# Towards Self-Adaptivity in Resource-Constrained Cyberphysical Systems

Mikhail Afanasov*, Luca Mottola*†, and Carlo Ghezzi*

*Politecnico di Milano,†SICS Swedish ICT
afanasov@elet.polimi.it
{luca.mottola,carlo.ghezzi}@polimi.it

**Abstract.** Cyberphysical systems (CPSs) play a great role in gathering data from and possibly taking actions on the real world. Because of unpredictable environment dynamics, software for CPSs needs to be adaptive. So far, however, there is neither dedicated design solutions nor programming support for such software. We address this issue by bringing Context-Oriented Programming (COP) to CPSs. Using COP, programmer use a notion of *layered function* to implement context-dependent variations of the functionality. We provide *language-independent* design concepts, and implementation called ConesC – a context-oriented extension of nesC language. In our tests we observe a significant reduction in complexity of the software, at a price of a maximum 2.5%(4.5%) overhead in program (data) memory.

**Keywords:** resource-constrained, self-adaptivity, context, cyberphysical systems, CPSs

## 1 Introduction

Cyberphysical systems (CPSs) are inherently environment dependent. Because of tight interaction between system and the physical world, CPS software should adapt against the unpredictable environmental dynamics. It is hard to achieve in general [1], and even more whenever a programmer faces a resource limitation.

The challenge arises from taking into account every possible environment situation in the design of CPS software. Multiple combined aspects concurrently determine how the software should adapt its execution. Although the similar problem has been already investigated [1], the solution for extremely resource-constrained platforms is missing. The platform characteristics, such as battery level and memory limitations, make this a challenge.

Current approaches [2] sacrifice self-adaptivity at run-time to mitigate memory limitations. Languages for resource-constrained platforms also prevent dynamic instances creation and binding, which may have helped to implement self-adaptive software. Programmers instead "emulate" adaptivity with hand-written specialized code [2]. The resulted implementations become difficult to maintain and evolve.

## 2 Contribution

We address this issue by presenting context-oriented design concepts and programming model for resource-constrained CPS platforms. To this end we define two key notions: *i) contexts*, and *ii) context groups*. Context represents an individual environmental situation the system may encounter, and in the code it corresponds to behavioral variation specific to a given situation. As the environment mutates, the software adapts by *activating* suitable context. Context groups represent collections of contexts sharing common characteristics; e.g. whenever the *same* functionality needs to be changed to adapt to the given situation. These concepts provide a design-time support to identify common functionality, orthogonal aspects, mutual constraints, and to reflect the situations the software must adapt to, which helps at the implementation phase.

We render these concepts in a set of COP [3] constructs, and exemplify in ConesC[1] – a context-oriented extension of nesC [5]. However, our approach is not tied to it, and can be translated to other systems. nesC reflects the limitations dictated by target platforms, such as low memory (10kbytes of RAM on TMote devices), inability to create run-time instances and absence of memory protection. Thus, existing COP approaches can not be directly ported. At the core of our implementation is a notion of *layered function*, whose behavior depends on the given situation. Crucially, the behavior can be changed *transparently* to the caller, and the explicit managing of the adaptation is not required.

We have shown [4] that ConesC greatly simplifies the resulting code. Yielded software is much more decoupled and easier to evolve as compared to nesC. ConesC also shows a ≈50% reduction in the average number of variable declaration, function definitions, and the total number of combinations of values assumed by variables during every possible execution. The price, however, is negligible: we observe 2.5%(4.5%) as a maximum overhead in program (data) memory.

## 3 Future Work

We are currently extending the assessment of our work to more complex system implementations and to larger sets of performance metrics. As part of our research agenda, we plan to use static verification, e.g., using domain-specific model-checking techniques.

## References

1. Cheng, B. et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. Springer (2009).
2. Mottola, L., Picco, G., P.: Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Comp. Surveys (2011).

[1] `https://code.google.com/p/conesc/`

3. Hirschfeld, R. et al.: Context-oriented programming. Journal of Object Technology (2008).
4. Afanasov, M., Mottola, L., Ghezzi, C.: Context-Oriented Programming for Adaptive Wireless Sensor Network Software. In Proc. of DCOSS (2014).
5. Gay, D. et al.: nesC language: A holistic approach to networked embedded systems. In Proc. of PLDI (2003).