AdaptC: Programming Adaptation Policies for WSN Applications

Shashank Gaur
CISTER Research Centre, ISEP,
Polytechnic Institute of Porto
Porto, Portugal
sgaur@isep.ipp.pt

Luis Almeida
CISTER Research Centre, FEUP,
University of Porto
Porto, Portugal
lda@fe.up.pt

Eduardo Tovar
CISTER Research Centre, ISEP,
Polytechnic Institute of Porto
Porto, Portugal
emt@isep.ipp.pt

ABSTRACT

Evolution in both hardware and software technologies has enabled Wireless Sensor Networks(WSNs) to target a multiplicity of domains. Programming for such advanced WSNs remains a challenging process for users, especially as the WSN may need to make changes as per outcomes from different scenarios during execution. Usually, various adaptation policies are written while programming such applications to enable changes. However it is difficult for the programmer to anticipate changes for new scenarios. It also becomes difficult to reuse these adaptation policies. In this paper, we propose AdaptC, an abstraction for such adaptation policies that facilitates re-usability and expansion across various WSNs. We also present concepts for the design and implementation of AdaptC. We evaluate the abstraction for multiple use cases and compare it against existing work.

CCS CONCEPTS

• **Computer systems organization** → **Sensor networks**; **Sensors and actuators**; • **Software and its engineering** → **Embedded middleware**; **Abstraction, modeling and modularity**; **Application specific development environments**; • **Networks** → *Sensor networks*; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools*; • **Hardware** → Sensors and actuators; Wireless integrated network sensors;

KEYWORDS

wireless sensor network, cyber-physical systems, macroprogramming, applications, adaptation, internet of things, context-awareness

ACM Reference Format:
Shashank Gaur, Luis Almeida, and Eduardo Tovar. 2019. AdaptC: Programming Adaptation Policies for WSN Applications. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3297280.3297304

1 INTRODUCTION

Wireless Sensor Networks can collect data from the environment using different hardware nodes and select different actions based on that data as defined by distributed software applications. Over time, both the hardware nodes and the programming capabilities have become more sophisticated. Due to better System on Chip (SoC) solutions, today WSN nodes are capable of collecting complex sensing data and also process such data before communicating to the network itself. On the software front, advanced simulators and programming support enable the user to express diverse goals instead of worrying about low-level features for writing the applications. Recently, WSNs with such advanced capabilities are considered part of the ecosystem of the Internet of Things (IoT), fostering the development of wide Cyber-Physical Systems (CPS). These advanced programming capabilities also open doors to application programmers from a wide range of industries such as agriculture [2, 20], healthcare [19], etc. In many CPS domains, such as smart manufacturing, efficient use of WSN can directly influence efficiency and productivity. Hence, it is important to provide the programmers with effective tools to focus on high-level concepts, by avoiding the idiosyncrasies of traditional WSNs.

However, many features of programming WSNs are still limited to experts only. Programmers have to go through elaborate software methodologies such as understanding of advanced simulators or operating systems built to program WSNs. For example, Contiki [13] has become one of the most popular operating systems for programming WSNs, which provides extensive libraries, communication support, and multi-threading. Just to learn about the features of Contiki and how to effectively use them requires a significant effort from the programmer. In addition, programmers need to obtain knowledge about the hardware nodes as well, which is further complicated by the ever-growing list of manufacturers of different WSN nodes.

Writing WSN applications is further complicated when there is a need to adapt to changes in operational scenarios. Such scenarios can include changes in physical parameters, such as location, or system changes, such as availability of nodes, energy levels, etc. For this purpose, programmers need to anticipate all possible future scenarios and thus program actions to adapt accordingly.

In the past, many efforts were developed to simplify programming for the WSN, e.g., via macroprogramming [22]. Other efforts attempted at reducing the complexities in designing systems where the hardware and software are closely integrated [11]. However, constraints on both hardware and software capabilities prevented applying these solutions to WSN nodes. Another effort aimed at providing a complete software solution for situational-aware systems [25]. But there is a need to exchange data between different applications and there is a lot of responsibility on end-user to manage the context. Conversely, there is not much effort in providing abstractions for programming the sensor nodes so that WSNs can

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04.
https://doi.org/10.1145/3297280.3297304

detect changes by themselves and directly adapt to those changes without delving into low-level details.

One of the most promising recent work is T-Res [4]. T-Res provides a programming abstraction, specially tailored for the IoT paradigm by providing data processing tasks for nodes. T-Res uses CoAP operations for this, which allow configuring applications on a node and the interaction among multiple nodes. The main drawback of T-Res is no support for adaptations and re-usability, according to changes in WSN or IoT nodes. Another similar work is PyFUNS [8], which also enables reprogramming in WSNs. PyFUNS leverages IoT based protocols such as Constrained Application Protocol (CoAP) and IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN). PyFUNS can be used as a complementary tool, but it does not provide support for adaptation in the application written by the programmer.

Context-Oriented Programming (COP) is also another popular effort to facilitate programmability of WSNs [18]. In embedded systems research, there have been many efforts to evolve existing programming solutions towards COP [6, 17, 21, 23, 24]. Recently there have been efforts to add extra features to COP [5].

There is also some recent work focusing on providing support for context-awareness [3]. Note that changing from one context to another the application needs to change accordingly, too. But there is still some opportunity to improve support to programmers and end-users. These efforts still lack an easy way of writing adaptations policies. It lacks an adequate abstraction that could foster re-usability.

In this paper, we study how a programmer would establish such adaptation policies and we show the difficulties in achieving so. In addition, we demonstrate the need for writing such adaptation policies considering a couple use cases from different domains. With the help of these use cases, we also examine essential features to build a generic model. Based on that model, we propose a new programming abstraction, named AdaptC, for writing continuous and complex adaptation policies that can be reused and extended.

The main contributions of this paper are the following:

- Design features for setting continuous and complex adaptation policies.
- A novel programming abstraction for adaptations that can be used with languages such as nesC.
- Examples of the ability to reuse and extend the adaptation policies written using the proposed abstraction.

This paper is composed as follows. Section II discusses design features for adaptation with help of different use cases. Section III showcases the programming of adaptation policies using a pseudo code and analyzes the design features. Section IV proposes the abstraction for writing the adaptation policies and showcases its advantages. We conclude the paper by briefly discussing the work and its conclusions in Section V. Section VI finishes the paper with remarks on future directions.

2 DESIGN FEATURES FOR ADAPTATIONS

In this section, we identify the design features which are critical in defining the desired adaptation policies. These design features can also express the complexities in writing these applications from a programmer's perspective. To express the design features we propose the generic model in 1.

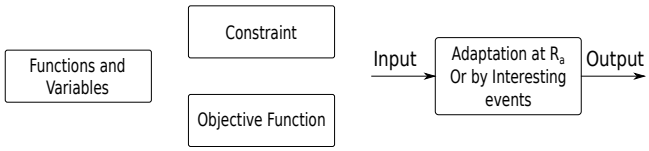

Figure 1: Generic Model for Design Features

This generic model is composed of several elements. *Functions and Variables* represent all the relationships between different variables of the application that are relevant to the desired adaptation. These relationships can either exist from design or can be obtained with the progress of the application. *Constraint* describes the restrictions for the adaptations. *Objective Function* represents the desired outcome for the user. There must be some relationship between *Constraint* and *Objective Function*, but it is optional to have a direct relationship.

The component on the right expresses the dynamics of the adaptation policy. Adaptations can either be carried out periodically at a *Rate of Adaptation* or triggered by other events taking place in the system. These other events are called *Interesting Events* in the generic model. The user can tag different elements of the program as Interesting Event and the adaptation will happen every time that element of the program is triggered. These Interesting Event can also impact on the *Functions and Variables* component. All these changes are included in the next adaptation cycle.

To better explain the design features, we consider two different use cases and try to apply the proposed model to these use cases.

2.1 Usecase 1 : GPS

Consider a use case in which a wildlife animal is tracked by a GPS-enabled sensor node attached to the animal itself. Base stations are deployed across the forest to collect the GPS data from the sensor node on the animal. The primary objective is to sample the GPS at a fixed rate and store the movements of the animal locally at the node itself. Whenever a base station is encountered all the recorded data can be transferred and then delivered to the user through the base station. After this, the user also wants to assure that the sensor node maintains sufficient battery level to encounter the next base station. To achieve that, the GPS polling rate must be adapted according to various situations such as the speed of the animal, other applications running on the same node, the amount of energy left or any encounter with other animals. The programmer can not anticipate the speed of the animal and program a GPS sensor polling rate for all the possible situations. Also, some new events which may affect such adaptation may be added later on. Hence, there is a need to support continuous adaptation capabilities while programming the goals.

This adaptation policy can be expressed, using pseudo-code, in the following way:

For Speed(S) and BatteryLevel(B):
Poll GPS with Rate(R)

so that BatteryLevel(B) is always greater than Threshold after Time(t)

The variables in this use case are Speed of the animal(S), Battery Level(B), Sampling Rate(R) and Time(t). There is a direct relationship between how frequently the GPS is sampled(R) and the level of the battery(B). This can be either pre-defined by the user based on earlier studies [7, 9] or studied by the system over time. Here, we can express it as B = f(R). This relationship helps to learn how much B will be affected by changes in R. There is a constraint on how much B can be affected. B should always be above a threshold battery levels(B_T), i.e. $B \geq B_T$. While the system must follow the constraint, the user may desire that the polling rate must be maximized in order to get as much as fresh GPS data. That would require a complex and continuous adaptation for this application and maximizing R would be the objective function for this adaptation.

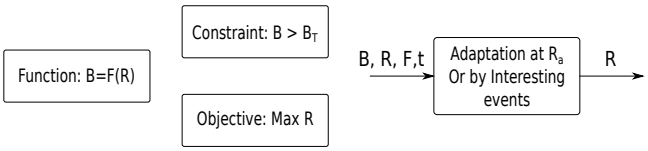

Figure 2: Design Features for GPS Use Case

The user can decide a fixed sampling rate at which this objective function should be solved, which will be the rate of adaptation(R_a). Also, some interesting events within the program can also trigger the adaptation. For example in this use case, these events can be *An encounter of another base station, Change in the number of other applications, Not able to get GPS position, Encounter of another wildlife animal,* etc. Figure 2 shows the design features for this use case.

2.2 Usecase 2 : HVAC

We consider a system for building automation such as Heating, Ventilation, and Air-Conditioning (HVAC) [10, 14]. HVAC systems are a common utility in modern infrastructures, such as offices, shopping malls or industrial buildings. Typical HVAC systems have different sensor nodes to monitor physical conditions such as temperature, pressure, humidity in the various parts of the building. According to user requirements for those physical parameters, certain actuation is performed on various cooling or heating devices or any other actuators. In some HVAC systems, there is only one user requirement such as maintaining a certain temperature. But in the case of more complex buildings, the user might want to minimize the cost of operations while satisfying multiple user requirements. During the operation, the HVAC system should adapt to different events to minimize the cost and provide sufficient performance. Also, adaptation is expected in other cases such as offices where different occupants might have different requirements and their behavior might affect the HVAC performance. Hence, it becomes difficult for a programmer to write an application which can keep up with all the above-mentioned factors and achieve its main goal.

Again using pseudo-code, we can express this adaptation policy as follows:

For Volume(V) and Time(t):
Provide Power(P) to maintain Temperature(T_F)
so that Cost of operation(C) is minimized over a time period(t)

The variables in this use case are Volume of the space(V), Power Consumption(P), Temperature of the space(T_F), Operational Cost(C) and Time(t). There is a direct relationship between the temperature being maintained(T_F), the volume of the space(V) and the power consumption(P). There is also a direct relationship between the power consumption(P) over time(t) and the operational cost(C). These can be either pre-defined by the user based on statistics or studied by the system over time. Here, we can express these as T_F = f(V,P) and C = g(P,t). These relationships show which variables can affect operational cost and how changing them can help in achieving the objective of the user, i.e. minimizing operational cost(min(C)). The constraint is on temperature since the system must always maintain a suitable temperature as well. If the required temperature is T_S, then there can be some tolerance(δ) to minimize operational cost. Hence, that would be the constraint for this use case, i.e. $T_S - \delta \geq T_F \geq T_S + \delta$. This would require a complex and continuous adaptation and minimizing cost would be the objective function for this application.

Concerning the dynamics of the adaptation, the user can decide a fixed sampling rate at which this objective function should be solved, which will the rate of adaptation(R_a). Also, some interesting events within the program can also trigger the adaptation. For example, in this use case, these events can be *Change in the occupied volume in the space, Body temperature of the occupants, Location of the occupants, Interaction with outside environment when doors open or close,* etc. Figure 3 shows the design features for this use case.

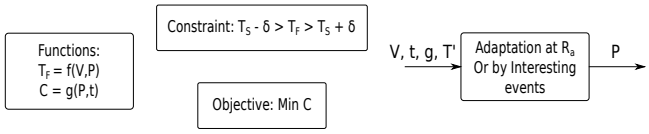

Figure 3: Design Features for HVAC Use Case

3 PROGRAMMING ADAPTATION POLICIES

Let us focus on how a programmer would write adaptation policies for WSNs, with current state-of-practice approaches, using the C language. In particular, let's examine the use case 1 (GPS) presented in the previous section, for which we provide the code in the Listing 1. The relationship between the BatteryLevel and the GPS polling rate is described by the Function in line 4, which takes the Rate as input. In this function, we assume *alpha* is the factor by which the GPS sampling rate affects the battery level. In order to maintain the normal operations and the adaptation simultaneously, the programmer must create multiple threads. In line 7, `sensing_thread` is the function that describes the normal behavior of the application using all the parameters provided by the user. Another thread is required for adaptation, which is called `adaptation_thread` in line 19. This function checks the BatteryLevel using the earlier functions for current Rate. Then it gets a new polling rate under the constraint of keeping BatteryLevel above a threshold defined by the user. It uses haversine function to calculate the speed from the GPS coordinates in line 24.

One of the design features is that the adaptation must either occur at a fixed rate or triggered by some interesting events across the application. Hence, a third thread has to be created to trigger adaptation at a fixed rate by using timers, which is defined in line 13. In addition, the `adaptation_thread` must be called back inside all the functions that may generate interesting events.

```
1  int Solve(Speed, BatteryLevelcurrent ) {
2      // Calculate R using the solution for optimization
3      Return R; }
4  int Function(Rate){
5      BatteryLevel = alpha*Rate;
6          return BatteryLevel; }
7  void sensing_thread () {
8          while() {
9                  timer = clock();
10                 sleep(Rate);
11                 GPS[time] = getGPS();}
12         Return 0; }
13 void timer_thread () {
14         while(){
15                 timer_set(timer2, Ra);
16                 if(timer_expired(timer2)){
17                         adaptation_trigger = 1;}
18                 Timer_reset(timer2); }}
19 void adaptation_thread () {
20         While(){
21         If( adaptation_trigger == 1) {
22                 BatteryLevel = Function(Rate);
23                 If(BatteryLevel < Batterythreshold){
24                         Speed = haversine(GPS[time-1:time]);
25                         Rate = Solve(Speed,BatteryLevel) ;}
26                 Timer_reset(timer2) // reset the timer
27                  adaptation_trigger = 0; }}}
```

Listing 1: Pseudo-C code for GPS Use Case.

As shown in Listing 1, the pseudo-code for the adaptation includes threads, timers, function dependencies, etc. With dynamic and complex adaptation policies it becomes difficult for the programmer to write the code. For example, if the programmer wants to initiate the adaptation at a fixed time every day or according to the time stamps of particular data in addition to the rate of adaptation, then the programmer must start a new timer for that purpose. That creates additional complexity when the programmer wants to add new interesting events over time, change the relationships between variables, or change the solution itself. For all these changes, the programmer must dive into the low-level details which are not always necessary for each change. If the programmer just wants to add a new time-stamp to trigger the adaptation, it should not require extensive knowledge of timers and threads in the whole program. Hence, there is a need for an abstraction that can enable the programmer to achieve goals without delving for every low-level detail. Such an abstraction did not exist until now, to the best of the authors' knowledge, and it has further implications since it also brings the ability to reuse and extend adaptation policies.

4 PROPOSED ABSTRACTION

In this section we propose, AdaptC, a high-level programming abstraction that can enable not only ease in programming but also ease in debugging and re-usability of the adaptation policies written by the programmer. One of the examples of the complexities with the current state of practice, shown in the code in Listing 1, is that the interesting events are not integrated into the code. Hence the programmer would have to always remember which functions to check for the adaptation. AdaptC can provide a better structure to write complex adaptation policies for various applications.

```
1  Block Trigger T {
2          //Combination of different triggers
3          // a fixed rate
4          Use consecutive_time 10s
5          // at fixed system time
6          Use time_stamp 00:00
7          // use different flags or events
8          Use flags }
9  Block Solution S {
10         Use Function f
11         Use Function g
12         Use Constraint
13         Uses Variables a b c d
14         //solve
15         return a }
16 Block Constraint B {
17         //define the constraint
18         return true/false }
19 Block Function f {
20         Use variables b, c, d
21         //operation
22         return b }
23 Block Function g {
24         Use variables a,c
25         //operation
26         return a }
27 // Adaptation here
28 Block Adaptation {
29         If Trigger = Active:
30                 Solve a
31         return a }
```

Listing 2: Abstract Pseudo-C code

Listing 2 shows an implementation of AdaptC. We divide the code into five blocks based on the proposed generic model of the design features described in Section II. These components are called blocks and named as follows: *Function, Constraint, Trigger, Adaptation, and Solution.*

- The Trigger block in line 1 allows the programmer to define when the adaptation must occur. In this block, the programmer can have fixed timers, periodic timers, or flags across the application. Hence, the programmer can have a flag named interestingEvent, use it across the complete application and, whenever that flag is raised the adaptation is triggered.
- The second block is Constraints in line 16. It allows the programmer to define one or many constraints that affect the adaptation. This block will return a boolean result of true or false respectively, depending on whether the constraints are satisfied or not.
- Next block, expressed in line 19, is the Function block or possibly a set of the block. This set of blocks defines all the variables and relationships between them. The number of function blocks depends, by convenience, on how to define multiple relationship between different variables. There can also be local variables used in these blocks and this allows ease of access in defining these relationships.

- The next block is the Adaptation block in line 28. This block basically checks for the trigger and if the trigger is active it calls the next block named Solution.
- Finally, the Solution block in line 28 solves the objective function subject to the defined constraints. Once a solution is achieved, the block returns it to the Adaptation block.

As an example, Listing 3 shows AdaptC applied to the HVAC use case as following.

```
1  Block Trigger_HVAC {
2          //Combination of different triggers
3          // a fixed rate
4          Use consecutive_time 10s
5          // at fixed system time
6          Use time_stamp 00:00
7          // use flags or events across application
8          Use flags }
9  // The soltuion for desired goals
10 Block Solution_COST {
11         call Function_TEMP()
12         call Function_COST()
13         //calculate required power
14         return Power }
15 Block Constraint_TEMP {
16         //define the constraint
17         if T_{S}+\delta<TEMP<T_{S}-\delta:
18                 return true
19         else
20                 return false
21 }
22 Block Function_TEMP {
23         Use Volume, Power
24         //calculate Temperature
25         return TEMP }
26 Block Function_COST {
27         Use Time, Power
28         //calculate COST
29         return COST }
30 // Adaptation here
31 Block Adaptation_Power {
32         If Trigger = Active:
33                 a = Solution_COST()
34                 return a }
```

Listing 3: Abstract Pseudo-C code for HVAC Use Case

5 PRELIMINARY EVALUATION

To validate the feasibility of the proposed abstraction, AdaptC, we implemented it for Contiki using nesC [16] and Python to support only basic functionalities as a proof of concept. For a concrete example, we implement the HVAC application using AdaptC and we compare it against an implementation following state-of-practice approaches. In this section, we discuss the technical details of that implementation and highlight some characteristics that help in understanding the benefits of the AdaptC. The same characteristics can also apply to many other applications such as smart homes, and smart irrigation systems, whose details we do not discuss for the purpose of conciseness.

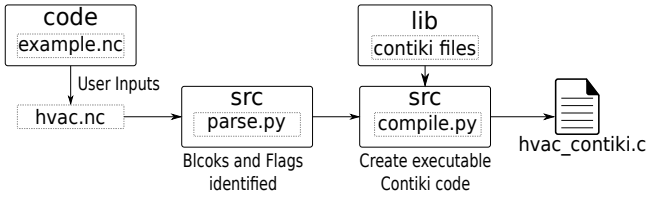

Figure 4: Implementation of proposed abstraction

AdaptC is implemented to parse the nesC code provided by the user and identify different *blocks* as mentioned in Listing 2. Then those are compiled into Contiki code. All of this is done using python scripts. The implementation is explained in Figure 4. To use the AdaptC, a programmer must obtain the repository. It is divided into three directories, *code*, *lib*, and *src*. The *code* folder contains already written examples. The programmer can find files such as of *example.nc* written in nesC equivalent to the abstract pseudo code showed earlier in Listing 2. This can be modified for the intended application by the programmer. For example, in the case of the HVAC application, the *example.nc* code can be modified according to Listing 3. The *lib* folder contains libraries built for the compilation of *example.nc* code into a workable Contiki code. That code can be executed inside Contiki or deployed using Contiki. These libraries are written in C and modified from the Contiki repository available on GitHub [12]. The *src* folder contains few scripts written in python. The *parser* script reads the code provided to by the programmer such as *example.nc* and identifies each block in it. The *compile* script uses the files from the *lib* directory to create the Contiki code and add each block from *example.nc* file in it.

With the help of AdaptC it becomes easier to reuse the same adaptation policies for different nodes, by just changing input parameters. For example, in the HVAC application, the programmer sets a constraint on the temperature which is affected by volume and power. Later on, another programmer might want to use the same policy in an area where the temperature is affected by other parameters such as the number of people, movement, etc. Hence, the second programmer can easily reuse the same adaptation policy by just slightly changing the input parameters. In addition to being reused, the applications are required to evolve with new requirements as well. For example, in wildlife monitoring using GPS sensor, the programmer might wish to monitor the health condition of the animal. This can add more interesting events such as any rest taken by the animal, elevation reached during the day, etc., and that could be easily achieved by adding these new events in the Trigger block.

Basic Demands	Rapid Development	Easy Maintenance	Reliability	Portable	Efficient	Learnable	Reusable	Pedagogic Value
Proposed Abstraction (AdaptC)	✓	✓	!	✓	!	✓	✓	✓

Figure 5: User demands for programming languages

A programming language must meet a few basic demands of the user community. We evaluate our abstraction, AdaptC, for those basic demands provided by [1] as shown in Figure 5. We have already discussed the re-usability earlier. The ability to extend

the code implies the ease of maintenance. Also using AdaptC the programmer is able to write diverse goals quickly since it takes away the complexities of timers, threads etc. Hence that enables Rapid Development. In addition, AdaptC allows adaptation policies to work on different nodes, hence the support of portability is provided. The design features and their modularity contribute to the ability to learn as it is easier to understand the role of each feature and their dependencies. Rest of the properties, namely Reliability and Efficiency still remains to be evaluated.

These are preliminary evaluations based on software engineering concepts. Since we are not aware of any existing abstractions for adaptation policies to build WSNs, a direct comparison was not possible. For further evaluation of the performance, we plan to implement it with different applications. With those implementations, a more detailed evaluation of software engineering elements such as the impact on variables, lines of code and functions will be possible.

Programming Abstraction	Hardware Agnostic	Evolvable Code
T-Res	✓	x
PyFuns	✓	x
AdaptC	✓	✓

Figure 6: Comparison with existing abstractions

Despite the absence of macroprogramming work specifically for adaptation policies, there is still a great amount of work to support the programmer. Thus, we try to compare the AdaptC with some of the relevant state of the art as shown in Figure 6. We selected two recent frameworks aiming at providing some sense of adaptation in WSNs and re-usability for the applications, namely T-Res [4] and PyFUNS [8]. Both aim at providing support to the programmer to write applications without the knowledge of low-level features, i.e., they are hardware agnostic. However, there is no support to write adaptations for the applications that change at run-time, i.e., evolvable applications. While AdaptC is able to support both changes in the node hardware and application code.

6 CONCLUSION

In this paper, we have introduced the problems in writing complex adaptation policies for Wireless Sensor Networks. We have demonstrated this by drawing on two different use cases. We have also exhibited the need of different user requirements. Drawing from the two different use cases, we have built a generic model which illustrates the complexities in adaptations.

Following the generic model for adaptation policies, we defined a new programming abstraction, AdaptC, that allows a programmer to write such policies without explicit dependence on low-level node features. In addition, we have implemented one of the use cases with and without abstractions in Contiki and nesC, and in both languages, it has offered the desired adaptability.

Our future work shall examine this abstraction further and evolve towards developing a complete macroprogramming solution with AdaptC for the Context-Aware Wireless Sensor Network. The immediate next steps would be to extend the implementation of AdaptC to support existing contributions for macroprogramming [4, 8]. We also plan on extending this abstraction for heterogeneous devices. With that, the aim is to build a system which can adapt to changes in both hardware and software requirements of the user.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Luca Mottola for his significant contribution to the *AdaptC*. The authors would also like to thank the anonymous referees of SAC 2019 for their valuable comments and helpful suggestions.

This work was partially supported by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology), within the CISTER Research Unit (CEC/04234); also by the Operational Competitiveness Programme and Internationalization (COMPETE 2020) under the PT2020 Partnership Agreement, through the European Regional Development Fund (ERDF), and by national funds through the FCT, within project(s) POCI-01-0145-FEDER-029074 (ARNET); and also by the EU ECSEL JU under the H2020 Framework Programme, within JU grant nr. 737422 (SCOTT project, www.scottproject.eu).

REFERENCES

[1] [n. d.]. Evaluating Programming Languages. https://courses.cs.washington.edu/courses/cse341/02sp/concepts/evaluating-languages.html
[2] Pooyan Abouzar, David G Michelson, and Maziyar Hamdi. 2016. RSSI-based distributed self-localization for wireless sensor networks used in precision agriculture. *IEEE Transactions on Wireless Communications* 15, 10 (2016), 6638–6650.
[3] Mikhail Afanasov, Luca Mottola, and Carlo Ghezzi. 2014. Context-oriented programming for adaptive wireless sensor network software. In *2014 IEEE International Conference on Distributed Computing in Sensor Systems*. IEEE, 233–240.
[4] Daniele Alessandrelli, Matteo Petraccay, and Paolo Pagano. [n. d.]. T-res: Enabling reconfigurable in-network processing in iot-based wsns. In *IEEE International Conference on Distributed Computing in Sensor Systems, 2013*.
[5] Tomoyuki Aotani and Gary T Leavens. 2016. Towards Modular Reasoning for Context-Oriented Programs. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*. ACM, 8.
[6] Jakob E Bardram. 2005. The java context awareness framework (JCAF)–a service infrastructure and programming framework for context-aware applications. In *International Conference on Pervasive Computing*. Springer, 98–115.
[7] Fehmi Ben Abdesslem, Andrew Phillips, and Tristan Henderson. 2009. Less is More: Energy-efficient Mobile Sensing with Senseless. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds (MobiHeld '09)*. ACM, 61–62.
[8] Stefano Bocchino, Szymon Fedor, and Matteo Petracca. 2015. Pyfuns: A python framework for ubiquitous networked sensors. In *European Conference on Wireless Sensor Networks*. Springer, 1–18.
[9] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, 21–21.
[10] Amol Deshpande, Carlos Guestrin, and Samuel Madden. 2005. Resource-Aware Wireless Sensor-Actuator Networks. *IEEE Data Eng. Bull.* 28, 1 (2005), 40–47.
[11] Jean-Philippe Diguet, Yvan Eustache, and Guy Gogniat. 2011. Closed-loop–based self-adaptive Hardware/Software-Embedded systems: Design methodology and smart cam case study. *ACM Transactions on Embedded Computing Systems (TECS)* 10, 3 (2011), 38.
[12] Adam Dunkels. 2003. The official git repository for Contiki, the open source OS for the Internet of Things. Retrieved September 22, 2018 from https://github.com/contiki-os/contiki
[13] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 455–462.
[14] Milan Erdelj, Nathalie Mitton, Enrico Natalizio, et al. 2013. Applications of industrial wireless sensor networks. *Industrial Wireless Sensor Networks: Applications, Protocols, and Standards* (2013), 1–22.

[15] Shashank Gaur, Raghuraman Rangarajan, and Eduardo Tovar. 2016. Extending t-res with mobility for context-aware iot. In *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE, 293–296.

[16] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC Language: A Holistic Approach to Networked Embedded Systems. *SIGPLAN Not.* 38, 5 (May 2003), 1–11. https://doi.org/10.1145/780822.781133

[17] Carlo Ghezzi, Matteo Pradella, and Guido Salvaneschi. 2010. Programming language support to context-aware adaptation: a case-study with Erlang. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 59–68.

[18] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-oriented programming. *Journal of Object Technology* 7, 3 (2008).

[19] Xin Hu, Rahav Dor, Steven Bosch, Anita Khoong, Jing Li, Susan Stark, and Chenyang Lu. 2017. Challenges in Studying Falls of Community-dwelling Older Adults in the Real World. In *Smart Computing (SMARTCOMP), 2017 IEEE International Conference on*. IEEE, 1–7.

[20] Stepan Ivanov, Kriti Bhargava, and William Donnelly. 2015. Precision farming: Sensor analytics. *IEEE Intelligent systems* 30, 4 (2015), 76–80.

[21] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. 2011. EventCJ: a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*. ACM, 253–264.

[22] Gian Pietro Picco Luca Mottola. 2011. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. 43, 3 (2011).

[23] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. 2012. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software* 85, 8 (2012), 1801–1817.

[24] Sanjin Sehic, Fei Li, and Schahram Dustdar. 2011. COPAL-ML: a macro language for rapid development of context-aware applications in wireless sensor networks. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*. ACM, 1–6.

[25] Norha M Villegas. 2013. *Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems*. Ph.D. Dissertation. University of Victoria.