Context-Oriented Programming for Adaptive Wireless Sensor Network Software

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 programming abstractions for implementing *adaptive* Wireless Sensor Network (WSN) software. The need for adaptability arises in WSNs because of unpredictable environment dynamics, changing requirements, and resource scarcity. However, after about a decade of research in WSN programming, developers are still left with no dedicated support. To address this issue, we bring concepts from Context-Oriented Programming (COP) down to WSN devices. Contexts model the situations that WSN software needs to adapt to. Using COP, programmers use a notion of *layered function* to implement context-dependent behavioral variations of WSN code. To this end, we provide *language-independent* design concepts to organize the context-dependent WSN operating modes, decoupling the abstractions from their concrete implementation in a programming language. Our own implementation, called CONESC, extends nesC with COP constructs. Based on three representative applications, we show that CONESC greatly simplifies the resulting code and yields increasingly decoupled implementations compared to nesC. For example, by model-checking every function in either implementations, we show a ≈50% reduction in the number of program states that programmers need to deal with, indicating easier debugging. In our tests, this comes at the price of a maximum 2.5% (4.5%) overhead in program (data) memory.**

I. INTRODUCTION

Programmers design and implement Wireless Sensor Networks (WSN) software to enable interactions in the real world at unprecedented granularity. As such, WSN software is continuously confronted with a range of largely unpredictable environment dynamics and changing requirements, besides resource constraints. This demands WSN software to *adapt* to a range of different situations. Notwithstanding the advances in WSN programming [17], however, programmers are sorely missing dedicated support to realize adaptive WSN software.

Example application. Consider a wildlife tracking application [19]. Sensor nodes are embedded in collars attached to animals 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. Small solar panels harvest energy to prolong a node's lifetime. A low-power short-range radio allows the nodes to discover each other based on periodic radio beaconing. A node logs the radio contacts to track an animal's encounters with other animals. The radio is also used to offload the contact traces when in reach of a fixed base-station.

The nodes run on batteries, making energy a precious resource that programmers need to trade against the system's functionality, depending on the situation. For example, sensor sampling consumes non-negligible energy for the GPS. Depending on the desired granularity and on the difference between consecutive GPS readings—taken as indication of the pace of movement—programmers need 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 otherwise. When the battery is running low, developers may turn the GPS sensor off to make sure the node survives until the next encounter with a base-station, not to lose the collected contact traces.

```
1  module ReportLogs {
2    uses interface Collection;
3    uses interface DataStore;
4  }implementation {
5   int base_station_reachable = 0;
6   event msg_t Beacon.receive(msg_t msg) {
7    if (!acceleromenter_detects_activity())
8      return;
9    if (call Battery.energy() <= THRESHOLD)
10     return;
11   base_station_reachable = 1;
12   call GPS.stop()
13   call BaseStationReset.stop();
14   call BaseStationReset.startOneShot(TIMEOUT);}
15  event void BaseStationReset.fired() {
16   base_station_reachable = 0;}
17  event void ReportPeriod.fired() {
18   switch (base_station_reachable){
19    case 0:
20     call DataStore.deposit(msg);
21    case 1:
22     call Collection.send(msg);}}}
```

Fig. 1: Example nesC implementation of adaptive functionality: *several orthogonal functionality become entangled and need to share global data.*

Problem. Taking into explicit account every possible situation in the design and implementation of WSN software is a challenge. Crucially, *multiple combined dimensions* concurrently determine how the software should adapt its operation, e.g., battery levels and physical locations in our example application. Using available approaches, this typically results in entangled implementations that are difficult to debug, to maintain, and to evolve. As the number of dimensions affecting the execution (and their combinations) grows, the implementations quickly turn into "spaghetti code" [6].

Fig. 1 shows an intuitive, yet greatly simplified example, using nesC [8]. The code implements the behavior needed in wildlife tracking to send contact logs to the base-station whenever reachable, or to store them locally otherwise. Several

orthogonal concerns become intertwined and dependent on each other. For example, determining what operating mode to apply—implemented in line ❻ to ⓰—rests within the same module as the actual adaptive processing—implemented in line ⓱ to ⓬. Indeed, the two codes need to share global state, in this case, the **`base_station_reachable`** flag. Managing such global state rests entirely on the programmers' shoulders. Moreover, the checks to apply before changing operating mode, such as those on line ❼ to ❿, appear interleaved with the change of mode itself. Finally, the specific implementation of adaptive functionality—using either **`DataStore`** or **`Collection`**—is entirely visible from the caller module, further coupling the two.

In such a situation, debugging, maintaining, and evolving the implementations is going to be difficult. Modifying the code in one place would likely require changes in several others. Alternative nesC implementations of the functionality in Fig. 1 are of course possible to partly ameliorate the problem. However, qualitative evidence gathered by looking at publicly available implementations, e.g., within the TinyOS codebase [24], indicate that similar implementation patterns are indeed very common.

Contribution and road-map. We aim to redress this state of affairs by enabling a notion of Context-Oriented Programming (COP) [11] in WSN software. COP fosters a strict separation of concerns in implementing adaptive software. This is achieved through two key notions: *i)* the different situations where the software needs to operate are mapped to different *contexts*, and *ii)* the different context-dependent behaviors are encapsulated in *layered functions*, that is, functions whose behavior changes—transparently to the caller—depending on context.

COP already proved effective in creating context-aware mainstream software, such as user interfaces [14] and text editors [13], based on COP extensions of popular high-level languages [20]. At present, however, COP remains a far cry from being applicable to WSNs. The resource constraints that limit the functionality attainable with existing WSN programming languages, for example, the inability to create run-time instances of components, prevents applying COP in WSN programming as is.

To address this issue, we borrow concepts from COP and design context-oriented programming abstractions for WSN software. To this end, Section II illustrates design concepts conceived to remain independent of a specific programming language. In doing so, our goal is to decouple the abstractions from their concrete realization in a language, thus facilitating their application to multiple WSN languages. One such realization is CONESC, our own COP extension to nesC. We choose nesC as the target language in that, besides its widespread adoption and stable toolchain, it fosters a node-centric view [17]. We argue that in most WSN applications, adaptation decisions are most often local to individual nodes. In illustrating CONESC, Section III demonstrates how we render the processing in Fig. 1 nicely decoupled in different modules, and hence easier to debug and to evolve.

We implement a dedicated translator, described in Section IV, that converts CONESC code to pure nesC. Based on three representative applications, the results we illustrate in Section V indicate that CONESC implementations are increasingly decoupled and distinctively simpler. For example, the analysis we perform with a model-checking tool to measure the number of states that programmers need to deal with shows a ≈50% reduction in favor of CONESC, indicating that the latter implementations are likely easier to debug and to maintain. Crucially, these advantages come at a modest price: the MCU overhead when performing calls to layered functions is negligible, while we measure a a maximum 2.5% (4.5%) overhead in program (data) memory.

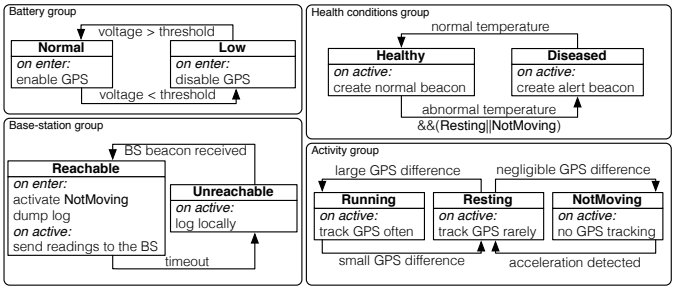


Fig. 2: Wildlife tracking diagram.

We conclude the paper by surveying related efforts in Section VI, and with brief concluding remarks in Section VII.

II. DESIGN CONCEPTS

We illustrate language-independent design concepts, providing a foundation to apply the COP model to different concrete languages, as we illustrate next. Throughout this section and the next, we refer to the wildlife tracking application described earlier as a running example.

We define two key concepts: *i)* individual contexts, and *ii)* context groups. Contexts represent the different environmental situations the system may encounter, and correspond to behavioral variations associated to a given situation. As the environment surrounding the system or the requirements mutate, the software adapts accordingly by activating given contexts. Context groups represent collections of contexts sharing common characteristics; for example, whenever the required adaptive behavior is determined by the same environmental quantity.

Fig. 2 exemplifies how programmers use these concepts in the design of the wildlife tracking application. Context groups are defined to describe behavioral variations corresponding to different battery levels and whether a node is within the communication range of a base-station, as well as an animal's health conditions and activity.

The contexts within a group define the individual behavioral variations depending on the situation. For example, the function to report sensor readings and contact traces must behave differently depending on whether the base-station is reachable. If so, the data may be relayed immediately to the base-station using the radio. To this end, the programmer activates context *Reachable* within the *Base-station* group. Otherwise, the programmer activates context *Unreachable*, as the software must log the data locally; for example, on flash.

The contexts in a group are tied with transitions that express the conditions triggering the context change. For example, within the *Base-station* group, the system transitions

```
1 context group BaseStationG {
❷  layered command void report(msg_t msg);
3 }implementation {
❹  contexts Reachable,
❺           Unreachable is default,
❻           MyErrorC is error;
7  components Routing, Logging;
8  Reachable.Collection -> Routing;
9  Unreachable.DataStore -> Logging;}
```

Fig. 3: Context group in CONESC.

from context *Reachable* to *Unreachable* whenever no base-station beacons are received within a timeout. This entails a node is out of the base-station communication range and the software must adapt accordingly, that is, by locally storing the contact logs instead of sending them over the radio.

The behavioral variations must not necessarily implement a complete functionality on their own, but they may just serve other functionality; for example, by providing context-dependent data. The group *Health conditions* is one such example. By using a body temperature sensor, the system detects whether the animal is *Diseased* or *Healthy*. The corresponding behavioral variations implement two ways to build the radio beacon used as a "proximity" sensor for detecting contacts between animals. If the animal is *Diseased*, additional information is added to the beacon for understanding how the disease spreads. Either type of beacon is then handed over to the radio stack for transmission.

The concepts we outlined suffice to organize the environment-dependent functionality in a large class of WSN applications, as we further argue in Section V. On the other hand, unlike the vast majority of WSN programming approaches [17], these concepts remain largely decoupled from a concrete language implementation. Although the following section describes a nesC-based implementation, our design can be straightforwardly embedded within other WSN languages. For example, within functional languages such as Regiment [18] or Flask [16], one would simply enable behavioral variations of programmer-defined functions through a proper syntax, together with dedicated keywords for context transitions.

III. CONESC

We illustrate how we render the concepts in Section II within CONESC: our own context-oriented extension to nesC. We describe a notion of context module and configuration in Section III-A, and discuss in Section III-B how programmers use these constructs to specify an application's adaptive behavior. Section III-C describes how CONESC programmers deal with context transitions and their relations.

A. Context Group and Individual Contexts

Context groups in CONESC extend nesC configurations. Programmers use context groups to declare layered functions and the contexts providing the corresponding behavioral variations. Fig. 3 shows an example for the *Base-station* group. A layered **report** function is declared on line ❷ by using the keyword **layered**. The contexts providing the necessary behavioral variations are specified following the keyword **contexts** on line ❹. In this case, programmers define two such contexts, depending on base-station reachability. The

```
1 context Reachable {
2  uses interface Collection;
3  uses context group BatteryG;
4 }implementation {
❺  event void activated(){
6   call GPS.stop();}
❼  event void deactivated(){//...}
❽  command bool check(){
9   return call BatteryG.getContext() == BatteryG.Normal;}
10 layered command void report(msg_t msg){
❶❶  call Collection.send(msg);}}
```

Fig. 4: *Reachable* context.

```
1 context Unreachable {
❷  transitions Reachable iff ActivityG.Running;
3  uses interface DataStore;
4 }implementation {
❺  event void activated(){//...}
❻  event void deactivated(){//...}
❼  command bool check(){//...}
8  layered command void report(msg_t msg){
❾   call DataStore.deposit(msg);}}
```

Fig. 5: *Unreachable* context.

```
1 module BaseStationContextManager {
2  uses context group BaseStationG;
3 }implementation {
4  event msg_t Beacon.receive(msg_t msg) {
❺   activate BaseStationG.Reachable;
6   call BSReset.stop();
7   call BSReset.startOneShot(TIMEOUT);}
8  event void BSReset.fired() {
❾   activate BaseStationG.Unreachable;}}
```

Fig. 6: Base-station context manager.

is default modifier, shown on line ❺, indicates what context is active at start-up. The next **is error** modifier on line ❻ declares context **MyErrorC** as an *error* context, which programmers may optionally use to handle errors during the execution, as we discuss in Section III-C. If an error context is not declared, it is generated automatically.

The individual contexts in CONESC extend the standard nesC modules by providing context-dependent implementations of layered function declared in context groups. Only one context at a time can be *active* in a group to provide an implementation for the given layered functions. For example, Fig. 4 and 5 show CONESC snippets for the *Reachable* and *Unreachable* contexts of Fig. 3. They provide different implementations for **report** depending on the situation. If the base-station is *Reachable*, and thus the corresponding context is active, the code transmits the message to the base-station, as in line ❶❶ of Fig 4. Differently, the code deposits a message in local memory as in line ❾ of Fig. 5.

Programmers can specify operations upon activating a context, such as initializing variables or enabling/disabling hardware modules. For example, on entering the *Reachable* context, programmers may decide to disable the GPS sensor, as location information can be inferred from the (static) base-station. Programmers specify this functionality within the body of a predefined **activated** event, as in line ❺ of Fig. 4. Similarly, programmers may specify clean-up operations within **deactivated** events, as in line ❼ of Fig. 4. Providing an implementation for these events, however, is not mandatory.

B. Execution

Fig. 6 shows a sample snippet of code to detect and to activate the proper context in the base-station example. Pro-

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

Fig. 7: Caller module.

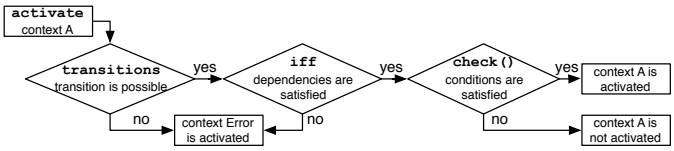


Fig. 8: Context activation rules.

grammers 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. In this fragment of code, the *Reachable* context is activated on line ❺ 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 ❾. Either context change results in a different context-dependent implementation of **report** to be activated.

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. 7 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 individual 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.

Should programmers of caller modules need to find out about context changes, a predefined event **contextChanged** is fired corresponding to every context change, as in line ❻ in Fig. 7. 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 ❼.

C. Transition Rules

In general, programmers need to take significant care of context transitions, in that the latter may drastically change an application's behavior. To better support programmers in doing so, every context transition in CONESC entails several checking stages, as shown in Fig. 8. A successful check allows the transition to continue, while the failure leads either to the canceling of the transition or to activation of the *Error* context.

The first check in Fig. 8 looks at feasible transitions. In the context diagram of Fig. 2, within the *Activity* group, it is only possible to transition from *NotMoving* to *Resting*. Feasible transitions are specified within the individual contexts using the keyword **transitions** as in line ❷ of Fig. 9. An attempt to initiate a transition from a context to one that is not explicitly listed in the former leads to the activation of the *Error* context. Indeed, such occurrences typically represent a significant design or implementation flaw requiring special

```
 1 context NotMoving {
❷  transitions Resting;
 3 }implementation {//...}
```

Fig. 9: *NotMoving* context.

```
 1 context Low {
❷  triggers BaseStationG.Unreachable;
 3 }implementation {//...}
```

Fig. 10: *Low* context.

handling at run-time, which programmers implement within the *Error* context.

There may also exist relations across context groups. For example, within the *Base-station* group, a transition from *Unreachable* to *Reachable* is likely only meaningful if context *Running* within the *Activity* group is active, indicating the animal was actually moving when the node gained base-station connectivity. These inter-group relations are covered in our design by context dependencies, declared as shown on line ❷ in Fig. 5. Within the **transitions** clause, the keyword **iff** is optionally employed to indicate the full name of another context whose activation is a requisite to perform the given transition. The second check in Fig. 8 verifies this rule, again leading to the *Error* context in case of violations, giving programmers a chance to handle the situation.

The last check in Fig. 8 considers violations to "soft" requirements that do not necessarily indicate a design or implementation flaw. For example, before activating the *Reachable* context, programmers may want to check that sufficient energy is available to invest in bulk data transfers to the base-station. Should this not be the case, they may defer the activation of the *Reachable* context until the solar panels gather sufficient energy. To implement such processing, CONESC programmers specify the proper conditions in the body of a predefined **check** command, as shown in line ❽ of Fig. 4. If **check** returns false, the initiated context transition does not occur, and the system remains in the previous context.

Dually, programmers may need to proactively initiate context transitions as the result of other contexts being activated. The scenario is symmetric to the previous one: if the base-station is *Reachable*, but a context transition is initiated to context *Low* in the *Battery* group of Fig. 2, the available energy is running low and it is probably better to refrain from radio communications. This makes sure the node does not completely turn off before the solar panels re-gain energy. Our design allows programmers to express this processing by using the **triggers** keyword, as shown on line ❷ in Fig. 10. The **triggers** keyword points to a context that is to be activated as the result of the enclosing context being activated. The same checks shown in Fig. 8 apply to this type of transitions.

IV. TRANSLATOR

We develop a translator to convert CONESC code to plain nesC. Our translator performs two passes through the input code. First, it reads the main **Makefile** 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, the translator parses every input file to convert the CONESC code to plain nesC and to generate

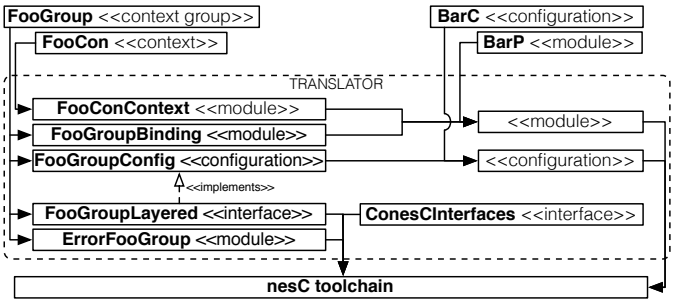


Fig. 11: CONESC translation to nesC code for a generic **FooGroup** context group and an individual context **FooCon**.

a set of support functionality. The resulting sources are then compiled using the standard nesC toolchain.

Fig. 11 details the operations during the second pass. Generally, the input to the translator includes four types of components: context groups and contexts, as well as nesC configurations and modules where CONESC constructs appear. In Fig. 11, context groups and contexts are represented by a sample **FooGroup** context group and an individual **FooCon** context, whereas nesC configurations (modules) with CONESC constructs are represented as **BarC** (**BarP**).

Based on every context group, we generate a custom nesC module, such as **FooGroupBinding** in Fig. 11, that implements the dynamic binding of layered functions to the active context. This module is part of a configuration, such as **FooGroupConfig** in Fig. 11, also automatically generated. This configuration implements a nesC interface our translator produces, such as **FooGroupLayered** in Fig. 11, that exports the layered functions defined in the group. Optionally, an error context is also generated in plain nesC, as indicated by **ErrorFooGroup** in this case, if the programmer does not provide one. Each individual context is translated to a corresponding nesC module with the proper interfaces to be wired within the aforementioned configuration, as in the case of **FooConContext** for the **FooGroupConfig** in Fig. 11.

At this stage, context and context groups disappeared, yet CONESC constructs, such as **activate**, may still appear within the source code. Our translator converts these constructs to functionally-equivalent nesC code both in the nesC files generated out of context groups and individual contexts, and in the plain nesC files that possibly includes them, such as **BarC** and **BarP** in Fig. 11. The resulting sources are then wired to generic interfaces that define the predefined commands and events in CONESC, such as **contextChanged** for context groups, as in Fig. 7, and **activated/deactivated** for individual contexts, as in Fig. 4 and 5. The result is plain nesC code that can be given as input to the nesC toolchain.

Our translator is implemented using JavaCC [12]. Three aspects are worth noticing. First, the generated code is still human-readable, and a programer can modify it to implement fine-grained optimizations. Second, the code is completely hardware-independent. Therefore, hardware compatibility is the same as the original nesC toolchain, allowing us to support a wide range of WSN 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 the application we use for evaluation, described next, 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.

V. EVALUATION

We implement three representative applications, as described in Section V-A, using either CONESC or nesC. The implementations are functionally equivalent. Based on these, we evaluate our approach along four dimensions. Section V-B analyzes the severity of different *coupling types* in our implementations. Tighter forms of coupling are generally detrimental to code maintenance and evolution [15]. Section V-C reports code metrics assessing the *complexity* of the implementations, which often impacts a system's reliability and ease of debugging [15]. The efforts required for evolving the software are measured in Sec. V-D based on illustrative case studies. Finally, Section V-E quantifies the performance overhead when using CONESC in terms of MCU and memory penalty.

A. Applications

To demonstrate the generality of our design, we implement a smart-home controller and an adaptive protocol stack in addition to the wildlife tracking application.

The smart-home controller, whose design is shown in Fig. 12, relies on context information to regulate temperature and lighting conditions in a room, as well as to deal with emergency situations. The former functionality are driven by user-provided preferences that depend on the current context. The preferences are managed within the *Preferences* group, whose contexts provide different operating parameters depending on day/night and working days vs. weekend conditions. The context transitions within the *Light* and *Temperature* groups are driven by thresholds found in such parameter set, compared against current temperature and light readings. When transitioning between these contexts, the node operates actuators to control the HVAC and lighting systems. The controller exploits image, fire, and smoke sensors to detect housebreaking and fire situations. It may notify the user about the incident and possibly relay data to a controller in a different room, depending on the situation.

The adaptive protocol stack, whose detailed context diagram we omit for brevity, implements dynamic protocol switching in situations where a node may alternative periods of significant mobility to periods of static operation. The node roams within a network of static nodes running CTP [10]. As long as the node remains static, it joins the existing routing tree by running an instance of CTP. As soon as the on-board accelerometer detects a significant movement, it switches to a route-less gossip protocol, which allows the node to relay data to the static infrastructure opportunistically [7]. In addition, the node may switch between three parameter sets for CTP, depending on context information that determine whether lifetime, bandwidth, or latency is to be favored.

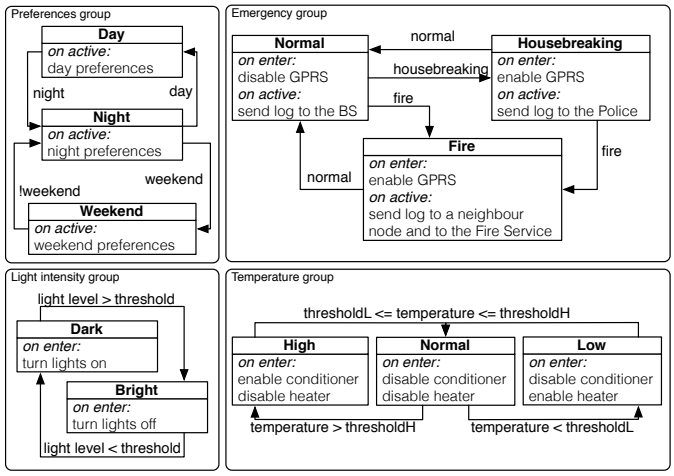


Fig. 12: Smart-home controller context diagram.

TABLE I: Coupling types.

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.

B. Coupling

According to Stevens et al. [22], seven types of coupling between software modules exist, as summarized in Table I. It is generally known that the tightest is coupling, the more difficult is debugging, maintaining, and extending the implementations. We investigate the types of coupling we can observe in CONESC and nesC implementations.

Results. Table II illustrates the results of our analysis. Generally, the ConesC implementations are significantly more decoupled compared to their nesC counterparts. CONESC avoids *Content* coupling in that different behavioral variations are encapsulated in different contexts. NesC programmers, on the other hand, cannot dynamically bind command calls or event signals to different modules, which forces them to expose internal module information that make one module's operation depending on that of several others'. For the same reason, nesC programmers are forced to use global state to switch between different functionality depending on the situation. This creates *Common* coupling that is not found in CONESC, in that the necessary functionality is automatically generated by our translator. Finally, CONESC spares *Control* coupling as well. This is a result of allowing dynamic binding across modules driven by the context transitions. Such functionality needs to be hand-coded in nesC.

However, both CONESC and nesC force *Data* and *External* couplings. This is unavoidable, in that both rely on typed interfaces and different modules in both implementations must necessarily agree on a common data format.

TABLE II: Coupling comparison: CONESC *implementations save most types of coupling that are unavoidable in nesC.*

Application	Content	Common	External	Control	Stamp	Data	Message
Wildlife tracking – nesC	yes	yes	yes	yes	–	yes	–
Wildlife tracking – ConesC	–	–	yes	–	–	yes	–
Smart-home – nesC	yes	yes	yes	yes	–	yes	–
Smart-home – ConesC	–	–	yes	–	–	yes	–
Adaptive stack – nesC	yes	yes	yes	yes	–	yes	–
Adaptive stack – ConesC	–	–	yes	–	–	yes	–

TABLE III: Complexity comparison: CONESC *yields simpler implementations that are easier to debug and to reason about.*

	Average per-module		
Application	Variable declarations	Functions	Per-function states (avg)
Wildlife tracking – nesC	6	8	12567.3
Wildlife tracking – ConesC	3	2	6231.2
Smart-home controller – nesC	2	2	18654.2
Smart-home controller – ConesC	0,8	1,9	5678.3
Adaptive stack – nesC	2,5	3,25	9830.3
Adaptive stack – ConesC	0,4	1,6	3451.8

C. Complexity

We estimate the complexity of the implementations by measuring the number of variable declarations and the number of functions in every module. These are generally considered as intuitive indicators of a program's complexity [15]. It is also observed that complexity is a function of the number of states in which the program can find itself [15]. A state here is any possible assignment of values to the program variables. Thus, the number of states must be computed by looking at the different combinations of values assumed by variables during every possible execution.

To carry out the latter analysis, we use SATABS [3], a model-checking tool for C programs. SATABS performs offline verification of C programs against user-provided assertions. To do so, it searches through the relevant program executions to check whether the assertion always holds. At the end of the process, SATABS returns the number of states it explores in the program. Using a specific configuration, it is possible to force SATABS to explore *all* program executions. If the procedure terminates, SATABS returns the total number of distinct states in a program. We use SATABS on a per-function basis, implementing empty stubs to replace code that we cannot process with SATABS, e.g., hardware drivers.

Results. Table III illustrates our results. On a per-module basis, CONESC shows significant reductions in both the number of declared variables and defined functions. This comes from the ability to dynamically bind a function call to the required context-dependent implementation transparently to the caller. In nesC, on the other hand, this requires defining global variables to check what behavior needs to be triggered depending on the situation. As a result of this, the number of per-function states programmers must manage also drastically decreases, making the implementations simpler to understand.

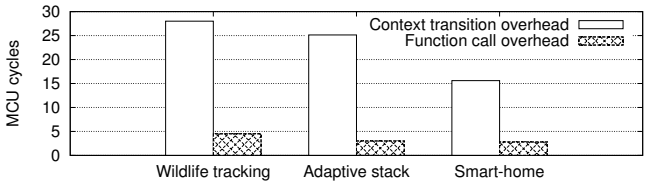
As debatable as it may be for measuring the effectiveness of a programming abstraction [17], we also measured the number of lines of code in both nesC and CONESC implementations: the two are roughly comparable. More interestingly, however, as already discussed, we also measured the size of the code generated by our translator, described in Section IV. Besides giving an intuition of the complexity involved in the translation process, this figure also indicates the "expressive power" of the abstraction, that is, the amount of processing that CONESC programmers can succinctly express using the language constructs we design. As already mentioned, it turns out that the output of our translator is roughly *three times* the size of the input code, demonstrating that our abstractions do capture a significant portion of processing in a few simple concepts.

D. Software Evolution

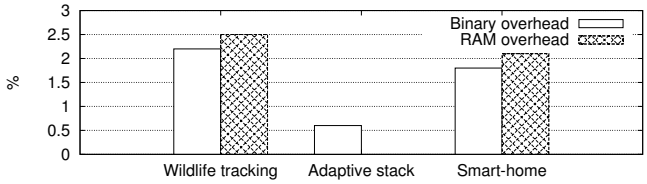
WSN software needs to constantly evolve due to changes in requirements. Generally, the better an implementation is modularized, the easier are the modifications, since the changes will affect an isolated portion of the system [15]. For each application we consider, we estimate the effort to modify the CONESC implementation compared to the nesC counterpart. We study three types of modification: removing a context, adding a new context, and adding a new context group.

To estimate the effort for removing a context, we rework the scenario of the adaptive protocol stack. Say developers want to remove of one of the CTP parameter sets after testing, since those parameters performed ineffectively. To study the addition of a new context, we extend the wildlife tracking application to the case where it becomes necessary to monitor the spread of a disease. To do this, developers add a new context *Carrier* to the *Health conditions* group to create a beacon for an animal who was in contact with a diseased one, but shows no symptoms yet. To study the addition of an context group, in the smart-home controller scenario we consider a case where developers need to monitor a device's state depending on periodic run-time checks. If a potential failure is discovered, the controller should change its behavior. To this end, programmers add an entire context group *Status* with two contexts *Normal* and *Failure*.

Results. Our analysis shows that removing a context in the CONESC implementation of the adaptive protocol stack only requires modifying 3 lines of code, besides deleting the context itself. To remove unnecessary functionality in nesC, developers must modify several lines of code scattered throughout the main module. To add a context in the wildlife tracking application, using CONESC it is necessary to modify 5 lines of code, besides providing the implementation for the new context. Implementing the same extension in nesC requires, among other code modifications, adding two new global states, further complicating the control flow. Adding a new context group in the smart-home controller requires modifications in about 40 lines of code using CONESC, besides providing the implementation for the new contexts. Using nesC, about the same amount of code changes are required, yet these include adding two global states, again rendering the implementation necessarily more entangled. Finally, worth noticing is that the effort to apply context-unrelated changes in CONESC is the same as in nesC.



(a) MCU overhead.



(b) Memory overhead.

Fig. 13: MCU and memory overhead: *the resource usage penalty for using* CONESC *is almost negligible.*

E. MCU and memory overhead

The advantages brought to programmers come at the cost of additional system overhead. To assess this, 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 tools in the nesC and GNU-C toolchains. As the executions are determinstic, the run-time experiments constantly yield the same measures.

Results. Fig. 13 shows the results. The average MCU overhead for a layered function call ranges from 2 to 5 MCU cycles, depending on the application. Such figures are negligible in terms of energy consumption, since the simplest operation in TinyOS, that is, turning on/off an LED, already consumes 8 MCU cycles. The overhead of context transitions is slightly larger, but in the same order of magnitude. This arises from the activation rules, described in Fig. 8. Additional MCU cycles are needed to check if the transition is possible, then to check the dependencies, and finally to execute the body of `check()`.

Most importantly, the memory overhead is also negligible, measuring a worst-case 2.5% penalty for the size of the program binary and a worst-case 4.5% penalty for RAM usage. The complexity of the application largely dictates the corresponding memory overhead. For example, the wildlife monitoring application, being the most complex in terms of contexts, context changes, and data processing, shows the highest overhead. The overhead for the adaptive protocol stack, on the other hand, is negligible in that a nesC programmer would essentially leverage a similar set of variables compared to those that our CONESC translator automatically generates.

VI. RELATED WORK

Efforts related to ours are roughly divided in two categories. On one hand, as the need for adaptivity in WSNs was immediately recognized because of the intimate environment interactions, several system-level solutions exist to provide adaptive behaviors at different levels in the stack. Our work is complementary to these efforts: rather than devising problem-specific adaptation mechanisms, we present design concepts and programming constructs to facilitate the implementation

of such mechanisms. On the other hand, programming support for adaptation, including the application of COP, is more extensively studied for more traditional computing platforms, missing, however, a dedicated port of concepts and abstractions to resource-constrained devices. In the following, we briefly survey the literature based on examples closer to our work.

Adaptation in WSNs. Solutions in this category often target run-time adaptation of MAC and routing protocols. For example, Zimmerling et al. [25] focus on an adaptation of MAC protocol parameters depending on link qualities, topology dynamics, and traffic loads. Based on an mathematical formulation of the problem at hand, the base-station computes optimized MAC parameters to satisfy user-provided performance goals. Another example is that of Bourdenas et al. [2], who design a routing protocol with dedicated adaptive functionality. Using a custom forecasting approach, the system can predict the conditions of the network and anticipate the changes required in the routing protocol behavior. Our work is intended to serve the needs of those needing to implement such adaptive functionality, easing their implementation chore.

Closer to our goals are the works on self-organizing WSN architectures. For example, Subramanian and Katz [23], define a component model to build adaptive WSN architectures. Their work, however, is again intended for specific adaptation needs, that are, those arising in static WSNs deployed for large-scale sensing tasks. For example, the work is not applicable in mobile scenarios akin to a wildlife tracking application. Diguet et al. [4] blur the boundaries between software and hardware to gain additional flexibility in providing adaptive functionality. Their design, however, leads to application-specific implementations. Our work aims to be more general than these efforts, as we demonstrated by applying CONESC to diverse application scenarios, as discussed in Section V.

Programming support for adaptation. Some works explicitly provide programming support for adaptation *outside* the WSN, essentially regarding the latter as an application-agnostic source of raw data. For example, Sehic et al. [21] design a Java-based framework for context-aware applications using input data from a WSN. Differently, we bring a notion of context down to the resource-constrained devices, allowing to implement context-aware behaviors right on the WSN device.

A natural way to handle adaptivity at the programming level is to embed some notion of COP within an existing language, similar to what we do. Indeed, several high-level languages already feature COP extensions [1], [9], [13], [20], [21]. Such approaches, however, are far from being applicable in WSNs, due to specific application requirements and resource limitations. For example, the multiple dimensions of adaptive behavior germane to WSN applications are rarely considered in existing works. In CONESC, we borrow concepts from COP and adapt them to the typical requirements arising in WSN applications and to the limitations of related platforms.

VII. CONCLUSION

We presented programming abstractions for implementing adaptive WSN software. By borrowing from COP, we conceived language-independent design concepts mirrored in a concrete language implementation—CONESC—that extends nesC with COP constructs. Our dedicated translator converts CONESC code to plain nesC code, then handed over to the standard nesC toolchain. Based on three representative applications, we observed that CONESC greatly simplifies developing adaptive WSN software. For example, we found that, along with increased decoupling of software components, we gain a ≈50% reduction in the number of per-function states that programmers need to deal with, and applications can be evolved with reduced efforts compared to their nesC counterparts. The price for gaining such advantages is, however, negligible: we observed an overhead of 2.5% (4.5%) in program (data) memory, whereas the MCU overhead is negligible. The CONESC toolchain is available at code.google.com/p/conesc.

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] E. Clarke et al. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. of TACAS*, 2005.
[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] J. Eriksson et al. COOJA/MSPSim: Interoperability testing for wireless sensor networks. In *Proc. of Simutools*, 2009.
[6] N. Finne et al. Improving sensornet performance by separating system configuration from system logic. In *Proc. of EWSN*, 2010.
[7] H. Fotouhi et al. Smart-HOP: A reliable handoff mechanism for mobile wireless sensor networks. In *Proc. of EWSN*, 2012.
[8] D. Gay et al. nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI*, 2003.
[9] C. Ghezzi et al. Programming language support to context-aware adaptation: A case-study with Erlang. In *Proc. of ICSE SEAMS*, 2010.
[10] O. Gnawali et al. Collection Tree Protocol. In *Proc. of SENSYS*, 2009.
[11] R. Hirschfeld et al. Context-oriented programming. *Journal of Object Technology*, 2008.
[12] JavaCC - The Java Compiler Compiler. javacc.java.net.
[13] T. Kamina et al. EventCJ: A context-oriented programming language with declarative event-based context transition. In *Proc. of AOSD*, 2011.
[14] R. Keays et al. Context-oriented programming. In *Proc. of MobiDe*, 2003.
[15] P. Koopman. *Better Embedded System Software*. Carnagie Mellon Press, 2010.
[16] G. Mainland et al. Flask: Staged functional programming for sensor networks. In *Proc. of ICFP*, 2008.
[17] L. Mottola and G.P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comp. Surveys*, 2011.
[18] R. Newton et al. The Regiment macroprogramming system. In *Proc. of IPSN*, 2007.
[19] B. Pasztor et al. Selective reprogramming of mobile sensor networks through social community detection. In *Proc. of EWSN*, 2010.
[20] G. Salvaneschi et al. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 2012.
[21] 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.
[22] W. Stevens et al. Classics in software engineering. chapter Structured Design. 1979.
[23] L. Subramanian and R.H. Katz. An architecture for building self-configurable systems. In *Proc. of MobiHOC*, 2000.
[24] TinyOS 2.1.2. www.tinyos.net.
[25] M. Zimmerling et al. pTunes: Runtime parameter adaptation for low-power MAC protocols. In *Proc. of IPSN*, 2012.