicable.
- Embedded platforms also have restrictions on a hardware level, such as low memory, power consumption constraints, low performance.

Let’s consider a simple example of the application of a WSN.

Figure 3 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

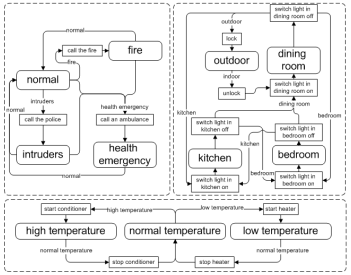

Figure 1: Insert a caption below each figure.

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

Figure 2: Insert a caption below each figure.

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.

Emergency class. 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.

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.

ConesC

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*.

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

```
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 3: Insert a caption below each figure.

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 5) 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.

```
context Indoor {
  transitions Outdoor while Activity.Moving;
}
implementation {
}
```

Figure 4: 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 5.

[H]

```
context Emergency {
  triggers SmartHome.Emergency;
}
implementation {
}
```

Figure 5: 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*.

```
context configuration Temperature {
  layered void toggle_leds();
}
implementation {
  contexts High,
    Normal is default,
    Low;
  components LedsC;
  High.Leds -> LedsC;
  Normal.Leds -> LedsC;
  Low.Leds -> LedsC;
  Error.Leds -> LedsC;
}
```

Figure 6: Context configuration component.

In the listing 6 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.

```
module DemoC {
  uses context configuration Temperature;
  uses interface Boot;
}
implemetation {
  event vois Boot.booted() {
    activate Temperature.Low;
    call Temperature.toggle_leds();
  }
  event void Temperature.contextChanged(context_t con) {
  }
}
```

Figure 7: Components usage example.

Figure 7 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.

```
Configuration DemoAppC {
}
implementation {
  components
    Temperature,
    MainC,
    DemoC;
  DemoC.Temperature -> Temperature;
  DemoC.Boot -> MainC;
}
```

Figure 8: Components usage example (configuration).

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

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. As one 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. In future work we will focus on the source-to-source compiler.

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 [**?**].

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.