Towards Context-oriented Self-adaptation in Resource-constrained Cyberphysical Systems

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

Luca Mottola
Politecnico di Milano, Italy and
SICS Swedish ICT
luca.mottola@polimi.it

Carlo Ghezzi
Politecnico di Milano, Italy
carlo.ghezzi@polimi.it

***Abstract*—We present a context-oriented approach to design and implement self-adaptive component-based software in resource-constrained Cyberphysical Systems (CPSs). Because of unpredictable environment dynamics, developers must design and implement CPS software to dynamically adapt to widely different situations. Our approach provides design concepts and language support to meet this requirement against severe resource constraints. To this end, we bring a notion of context-oriented design and programming down to platforms that—because of extreme resource constraints—currently leverage fairly undisciplined design techniques and rather rudimentary component-based frameworks. Early results demonstrate that our approach improves the quality of the resulting implementations facilitating testing, maintenance, and evolution at the price of a negligible system overhead.**

I. Introduction

Cyberphysical systems (CPSs) place a computing and communication core in the environment to gather data from, and possibly take actions on the real world. Because of the intimate interactions between the system and the physical world it is immersed in, CPS software is eminently required to self-adapt against the many and unpredictable environment dynamics. This is difficult to achieve in general [3], and even more so whenever developers are to battle against the resource limitations of many existing CPS platforms.

Example. Consider a wireless sensor network application for wildlife monitoring [13]. Sensor nodes are embedded in collars attached to animals, e.g., badgers, to study their social interactions. The nodes are equipped with sensors to track an animal's movement, e.g., using GPS and accelerometers, and to detect its health conditions, e.g., based on body temperature. A low-power short-range radio allows the nodes to discover each other through periodic radio beaconing. A node logs the radio contacts to track an animal's encounters with other animals. The radio is also used to off-load the contact traces when in reach of a fixed base-station.

The nodes run on batteries, making energy a precious resource that developers may need to trade against the system's functionality, depending on the situation. For example, sensor sampling consumes non-negligible energy, especially for devices such as the GPS. Depending on the desired granularity and on the difference between consecutive GPS readings—the latter taken as indication of the pace of movement—developers may tune the GPS sampling frequency accordingly. The contact traces can be sent directly to the base-station whenever in reach, but they need to be stored locally on a node otherwise. When the battery is running low, developers may turn the GPS sensor off to make the node survive until the next encounter with a base-station, not to lose the collected contact traces.

Contribution and road-map. Taking into explicit account every possible enviroment situation in the design of CPS software is a challenge. Crucially, *multiple combined aspects* concurrently determine how the software should adapt its operation, e.g., battery levels and physical locations in our example application. Although the existing literature already investigates similar problems [3], a principled approach at tackling these issues in the design and implementation of CPS software for extremely resource-constrained platforms is largely missing. The platforms' characteristics, such as battery-powered operation and limited memory budgets, make this a challenge.

Existing component-based frameworks for sensor networks [12], for example, employ component-based programming during development, but sacrifice this notion at run-time to mitigate resource limitations. In the nesC language [6], components are in-lined during compilation to enable whole-program analysis, meant to aggressively reduce the size of the program binary to fit the limited memory. This prevents runtime creation of component instances and dynamic component binding, which may help implement self-adaptation functionality by employing different components according to the situation at stake. Programmers often circumvent these limitations by "emulating" these functionality with hand-written specialized code [12]. As a result, implementations become entangled, and are thus difficult to maintain and evolve [14].

We address this issue by presenting context-oriented design concepts and a corresponding programming model expressly conceived for resource-constrained CPS platforms. To this end, we define in Section II a specific notion of *context* and *context group*, useful to conceptually organize the different situations the system may find itself in, and their combinations. This provides support during the design phases. We reflect these notions in a custom *programming model*, described in Section III, which brings concepts of context-oriented programming [8] in existing component-based frameworks for resource-constrained CPSs [6].

Section IV illustrates early results indicating that our approach may result in better structured implementations, where components are increasingly decoupled. We further demonstrate that accounting for changing requirements is likely easier

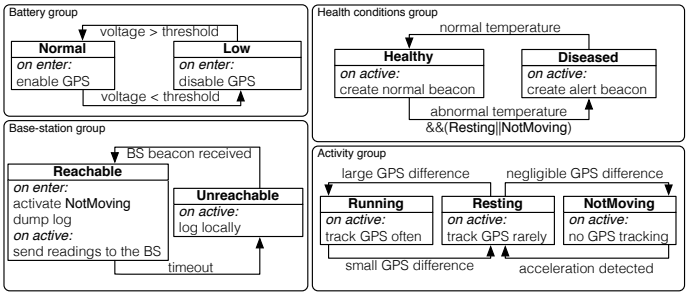

Fig. 1: Wildlife monitoring application design.

in our approach. These results come at a negligible increase in resource consumption, which does not impact the feasibility of our approach on the target platforms. For example, we observe a mere 3% increase in program memory, whereas the energy overhead is negligible.

We conclude the paper by discussing in Section V recurring design and programming patterns that we already observe emerging in our experience, and by briefly surveying related efforts in Section VI. Section VII ends the paper with brief concluding remarks and an agenda for future work.

II. Design

We define two key concepts: *i) contexts*, and *ii) context groups*, along with the notions necessary to weave these concepts into a complete design. Contexts represent the different environmental situations the system may encounter, and correspond in the code to behavioral variations associated to a given situation. As the environment surrounding the system mutates, the software adapts accordingly by *activating* a suitable context. Context groups represent collections of contexts sharing common characteristics; e.g., whenever the *same* functionality must adapt to changes in the surrounding environment.

As an example, Figure 1 depicts the context-oriented design of the wildlife monitoring application described earlier. Context groups are defined to describe behavioral variations corresponding to battery levels, base-station reachability, as well as an animal's health conditions and activity. The contexts within a group define the single behavioral variations. For example, the software behaves differently depending on whether the base-station (BS) is reachable, as shown within the "Base-station" group.

The contexts in a group are tied with explicit *transitions*, labeled with the conditions triggering the context change. For example, within the "Base-station" group, the system transitions from context "Reachable" to "Unreachable" whenever no beacons are received from the base-station within a specific timeout. This entails a node is out of the base-station radio range and the software must adapt accordingly; for example, by locally storing the contact logs instead of sending them over the radio.

Within the single contexts, it is useful to distinguish between *one-time operations* executed at the time of entering or exiting a context, and *continuous activities* that occur as long as a context is active. For example, on entering context "Reachable", the software dumps on the base-station the contact logs locally accumulated while the base-station was unreachable. Similarly, when in the latter situation, the software must log the contacts locally as long as the "Unreachable" context persists, as shown in Figure 1.

The required adaptation may span multiple context groups. To this end, developers can bind context activations across groups. An example is in the "Reachable" context within the "Base-station" group: on entering, context "NotMoving" should also consequently activate. As base-stations are typically deployed at known locations and their radio range is very limited, continuing to sample the GPS sensor is likely a waste of energy, as a node's locations can be approximated with the base-station one. Hence we can avoid waiting for the next GPS sample before deciding to stop sampling until an acceleration is detected.

Developers may also bind context transitions to the activation of other contexts, which is useful to check at run-time for design errors. An example is when activating the "Diseased" context in the "Health conditions" group. Besides an abnormal body temperature that may reveal a disease, the adaptation process must check that either "Resting" or "NotMoving" in the "Animal activity" group is currently active. Indeed, if an animal is diseased, it is probably not very active. Should that not be the case, developers might have not correctly captured how contexts evolve, potentially indicating a design error.

The concepts we define provide design-time support to reason on the different situations the software must adapt to, and to identify common functionality, orthogonal aspects, and mutual constraints. This helps separate concerns during the implementation phase, as we illustrate next.

III. Programming Support

We render the design concepts above in a set of context-oriented programming (COP) [8] constructs feasible within existing component-based frameworks for resource-constrained CPS platforms [12]. We exemplify our approach based on nesC [6]. However, our approach is not tied to it, and may be readily translated to other similar systems [12].

Target language. nesC is a component-based event-driven programming framework for sensor networks, derived from C. Applications are built by interconnecting *components* that interact by providing or using *interfaces*. An interface lists one or more functions, tagged as *commands* or *events*. Commands are used to execute actions, while events are used to collect the results asynchronously. Interfaces in nesC are bidirectional: data flows both ways between components connected through the same interface. Component *configurations* specify the wirings among components. Configurations are component themselves, so they can provide interfaces and be wired to other components.

nesC exemplifies the limitations dictated by the target platforms, and hence the reasons why existing COP approaches cannot be directly ported. Besides the inability to create run-time instances of components and to reconfigure component wirings we already mentioned, for example, the use of dynamically-allocated memory is also discouraged: the Microcontroller Units (MCUs) provide no memory protection, so bugs in memory handling may have disastrous effects.

```
1 context group BaseStationG {
❷  layered command void report(contact_t contact);
3 }
4 implementation {
❺  contexts Reachable,
❻           Unreachable is default,
❼           ErrorC is error;
8  // Standard nesC component wirings...
9 }
```

Fig. 2: Context group in CONESC.

```
1 context Reachable {
❷  transitions Unreachable;
❸  triggers NotMoving;
4  uses interface Radio;
5 }
6 implementation {
❼  layered command void report(contact_t contact){
8   call Radio.send(contact);
9 }
❿  event void activated(){// Dump logs on base-station }
⓫  event void deactivated(){ // Radio clean-up }
12 }
```

Fig. 3: CONESC context.

CONESC. We design a context-oriented extension to nesC, called CONESC, that incorporates the design concepts described in Section II.

At the core of CONESC is a notion of *layered function* [8]. These are functions whose behavior depends on the currently active context, and are hence the primary means to implement the behavioral variations necessary for self-adaptation. Crucially, the behavior of layered functions may change *transparently* to the caller, which is then relieved from explicitly managing the adaptation required by a given situation. We embed layered functions in nesC interfaces as specialized commands.

A *context group* in CONESC extends nesC configurations by specifying the contexts included in the group and the layered functions that such contexts provide. Figure 2 shows a snippet of CONESC code to implement the "Base Station" group in Figure 1. In this example, the **`report()`** command on line ❷—used to report a contact with another animal to the end-user—changes the behavior depending on whether the base-station is **`Reachable`** or **`Unreachable`**. The latter are the contexts included in this group, specified on line ❺ after the keyword **`contexts`** with optional modifiers: **`is default`** (line ❻) specifies the active context at start-up, and **`is error`** (line ❼) indicates an error context. The latter is automatically activated should there be violations to constraints defined over context transitions; for example, the fact that "Resting" or "NotMoving" must be active when transitioning from "Healthy" to "Diseased", as shown in Figure 1.

The notion of *context* extends a nesC component by providing the context-dependent implementations of layered functions. Figure 3 shows the CONESC implementation of the "Reachable" context. The keyword **`transitions`** on line ❷ specifies the allowed outgoing transitions, whereas the keyword **`triggers`** on line ❸ binds context activations across groups. In this case, entering the "Unreachable" context consequently activates "NotMoving", as indicated in Figure 1. The specific implementation of the layered function is shown on line ❼ with the **`layered`** keyword. The implementation of other commands or events is as in standard nesC. Particularly, the predefined events **`activated()`** and **`deactivated()`**,

```
1 module BaseStationContextManager {
2  uses context group BaseStationG;
3 }
4 implementation {
5  event msg_t Beacon.receive(msg_t msg) {
❻   activate BaseStationG.Reachable;
7   call BSReset.stop();
8   call BSReset.startOneShot(TIMEOUT);
9 }
10  event void BSReset.fired() {
⓫   activate BaseStationG.Unreachable;
12 }}
```

Fig. 4: Base-station context controller.

```
1 module User {
❷  uses context group BaseStationG;
3 }
4 implementation {
5  event void Timer.fired() {
❻   call BaseStationG.report(msg);
7 }
❽  event void BaseStationG.contextChanged(context_t con) {
❾   if(con == BaseStationG.Reachable) // DO SOMETHING...
10 }}
```

Fig. 5: Caller module.

shown on lines ❿ and ⓫, are automatically signalled when entering or exiting the context, allowing the implementation of one-time operations and the setup/shutdown of continuous activities in a context.

The contexts indicated after the **`transitions`** keyword inside single contexts are possibly followed by the **`iff`** keyword to state constraints on the transitions, as in

```
transitions Diseased iff Resting || NotMoving;
```

used in the definition of the "Healthy" context to encode the constraints in Figure 1. If such a transition is attempted at runtime, but the constraints are violated, the error context defined in the corresponding context group is activated.

Programmers can, anywhere in the code, trigger explicit transitions between contexts in a group. This is as simple as using the **`activate`** keyword followed by a full context name, as shown in Figure 4. The full context name is determined by concatenating the name of the enclosing context group and of the single context. For example, the *Reachable* context is activated on line ❻ of Figure 4 as soon as a beacon from the base station is received. Should the timeout expire with no more beacons received, context *Unreachable* is activated on line ⓫. Either context change results in a different context-dependent implementation of **`report`** to be activated. These implementations are found within the single contexts.

Modules using layered functions perform function calls transparently w.r.t. the available contexts and, most importantly, independently of what context is active at a given moment. Fig. 5 shows one such example for function **`report`**. Following the indication that context group *BaseStationG* is used, as specified on line ❷, the call to the layered function **`report`** does not refer to the single contexts. The net advantage is that the use of context-dependent functionality is fully decoupled w.r.t. context detection and activation. The two may be implemented even in different modules. Such feature helps separate orthogonal concerns and hence facilitates testing, maintenance, and evolution of the software, as we discuss next.

Finally, should programmers of caller modules need to find

Type	Description
Content (tightest)	One module relies on the internal working of another. Changing one module requires changes in the other as well.
Common	Two or more modules share some global state, e.g., a variable.
External	Two or more modules share a common data format.
Control	One module controls the flow of another, e.g., passing information that determine how to execute.
Stamp	Two or more modules share a common data format, but each of them uses a different part with no overlapping.
Data	Two or more modules share data through a typed interface, e.g., a function call.
Message (loosest)	Two or more modules share data through an untyped inter- face, e.g., via message passing.

Fig. 6: Coupling types.

out about the run-time evolution of contexts, a predefined event **`contextChanged`** is fired corresponding to every context change, as on line ❽ of Figure 5. Within the event handler, programmers can access constant values that our translator automatically generates to find out what context was activated and to react accordingly, as shown on line ❾.

Translation. We develop a dedicated translator to convert CONESC code to plain nesC. To do so, our translator performs two passes through the input code. First, it reads the main nesC **`Makefile`** to determine the main configuration component and to recursively scan the component tree. Based on the information gained during the first pass, including the list of every context and context groups defined in the code, it parses every input file to convert the CONESC code to plain nesC and to generate a set of support functionality managing context transitions at run-time. The resulting sources are then compiled using the standard nesC toolchain.

Our translator is implemented using JavaCC [9]. Two aspects are worth noticing. First, the generated code is completely hardware-independent. Therefore, hardware compatibility is the same as the original nesC toolchain, allowing us to support a wide range of platforms and not to modify our translator due to hardware idiosyncrasies. Second, the whole translation process is only seemingly straightforward. Rendering the logic embedded within the CONESC abstractions does require a fairly sophisticated processing. To give an intuition, we measured the size of the CONESC implementations of a set of representative applications against the size of the nesC implementations output by our translator. On average, we observe three times as much lines of code in the automatically-generated nesC code.

IV. PRELIMINARY EVALUATION

Using the translator, we compare an implementation of the wildlife monitoring application against a functionally-equivalent nesC implementation that would arguably result from current practice [12], [14], [13]. Nonetheless, the following considerations more generally apply to a larger set of applications we are experimenting with, including a smart-home controller and an adaptive sensor network protocol stack, whose detailed discussion we omit for brevity. Our comparison addresses three key dimensions, as described next.

1. Coupling and cohesion. According to Stevens et al. [19], seven types of coupling between software components exist, as summarized in Figure 6. It is generally known that the tightest is coupling, the more difficult is debugging, maintaining, and extending the implementations.

Implementation	Content	Common	External	Control	Stamp	Data	Message
nesC-based	yes	yes	yes	yes	–	yes	–
ConesC-based	–	–	yes	–	–	yes	–

Fig. 7: Coupling comparison.

Our analysis reveals that the CONESC implementation is significantly more decoupled than its nesC counterpart, as qualitatively indicated in Figure 7. Based on current practice, the adaptation functionality would be implemented as a monolithic nesC component, using global state variables to dispatch function calls to different components depending on the situation. With our approach, most types of coupling among components are removed, as context transitions implicitly cater for the appropriate dispatching of function calls. Consequently, programmers are relieved from explicitly managing global state transitions, which facilitates maintenance and testing.

The individual CONESC components also appear more cohesive and simpler: on average, using our approach, a component is almost half the lines of code than using plain nesC and defines half the number of global variables compared to nesC. We further observe a 75% reduction in the number of commands/events declared in CONESC components compared to a plain nesC implementation.

2. Evolving the software. Besides dealing with the run-time adaptation required to cope with environmental dynamics, CPS software also needs to evolve in response to changing requirements [14]. Generally, the better an implementation is modularized, the easier are the modifications, since the changes are limited to isolated portions of the system.

Say, for example, developers of the wildlife monitoring applications need to track the spreading of a disease. To this end, they modify the application design by adding a new context "Carrier" in the "Health conditions" group to mark an animal who was in contact with a diseased one, but shows no increase in body temperature yet.

Using our approach, this is as simple as changing 5 lines of code in the CONESC implementation, besides the functionality needed in a new "Carrier" context. Based on current practice, the same modification would crucially require adding at least two global states. This adds the burden of explicitly managing their transitions and is further detrimental to ease of testing and maintainability.

3. System overhead. The advantages above come at the price of a negligible system overhead. We assess this aspect on the widespread TMote Sky sensor node [15], featuring a 16-bit MSP430 MCU with 10 Kb RAM and 48Kb for program memory. In particular, we measure the MCU overhead for context transitions and calls to layered functions, as well as memory overhead when using CONESC as compared to nesC. To measure the MCU overhead we use the MSPSim MSP430 emulator [5], while we estimate the memory overhead using the tools in the nesC and GNU-C toolchains.

Aboard the TMote Sky, calls to layered functions in our example application add only 3 CPU cycles over standard function calls, whereas context transitions involve at most 20 CPU cycles. To put these figures in prospective, turning an LED on takes 8 CPU cycles on a TMote Sky. The increased

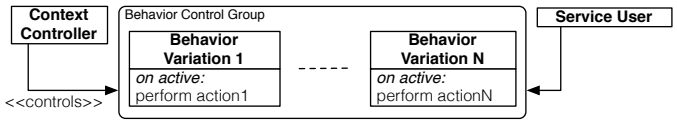

Fig. 8: Behavioral control pattern.

latency and consumed energy due to the additional CPU cycles are hence immaterial. As for memory consumption, we measure a mere 3% increase in both RAM and program memory. These figures demonstrate that the price to pay for obtaining significant advantages in the quality of the implementations is modest, and hence our approach promises to be feasible even on extremely resource-constrained platforms.

V. Emerging Patterns

Despite the limited experience we hitherto gathered using CONESC, we already observe quite distinctive design and programming patterns, representing solutions to commonly occurring problems. As discussed next, our approach allows developers to deal with diverse requirements using only a handful of concepts.

Behavior control. Programmers often employ a single context group to specify different *behaviors* for the same high-level functionality. One such example is the "Base-station" group in Figure 1, which includes two different behaviors for the functionality to report contact logs to the users. The functionality itself is exported by one or more layered functions defined on the group. The chosen behavior is then determined by activating a single context within the group.

We found similar designs in other applications as well. In the adaptive protocol stack, for example, the packet relay functionality also matches a similar design. Depending on a node's mobility, the chosen behavior is picked out of a pool of available protocols, whose functionality are encapsulated in single contexts. These are in turn included in a single context group, which exports a layered function used by the application to transparently accesses whatever protocol is in operation at a given time.

Figure 8 shows an abstract view of such commonly recurring pattern. In addition to the context group exporting the adaptive functionality and the single contexts therein, programmers also define an additional "context controller" component, which activates the single contexts within the group depending on the situation. Figure 4 shows a CONESC example for the wildlife monitoring application. Similar designs apply to the smart-home controller and the adaptive protocol stack as well.

Content provider. Different from the behavior control pattern, which provides non-trivial context-dependent processing, we observe cases where context-dependent *data* is offered to other functionality with little to no processing involved. In the wildlife monitoring application, for example, the "Health conditions" group in Figure 1 provides differently formatted beacons to the radio driver for broadcast transmissions. Layered functions are, in this case, defined for the group merely to retrieve the context-dependent data.

In this case as well, we notice the same pattern in other applications. In the smart-home controller, for example, a context group is defined to manage the user preferences depending on

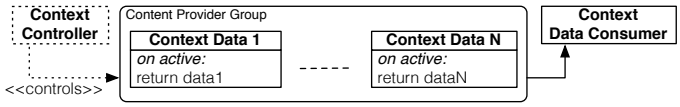

Fig. 9: Content provider pattern.

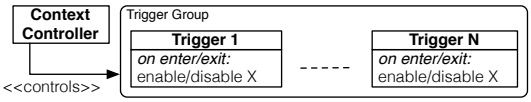

Fig. 10: Trigger pattern.

day vs. night. These data are simply retrieved differently from a data storage by two different contexts modeling day or night situations. Whatever user preference is to be considered at a given point is then handed over to the control loop in charge of setting the functioning of the climate systems.

As shwn in Figure 9, this pattern's structure differs from that of behavior control in that the role of the "controller" component is often fairly trivial (and hence omitted in the picture). In the smart-home controller, for example, the controller component is simply based on the time of the day. On the other hand, the component consuming the context-dependent data plays a key role. Indeed, while functionality structured according to behavior control can be considered stand-alone, the context provider needs to be tailored to the data consumer.

Trigger. We also recognize designs where single contexts are used only to trigger specific operations when entering/exiting, but no significant context-dependent functionality or data is offered as the context remains active. One example in the wildlife monitoring application is the "Battery" group in Figure 1. The included contexts are used to enable/disable the GPS sensor depending on battery levels, but no other functionality is provided to other components. In this case, layered functions are often not defined, in that the predefined **`activated`** and **`deactivated`** events within the single contexts suffice.

In the smart-home controller, for example, we notice a similar pattern in the context group regulating light dimming. Depending on perceived light levels in a room, either context "Too bright" or "Too dark" is activated, and lights are tuned accordingly when entering either context. This processing is entirely implemented within the corresponding **`activated`** event handlers.

In more general terms, a "context controller" component is present in this case as well to drive the context transitions in the group, as shown in Figure 10. However, unlike the other patterns, there is no other significant component that either uses context-dependent functionality or consumes context-dependent data. The functionality is mostly self-contained.

VI. Related Work

Efforts close to ours particularly address the design and implementation of self-adaptive embedded system software, context-oriented programming, and system-level adaptiveness in CPSs. We briefly survey paradigmatic examples.

Works in self-adaptive embedded system software spans phases from requirement engineering to verification [3]. Co-design approaches also exist where the hardware/software boundaries blur for greater flexibility [4]. Unlike in our work, these solutions focus on a few environmental dimensions, each

requiring ad-hoc self-adaptive functionality. In our target applications, complexity arises especially from the combinations that multiple environmental dimensions concurrently generate. Nevertheless, most of these solutions would be hardly applicable in resource-constrained CPSs, due to run-time overhead.

Villegas [20] designed dedicated software support for situation-aware software systems. Despite sharing some high-level challenges with our work, Villegas relies on the end-user as a controller of context management, whereas we focus on autonomous systems. Moreover, we directly deal with physical sensors to acquire context information, whereas these are abstracted in software-based sensor devices in Villega's work.

Fleurey et al. [5] present a model-driven approach for creating adaptive firmwares. They model the application as a single state machine and define behavioral variations based on predicates defined over the application state. When such predicates are found true, the system accordingly adapt the state machine transitions. Code is automatically generated from these specifications. Compared to this effort, we do not target completely automatic code generation, but provide dedicated programming constructs. This offers greater freedom in encoding the conditions to trigger context changes and enables finer-grained optimizations, which may be mandatory given the resource limitations. Moreover, we seek to integrate our approach in existing component-based CPS frameworks, leveraging the existing code base.

COP [8] made its way into several high-level languages [1], [7], [10], [16], [18]. These are generally unfeasible on our target platforms. We borrow a few of these concepts—for example, our layered functions in context groups are akin to the concept of layer-in-class [16]—and adapt them to the limitations of component-based frameworks for resource-constrained CPSs. In this area, the embedded devices are rather typically seen as application-agnostic providers of raw sensor data [18]. Differently, we bring COP down to the component-based CPS software, enabling self-adaptive functionality right on the devices that interact with the environment.

Aside from COP, Meta- and Aspect-oriented Programming (AOP) offer programming support to implement adaptive functionality [17]. The former requires self-modification of the binary, which is often unfeasible in resource-constrained CPSs. Similar requirements hold for AOP [11], which is often applied to large and complex software projects. Arguably, applying AOP to the relative simple processing running aboard the CPS devices, even if possible, would be quite overkill. Moreover, general agreement is that COP offers the best support for modularization [17]. This feature is fundamental to embed programming support for self-adaptation within existing component-based frameworks.

Specific cases of run-time adaptation are seen at system level in the CPS literature. For example, Zimmerling et al. [21] focus on run-time reconfiguration of MAC protocol parameters depending on environmental conditions. Routing protocols expressly designed with self-adaptive functionality also exist [2]. Such efforts essentially address a complementary problem. While we aim to provide design-time and programming support to implement self-adaptive software in this domain, these works focus on the actual problem-specific adaptation logic.

VII. Conclusion and Future Work

We presented a solution to provide design-time and programming support for self-adaptive software in resource-constrained component-based CPS software. In this domain, the lack of a principled design approach and the rudimentary programming environments result in entangled implementations. We thus conceived dedicated design concepts and COP extensions to existing component-based frameworks. Preliminary results indicate that our approach may yield implementations that are easier to test, maintain, and evolve. The run-time overhead to pay is, nonetheless, negligible.

We are currently extending the assesment of our work to more complex system implementations and to larger sets of performance metrics, while investigating ways to automatically generate CONESC skeletons based on graphical representations of contexts and context groups similar to Figure 1. As part of our research agenda, we plan to use the same notation as input to perform static verification, e.g., using domain-specific model-checking techniques.

Acknowledgments. This work was partly supported by project the ERC Advanced Grant EU-227977 SMScom and by the Swedish Foundation for Strategic Research (SSF).

References

[1] J. E. Bardram. The Java context awareness framework (JCAF) – A service infrastructure and programming framework for context-aware applications. In *Proc. of PERVASIVE*, 2005.
[2] T. Bourdenas et al. Self-adaptive routing in multi-hop sensor networks. In *Proc. of CNSM*, 2011.
[3] B. Cheng et al. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Springer, 2009.
[4] J. Diguet et al. Closed-loop-based self-adaptive hardware/software-embedded systems: Design methodology and smart cam case study. *ACM Trans. on Embedded Computing Systems (TECS)*, 2011.
[5] F. Fleurey et al. A model-driven approach to develop adaptive firmwares. In *Proc. of the 6th SEAMS*, 2011.
[6] D. Gay et al. nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI*, 2003.
[7] C. Ghezzi et al. Programming language support to context-aware adaptation: A case-study with Erlang. In *Proc. of ICSE SEAMS*, 2010.
[8] R. Hirschfeld et al. Context-oriented programming. *Journal of Object Technology*, 2008.
[9] JavaCC - The Java Compiler Compiler. javacc.java.net.
[10] T. Kamina et al. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proc. of AOSD*, 2011.
[11] G. Kiczales et al. Aspect-oriented programming. In *ECOOP*, 1997.
[12] L. Mottola and G. P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comp. Surveys*, 2011.
[13] B. Pasztor et al. Selective reprogramming of mobile sensor networks through social community detection. In *Proc. of EWSN*, 2010.
[14] G. P. Picco. Software engineering and wireless sensor networks: Happy marriage or consensual divorce? In *Proc. of FSE/SDP FOSER*, 2010.
[15] J. Polastre et al. Telos: Enabling ultra low-power wireless research. In *Proc. of IPSN*, 2005.
[16] G. Salvaneschi et al. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 2012.
[17] G. Salvaneschi et al. An analysis of language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.*, 2013.
[18] S. Sehic et al. COPAL-ML: a macro language for rapid development of context-aware applications in wireless sensor networks. In *Proc. of SESENA*, 2011.
[19] W. Stevens et al. Classics in software engineering: Structured Design. Yourdon Press, 1979.
[20] N. Villegas. *Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems*. PhD thesis, University of Victoria, Canada, 2013.
[21] M. Zimmerling et al. pTunes: Runtime parameter adaptation for low-power MAC protocols. In *Proc. of IPSN*, 2012.