Towards Self-adaptive Software for Resource-constrained Cyberphysical Systems

Mikhail Afanasov  
Politecnico di Milano, Italy  
afanasov@elet.polimi.it

Abstract—We present our work towards self-adaptive software for extremely resource-constrained cyberphysical systems (CPSs). The software for the latter generally needs to adapt to unpredictable environmental dynamics under limited resources. Our approach provides design concepts and language support for developing such software. To this end, we bring a notion of context-oriented design and programming down to embedded platforms, while the system overhead is negligible.

I. INTRODUCTION

Cyberphysical systems (CPSs) gather data from and take actions on the real world. The environmental dynamics determines the requirement for CPS software to be adaptable. Moreover, adaptation should occur simultaneously along different dimensions. In wildlife tracking applications [4], for example, sensor nodes are attached to animals to study their movements, social interactions, and health conditions. Nodes are running on batteries, which makes the energy a precious resource. To save it, such devices as GPS and radio should be disabled when not needed. Orthogonally, the node should send the data to the base-station, if it is reachable, or save it locally otherwise.

In the absence of design-time support for self-adaptive software, the adaptation described above would be achieved by ad-hoc code [8], [1], that makes the application cumbersome to design, and difficult to understand, debug, and maintain. A more general approach – context-oriented programming (COP) [3] – addresses these issues by providing a design-time abstractions for adaptations. There are many implementations of COP in high-level languages [2], [6], [7], most of which are not applicable to embedded systems, because of resource limitations.

By borrowing COP concepts and bringing them down to low-level CPS software, we provide a design-time and programming support to enable self-adaptive behavior. As a result, we introduce CONES C – a context-oriented extension of nesC language – in Sec. II, where we also describe the concepts the language was build upon. We discuss our preliminary evaluations in Sec. III, and open problems in Sec. IV.

II. CONCEPTS AND LANGUAGE SUPPORT

We introduce two main notions: i) individual context and ii) context group, which along with a number of secondary notions form a complete design. Context represents an environmental situation the system may find itself in, and defines a behavioral pivot point. The software adapts to the environmental dynamics by activating the corresponding context. Context groups represent collections of contexts, combine the variations of the same functionality, and identify orthogonal aspects and mutual constraints.

CONES C. We render the concepts above in a set of COP-oriented language constructs. Our target language nesC – one of the most popular for programming WSNs – exemplifies the restrictions of CPSs such as limited memory and CPU performance, as well as absence of memory protection. Our context-oriented extension of nesC, called CONES C, overcomes the restrictions and brings COP to CPSs by extending nesC modules and configurations.

The context-oriented design of an example wildlife tracking application is displayed in Fig. 1. Context groups describe possible dynamics corresponding to battery level, base-station availability, health conditions and activity of an animal. In CONES C context group extends a standard nesC configuration and declares layered functions – the core notion of COP [3] meant to implement a behavioral variation – by using the key-word layered, as shown in Fig. 2. Included contexts are declared by using the key-word contexts with optional modifiers is default to define the active context at startup, and optional modifier is error to specify an error context discussed later in this section. Explicit context activation can be initiated anywhere in the code, as:

```
activate BaseStationG.Unreachable;
```

Contexts correspond to the separate environmental situations. For example, two different reachability of the base-station would correspond to different behaviors of the software. Activated context is a pivot point, where the software changes its execution flow according to the implementation of
a layered function – e.g. a behavioral variation. For example, in “Reachable” context, as shown on line 6 in Figure 3, function report() sends the contacts via radio. This function represents continuous activities, as well as logging contacts locally as long as “Unreachable” context is active.

Opposite to continuous activities, there are also one-time operations, such as dumping the logs on the base-station on entering the context “Unreachable”. In CONESC, these operations can be implemented by using predefined events activated() and deactivated(), which are shown on lines 8 and 9. They are automatically signaled when entering or leaving contexts.

The contexts transitions within the group are governed by rules. For example, within the “Base-station” group, the system initiates the transition from “Reachable” to “Unreachable” whenever no beacons from the base-station are received within a specific timeout. Contexts indicated after the keyword transitions on line 12 represent possible outgoing transitions. They are also possibly followed by a key-word iff to state constraints on the transitions:

transitions Diseased iff Resting || NotMoving;

The example is in Fig. 1 in the “Health conditions” group: when activating the “Diseased” context based on body temperature, the software also should check that either “NotMoving” or “Resting” in “Activity group” is currently active. Should the constraints be violated, the “Error” context defined in the corresponding context group is activated. Indeed, a diseased animal is probably not very active. Otherwise, developers might have not correctly caught the contexts evolution.

The required adaptation may affect several context groups. For example, whenever the base-station is “Reachable”, context “NotMoving” should also be activated. The latter saves energy by disabling the GPS-sensor, since the base-station is static and deployed in a known location. To this end, the keyword triggers on line 3 binds the activation across the groups. For example, the context “NotMoving” is activated on entering the “Reachable” context, as specified in Figure 1.

III. Evaluation

We implemented three representative applications in scenarios of wildlife monitoring, smart-home, and adaptive protocol selection. For each of them we compare the nesC implementation against its CONESC counterpart.

We estimated complexity by using such metrics as the number of lines of code (LOC), the number of variables declared, and the number of functions defined [5]. Despite our results show an increase of the number of LOC and declared functions – 17% and 50% correspondingly – the average number of LOC and functions per-module is decreased by 50% and 44%. The modules in CONESC are smaller and more decoupled, thus, developing, debugging, and supporting individual functionality is much easier.

The memory overhead is less than 2.5% for binary size and less than 5% for RAM. The average CPU overhead for layered function calls oscillates from 2 to 5 CPU cycles depending on the application, which is negligible in terms of energy consumption, since the simplest operation in TinyOS – turning on/off LEDs – consumes 8 CPU cycles. The context transitions overhead is bigger – from 16 to 28 CPU cycles – but remains in the same order of magnitude.

IV. Future Work

An open problem of our research are possible conflicts and nontrivial logical errors in the context-oriented model of the application. They can be hardly foreseen at the design-time, but at run-time they could bring the system to an unpredicted behavior. To address this issue, we put a domain-specific model checking in our research agenda.

To our knowledge, there is also a lack of programming environment for self-adaptive applications for CPSs. Our early analysis shows that complexity can be decreased by 9%, if the skeleton of the application based on the diagram akin to the one in Figure 1 is automatically generated by an IDE.

As a part of our further research, we are about to investigate context-oriented programming for other CPS platforms. For example, aerial drones may form a smart mobile network for such task as assisting in archeological excavations or delivery service. We believe, our approach – possibly with minor changes – will fit for this type of systems as well.

V. Biography

Mikhail Afanasov is a second year student at the Department of Electronics, Information and Bioengineering (DEIB) of Politecnico di Milano. He was accepted there in November 2012 after receiving Masters degree in "Applied Physics and Mathematics" at Moscow Institute of Physics and Technology (2012). The expected date of graduation is November 2015. He has a pleasure to conduct his research in tight collaboration with Carlo Ghezzi and Luca Mottola in the area of self-adaptive software for extremely resource-constrained CPSs.

REFERENCES