Towards Context-Oriented Programming in Wireless Sensor Networks

Mikhail Afanasov
Politecnico di Milano, Italy
afanasov@elet.polimi.it
Luca Mottola
Politecnico di Milano, Italy and
Swedish Institute of Computer
Science
luca.mottola@polimi.it
Carlo Ghezzi
Politecnico di Milano, Italy
carlo.ghezzi@polimi.it

UbiComp'13 Adjunct, September 8–12, 2013, Zurich, Switzerland.
ACM 978-1-4503-2215-7/13/09.

http://dx.doi.org/10.1145/2494091.2494141

Abstract
We present our ongoing work towards applying the context-oriented programming (COP) paradigm to wireless sensor networks (WSNs). Context—as a representation of the environment where the system operates—plays a key role in WSN applications, which must often adapt their operation depending on environmental conditions. We argue that promoting a notion of context as a first-class citizen in WSN programming facilitates the design and implementation of context-dependent functionality. To this end, we conceive a context-oriented programming model expressly tailored to WSNs, coupled with dedicated language constructs. Unlike the existing literature on COP, we embed the latter within low-level C-like languages that do not rely on resource-intensive features such as dynamic memory management. To make our design concrete, we describe a context-oriented extension of nesC—a widely used WSN programming language—and report on a preliminary assessment of our design.

Introduction
The processing in WSN applications is often intimately tied to the environment the system is immersed in. Consider for example a health monitoring application [4], whose goal is to continuously monitor an individual's physical conditions. Body-worn sensors track quantities such as heart rate and body movements. In normal conditions, the

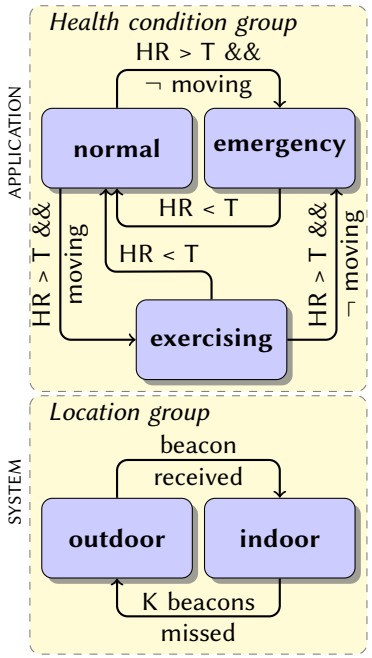

Figure 1: Context diagram for the example health care application. (HR stands for heart rate, T is a safety threshold for heart rate, K is the number of consecutive beacons to be missed before the node detects to be outdoor).

monitoring rate may be limited to save energy. Whenever a potentially harmful condition is detected, the sensors increase the monitoring rate to obtain finer-grained information. Orthogonal to application-level functionality, the system should periodically transmit sensed data to a back-end whenever a network infrastructure is within reach, such a residential Internet gateway, or locally log data in infrastructure-less situations.

Designing and implementing similar applications is currently intricate. The number of possible different situations that developers must account for may be prohibitively large. The problem is often exacerbated by the interplay between application-level and system-level functionality, which may often be a function of independent environmental quantities, as in our example health monitoring application. As a result, implementations become entangled and thus difficult to re-use, to debug, and to maintain.

We argue that promoting a notion of *context* as a first-class citizen in WSN programming facilitates the design and implementation of environment-dependent functionality. Context here indicates "any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical object" [1]. In this work, we leverage concepts of Context-Oriented Programming (COP) [3] to support programmers in developing software that can dynamically change its behavior depending on context information.

COP has already been applied to high-level languages such as Java and Pyhton. Our goal is to enable COP within low-level languages WSN-specific lanuages. This is, however, accompanied by several challenges:

- As our health-care application demonstrates, in WSNs different context information can have complex interactions and interconnections; it is then necessary to use custom abstractions and expressly tailored programming techniques.
- Traditional WSN programming languages favor static memory management and lack features such as object-orientation and dynamic programming; therefore, COP approaches designed for high-level languages like Java or Python are not applicable.
- WSN platforms feature constrained hardware resources, such as small amounts of memory, limited computation facilities, and power consumption constraints; the solution to the challenges above must thus live within such limitations.

We describe next how we meet these challenges. At the core of our solution is a dedicated programming model and language called ConesC. The latter extends the nesC [2] language for sensor networks with COP constructs. Particularly, we enable a notion of *layered function* [3] in nesC. Programmers provide different implementations of the *same* layered function that specify different behaviors depending on context information. The system then automatically dispatches function calls to the "right" implementation, that is, the one associated with the currently perceived context. This greatly simplifies implementing context-dependent functionality, rendering code simpler to read, to debug, and to maintain.

ConesC

We describe next the concepts and programming model of ConesC, along with a brief report of ongoing work and preliminary results.

```
context configuration Location {
❷ layered command void report();}
implementation {
❹ contexts Outdoor,
❺   Indoor is default;
 components Routing, Logging,
 LedsC;
 Indoor.Collection -> Routing;
 Outdoor.DataStore -> Logging;
 Error.Leds -> LedsC;}
```

Figure 2: Context configuration component.

Concepts

Programming in ConesC revolves around two key concepts: *i)* context groups, and *ii)* individual contexts and their transitions. The latter represent the different environment situations a system may find itself in. Each context maps to a different application-level or system-level processing. As the environment surrounding the system mutates, programmer triggers transitions from a context to a different one by *activating* a given context. Context groups, on the other hand, represent collections of individual contexts sharing common characteristics; for example, whenever transitions among the involved contexts are determined by changes in the same physical quantity.

Figure 1 shows how a design of the example health-care application maps to these concepts. At application level, the *health condition* context group includes three individual contexts representing a person's possible health conditions. In this example design, we deploy heart rate and acceleration sensors to determine a context. Starting from a context corresponding to *normal* conditions, a growth in heart rate may correspond to an *emergency* context if the person is not moving, or simply to an *exercising* context should the acceleration sensor record significant movements.

At the system level, the *location* context group includes two individual contexts representing whether a person's device is *indoor*, where the system can rely on some infrastructure to funnel data to a back-end, or rather *outdoor*, where no infrastructure is available. We can design transitions between these individual contexts as a function of periodic beacons received from the infrastructure.

As we describe next, the organization of context information in context groups and individual contexts is useful as a foundation for programming with ConesC. Moreover, it may also serve for automatically generating code skeletons and for statically verifying the possible system executions against safety or liveness properties, e.g., using model-checking techniques. We plan to explore these opportunities in the near future.

Programming Model and Language

We build upon nesC, a widespread sensor network programming language [2]. NesC itself extends the C language by providing component-based programming and a notion of split-phase execution. In nesC, *commands* are function calls that return immediately. The results of the corresponding operations, if any, is returned by means of asynchronous *events*.

We design additional language constructs for nesC to map the notions of context groups, individual contexts, and transitions among them. Context groups map to *context configurations*. These extend the traditional nesC configuration by specifying, in addition to the wiring among components, the necessary layered functions—being commands or events—and the individual contexts that provide context-specific implementations for such functions.

Figure 2 shows an example context configuration in ConesC, corresponding to the location context group of Figure 1. We define a layered report command (line ❷) that abstractly specifies the processing to report sensed data, and whose concrete implementation is delegated to the individual contexts. The individual contexts belonging to the group are declared with the keyword contexts (line ❹), including a tag is default to indicate the context to activate after initialization. Moreover, each context group has an Error context, which is activated if illegal context transitions are triggered, as we describe next. The

```
context module Indoor {
❷ transitions Outdoor;
  uses interface Collection;}
implementation {
❺ event void activate(){
    //...}
❼ event void deactivate(){
    //...}
❾ command bool check(){
    return TRUE;}
❶❶ layered command void report(){
    //...
    call Collection.send(msg);}}
```

Figure 3: Context component.

```
// in a periodic loop...
call Location.report();
// in a parallel task
activate Location.Indoor;
```

Figure 4: Examples of context transition and layered function call.

implementation of layered functions for the Error context is automatically generated by the ConesC tool-chain, but programmers can redefine it if necessary. The remainder of the context configuration specifies the components used and their wiring using standard nesC syntax.

The individual contexts map to *context modules*. These extend normal nesC modules by additionally specifying the outgoing transitions from an individual context, the context-specific implementation of layered functions, and additional support functions to check whether the execution is allowed to activate the context and to perform operations when activating/deactivating the context.

Figure 3 shows the context module for the individual Indoor context of Figure 1. Outgoing transitions from this context are defined with the transitions keyword (line ❷): if the execution activates a context where no outgoing transition from the current context exists, the execution moves to the Error context in the pertaining context configuration, as in Figure 2. If transitions are not defined, any transition is legal. By implementing events activate (line ❺) and deactivate (line ❼), programmers can optionally specify initialization and clean-up functionality for the context. Nevertheless, if the transition to this context is legal, before activating it the system executes the check command (line ❾). An individual context's implementation can use check to signal, for example, whether it is ready to be activated. If FALSE is returned, the currently active context remains unchanged.

The key to realize context-dependent functionality is in the implementation and use of layered functions. In the example of Figure 3—corresponding to the Indoor context—the implementation of the layered command report (line ❶❶) relies on a collection protocol to immediately relay the data to the back end, as an infrastructure is available when indoor. Code relying on the report functionality, for example in a periodic loop, can be written in a generic manner w.r.t. what specific implementation is to be used, as exemplified in Figure 4. The underlying run-time support takes care of dispatching the call to the implementation of the currently active context. The latter is determined at run-time with the activate keyword, also shown in Figure 4.

Ongoing Work and Early Results

We are implementing the tool-chain in support to ConesC. We design translation rules from ConesC to pure nesC, and implement such rules using JavaCC. Meanwhile, we are assessing the effectiveness of ConesC using hand-written examples, looking at how simpler is the implementation of context-dependent functionality against processing and memory overhead. Our analysis targets the TelosB platform and considers nesC as a baseline, because of its widespread adoption in WSNs. For example, a comparison of a simplified version of the Location context configuration and its individual context components shows that the compiled binaries grow by only 2% using ConesC, and yet the code is more decoupled, which fosters simpler maintenance, as well as easier to read and to debug.

References

[1] Dey, A. K., et al. The conference assistant: Combining context-awareness with wearable computing. In *Proc. of the Symp. on Wearable Computers* (1999).

[2] Gay, D., et al. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN PLDI Conf.* (2003).

[3] Hirschfeld, R., et al. Context-oriented programming. *Journal of Object Technology* (2008).

[4] Ko, J., et al. Wireless sensor networks for healthcare. *Proceedings of IEEE* (2010).