Software Adaptation in Wireless Sensor Networks

Mikhail Afanasov, Politecnico di Milano, Italy
Luca Mottola, Politecnico di Milano, Italy and SICS Swedish ICT
Carlo Ghezzi, Politecnico di Milano, Italy

We present design concepts, programming constructs, and automatic verification techniques to support the development of *adaptive* Wireless Sensor Network (WSN) software. WSNs operate at the interface between the physical world and the computing machine, and are hence exposed to unpredictable environment dynamics. WSN software must adapt to these dynamics to maintain dependable and efficient operation. However, developers are left without proper support to develop adaptive functionality in WSN software. Our work fills this gap with three key contributions: *i)* design concepts help developers organize the necessary adaptive functionality and understand their relations, *ii)* dedicated programming constructs simplify the implementations, *iii)* custom verification techniques allow developers to check the correctness of their design before deployment. We implement dedicated tool support to tie the three contributions, facilitating their practical application. Our evaluation considers representative WSN applications to analyze code metrics, synthetic simulations, and cycle-accurate emulation of popular WSN platforms. The results indicate that our work is effective in simplifying the development of adaptive WSN software; for example, implementations are provably easier to test and to maintain, the run-time overhead of our dedicated programming constructs is negligible, and our verification techniques return results in a matter of seconds.

ACM Reference Format:
Mikhail Afanasov, Luca Mottola, Carlo Ghezzi, 2017. Software Adaptation in Wireless Sensor Networks. *ACM Trans. Autonom. Adapt. Syst.* V, N, Article A (January YYYY), 28 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Wireless Sensor Networks (WSNs) bridge the gap between the physical world and the computing machine [Jackson 1995] by seamlessly gathering data from the environment through sensors, and by taking actions on it through actuators. Because of their intimate interactions with the physical world, WSNs are exposed to multiple and unpredictable environment dynamics that affect their operation.

Example. Consider the use of WSNs to track wildlife [Pásztor et al. 2010]. Battery-powered WSN nodes are embedded in collars attached to animals, such as zebras or badgers. The devices are equipped with sensors to track the animals' movement, for example, based on GPS and accelerometer readings, and to detect their health conditions, for example, based on body temperature. Low-power short-range radios are used as proximity sensors by allowing nodes to discover each other whenever they are within communication range, using a form of periodic radio beaconing. A node logs the radio contacts to track an animal's encounters with other animals, enabling the study of their social interactions. The radio is also used to off-load the contact traces when in reach of a fixed base-station. Small solar panels harvest energy to prolong the node lifetime [Bhatti et al. 2016].

This work is supported by ... Author's addresses: {name}.{surname}@polimi.it

DOI: http://dx.doi.org/10.1145/0000000.0000000

Using battery-powered WSN devices makes energy a precious resource that developers need to trade against the system functionality, depending on the situation. For example, GPS sampling consumes non-negligible energy. The difference between consecutive GPS readings may be taken as an indication of the pace of movement, and used to tune the GPS sampling frequency and granularity. The contact traces can be sent directly to the base-station whenever the latter is within radio range, but they need to be stored locally otherwise. When the battery is running low, developers may disable GPS sampling to make sure the node survives until the next encounter with a base-station, not to lose the collected contact traces.

Problem. The traits of wildlife tracking applications, which we present here simply as an illustrative example, are commonly found in many WSN scenarios, including intelligent homes and buildings [Mattern et al. 2010], smart health-care [Ko et al. 2010], and mobile immersive computing [Mottola et al. 2006; Magerkurth et al. 2005]. *Multiple* environmental dimensions evolve *concurrently* and *independently*, such as location and battery levels. WSN software needs to adapt to such dynamics to maintain efficient performance. For example, in wildlife tracking, the inability to adapt to different situations may result in earlier battery depletion, preventing WSN nodes to eventually upload sensor data to the base-stations and thus hampering the analysis.

WSN nodes are, however, peculiar computing platforms with significant resource constraints. They typically feature 16-bit microcontrollers with a few KBytes of data memory. Mainstream development approaches are thus generally inapplicable: sound design concepts are largely missing [Picco 2010], programming often occurs using low-level languages [Mottola and Picco 2011], and the few dedicated verification techniques only target low-level functionality [Sasnauskas et al. 2010; Mottola et al. 2010; Li and Regehr 2010]. The problem is exacerbated whenever developers are to realize *adaptive* WSN software, whose complexity generally increases compared to the static case. In this area, proper developer support to implement adaptive functionality in concrete systems is arguably lacking.

To give a concrete feeling of the issues at stake, Figure 1 shows a simplified implementation of adaptive functionality using nesC [Gay et al. 2003], a dialect of C commonly used for WSN development. NesC function calls are asynchronous; results are returned using a notion of *event* that essentially operates as a callback. The code implements *only one* aspect of the adaptation needed in wildlife tracking: to send readings to the base-station whenever reachable, or to store them locally otherwise.

In Figure 1, multiple orthogonal concerns are intertwined and functionality are tightly coupled. For example, the decision on what operating mode to employ, that is, whether to consider the base-station as reachable, is implemented from line ⓳ to ㉔. This lies in the the same module as the adaptive processing itself from line ❼ to ⓱. Both functionality depend on the same global variable **`base_station_reachable`**, whose management is entirely on the programmer's shoulders. Moreover, the checks to perform before changing operating mode, such as those in lines ❽ and ⓫, are mixed with the functionality that changes the mode itself.

Using existing approaches to implement WSN software thus often results in entangled implementations that are difficult to debug, maintain, and evolve. Modifying the code in one place, for example, likely leads to further changes in several other places, increasing the programmer's effort even in the simplest cases. As the number of relevant environment dimensions and their combinations grows, WSN implementations quickly turn into "spaghetti code" [Finne et al. 2010], as visible also in publicly available WSN codebases [TinyOS 2016]. Moreover, verifying the correctness of the combined functioning of different adaptation strategies becomes a challenge. Manual testing of all possible combinations is immensely time-consuming; therefore, WSN de-

```

module ReportLogs {
  uses interface Collection;
  uses interface DataStore;
}implementation {
 int base_station_reachable = 0;
 event msg_t Beacon.receive(msg_t msg) {
  if (!acceleromenter_detects_activity())
    return;
  if (call Battery.energy() <= THRESHOLD)
    return;
  base_station_reachable = 1;
  call GPS.stop()
  call BaseStationReset.stop();
  call BaseStationReset.startOneShot(TIMEOUT);}
 event void BaseStationReset.fired() {
  base_station_reachable = 0;
 }
 event void ReportPeriod.fired() {
  switch (base_station_reachable){
   case 0:
    call DataStore.deposit(msg);
   case 1:
    call Collection.send(msg);
  }
 }
}
```

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

velopers often deploy systems with little confidence in their correctness [Sasnauskas et al. 2010; Iwanicki et al. 2014].

Contribution. Our work is centered on a notion of *context*. Such a notion is vastly employed in various areas of computing, including proximate selection, contextual reconfiguration, contextual information, and context-triggered actions [Schilit et al. 1994; Abowd et al. 1999; Dey 2001], yet not in WSN software. We specifically consider a *context* to be a specific situation, including both environmental and system features, that WSN software might find itself in. This is similar to the notion of “situation” employed in context-aware computing [Coutaz et al. 2005]. A large body of work exists on context recognition through sensor data, in the presence of noise, and based on incomplete information [Schmidt et al. 1999; Kang et al. 2008; Kern et al. 2003]. Our work is complementary to these efforts, which are typically application-specific. We rather provide *general-purpose* support to help developers realize adaptive functionality in WSN systems[1], thus leveraging the existing body of work on top of a severely resource-constrained platform.

Following a brief survey of the state of the art in Section 2, our contribution unfolds in three parts:

— We conceive dedicated design concepts, described in Section 3, to support developers in the early phases when identifying the possible system’s evolutions. In our work, these emerge as a result of determining the different situations a WSN system may find itself in, what environmental dimensions are responsible for these situations, and the relations between different adaptation decisions.

[1]In this sense, the wildlife-tracking application, as well as all other applications we mention throughout the paper, are merely and intentionally-simplified illustrative examples to help the reader understand.

— We extend nesC with notions of Context-oriented Programming (COP) [Hirschfeld et al. 2008]. Section 4 describes the resulting language, called CONESC, which ameliorates the coupling between functionality, rendering implementations easier to understand and to maintain. The design concepts of Section 3 map to the programming constructs we introduce, easing the transition from design to implementation.
— We conceive automatic verification techniques to check the correctness of an application's design against the possible environment evolutions, as we describe in Section 5. Our techniques operate before deployment, thus requiring reduced effort than most exiting approaches [Iwanicki et al. 2014; Romer and Ma 2009]. Further, they quickly return counterexamples expressed with the same design concepts of Section 3, facilitating the identification of issues.

The three contributions are tied together by dedicated tool support. Section 6 describes GREVECOM, a visual tool that allows developers to *i)* specify their designs using the concepts in Section 3, *ii)* use these to generate code templates using the CONESC constructs in Section 4, and *iii)* automatically verify the designs against developer-provided correctness specifications using the techniques in Section 5. Still in Section 6, we also describe the compiler support we implement to automatically translate CONESC sources into plain nesC, which allows one to employ the nesC toolchain to obtain the binary for deployment on WSN devices.

Our evaluation, described in Section 7, considers three representative WSN applications. Based on these, we compare implementations based on our design concepts and CONESC, against functionally-equivalent implementations obtained using existing approaches and nesC. We find that the functionality in the former are less coupled, the complexity of code is decreased, and changes require less programming effort. When using CONESC, these benefits come at the price of a negligible run-time overhead in time and energy, which we quantify using cycle-accurate emulation. Next, we assess the scalability of our verification techniques, using increasingly complex application designs. For realistic instances, the verification process takes seconds, providing evidence of its practical use.

2. RELATED WORK

A substantial body of work exists on the design, implementation, and verification of adaptive software [Cheng et al. 2009]. Most of this, however, targets mainstream computing systems. WSNs, on the other hand, present peculiar characteristics that crucially impact the development process. Nevertheless, adaptation logic exists aplenty in WSNs [Zimmerling et al. 2012; Bourdenas et al. 2011], yet developers lack dedicated development support to embed these functionality in concrete systems.

Using embedded resource-constrained computing platforms, efforts close to ours can be divided into four categories. Model-driven approaches exist to support the design of adaptive applications, yet they tend to be domain specific or to result in monolithic implementations. Programming support for adaptive WSN applications is also investigated, although the application logic runs *outside* of the WSN, rather than right on the WSN devices as in our case. Finally, automatic verification tools exist for WSNs as well, but they are designed to operate directly on the code with no specific support to check the correctness of adaptive behaviors.

Design support. Subramanian and Katz [2000] define a component model to build self-adaptive WSN architectures. The work is domain-specific in that it targets static WSNs, that is, wherever nodes do not move in space. Many WSN applications, however, leverage the ability of WSN devices to operate autonomously in mobile settings, as in the example application in the Introduction. In this case, the approach of Subramanian and Katz [2000] would not be applicable.

In a similar vein, Diguet et al. [2011] provide design support that blurs the boundaries between hardware and software, gaining more flexibility in providing adaptive functionality. Their design, however, leads to application-specific implementations. In contrast, we aim to provide a *general* solution to the design, implementation, and verification of adaptive WSN software. We concretely demonstrate this by investigating diverse applications in Section 7.

Fleurey et al. [2011] propose a model-driven approach to develop adaptive WSN firmwares. They model an application as a state machine. The predicates defined over the application state determine behavioral variations. Whenever these predicates are found to hold, the state machine adapts its transitions. The source code is automatically generated. Differently, we do not aim at automatically generating the complete application code, but provide dedicated design concepts and programming constructs that allow for fine-grained optimizations.

Programming support. Three main programming approaches typically provide support for adaptation: Aspect-oriented Programming (AOP), Meta-programming, and Context-Oriented Programming (COP) [Salvaneschi et al. 2013].

AOP [Kiczales et al. 1997] allows developers to add behaviors to existing code with limited modifications. The new functionality is indicated through a "pointcut" specification within the existing code, which represents an entry point for the additional behavior to be triggered within the control flow. Such a technique is especially effective for so-called "crosscutting concerns", that is, functionality that cut across multiple abstractions. These typically affect the entire program, thus defying the traditional forms of modularization. Examples are logging, security, and transactions.

AOP can be implemented by modifying the underlying interpreter or execution environment, or through dedicated pre-processors. The difficulties in the former techniques led to most AOP implementations being realized through a process known as *weaving*, that is, a special case of program-to-program transformation. An aspect weaver reads the aspect-oriented code and generates appropriate functionally-equivalent source code in a traditional language with the aspects integrated.

AOP implementations based on weaving techniques proved effective in a number of complex scenarios [Filman et al. 2004]. On the other hand, WSN software is both significantly less complex than the kind of systems where AOP is typically employed, and rarely exhibits the kind of cross-cutting concerns AOP is most effective for. Further, AOP tends to obscure the control flow, making implementations more difficult to test. This may become a key concerns in WSNs because of the little visibility into the system's internals [Romer and Ma 2009].

In Meta-programming [Visser 2002], programs can treat themselves as their data. Therefore, a program may be designed to read, generate, analyse, or transform other programs, and even to modify itself while running. This enables greater flexibility to efficiently handle new situations without recompilation. While providing an extreme form of software adaptation, meta-programming however requires modifications of the binary at run-time, which is hardly feasible on resource-constrained devices.

Several systems employ some form of COP to implement WSN applications [Bardram 2005; Wood et al. 2008; Sehic et al. 2011]. In these cases, however, the WSN devices are considered as mere sources of raw data, whereas adaptation happens on the software running outside of the WSN, for example, on a standard machine that acts as a base-station. In our work, COP supports developers in implementing adaptive functionality that runs right onto the WSN devices. In doing so, our work needs to consider the limitations dictated by the target platforms.

Automatic verification. Most approaches to the automatic verification of embedded software work directly on the source code. For example, Clarke et al. [2004] present a

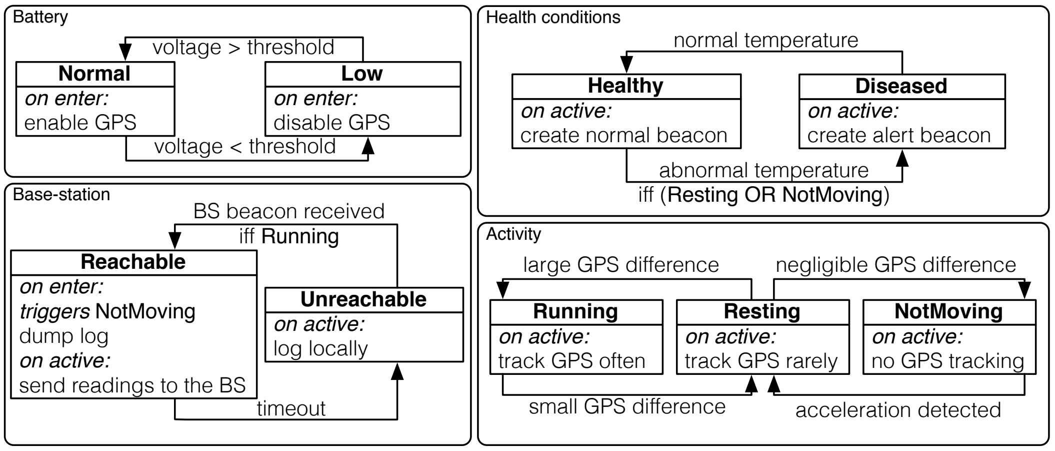


Fig. 2: Context-oriented model for a wildlife tracking application.

verification tool that checks ANSI-C sources against safety properties such as correctness of pointer constructs. The SLAM toolkit of Ball and Rajamani [2002] determines whether a C program violates programmer-provided correctness rules. BLAST [Beyer et al. 2007] formally proves that a C program satisfies given safety properties.

In the domain of WSN software, Bucur and Kwiatkowska [2009] focus on the automatic verification of applications written in nesC, similar to the T-Check tool of Li and Regehr [2010] that uses several heuristics to battle the state-space explosion due to operating at the level of nesC sources. Kleenet [Sasnauskas et al. 2010] focuses on the network interactions across WSN devices running the Contiki [Dunkels et al. 2004] operating system. Anquiro [Mottola et al. 2010] takes Contiki code as input as well, and implements several state abstraction mechanisms to combat the state space explosion.

Unlike the approaches above, our automatic verification occurs based on the context-oriented design, not on the actual implementation. This has pros and cons. On one hand, identifying potential issues early in the development process saves testing and debugging effort later. On the other hand, the transition from the design to the implementation still leaves the possibility of introducing defects in the actual code. Because of the direct mapping from the design concepts to the abstractions in CONESC and our tool support, however, we argue these risks are ameliorated.

3. DESIGN

We illustrate design concepts to support developers in identifying the different situations a WSN system may find itself in, their relations, and possible evolution in time. Next, based on the experience we accumulated in employing these concepts in multiple applications, we describe recurring design patterns.

3.1. Concepts

We introduce two key concepts: *i) individual contexts*, and *ii)* context *groups*. A context represents an individual situation the software running on a given WSN device may encounter [Coutaz et al. 2005]. Whenever that situation occurs, the software changes its functioning accordingly, implementing an appropriate adaptation decision. For example, in the wildlife-tracking application described in the Introduction, the reachability of the base-station based on the physical location of a device represents an individual context coupled to a corresponding functionality. This is different to the context and functionality representing the situation where the base-station is unreachable. A context group is a collection of contexts sharing common characteristics, for example, being determined by the same environment dimension. We may group together the two contexts representing the (un)reachability of the base-station, as both depend on a device's physical location.

Figure 2 represents the complete design of the wildlife tracking application based on contexts and context groups. The four context groups, shown as the outer boxes, represent collections of individual contexts depending on battery level, base-station reachability, as well as an animal's health conditions and activity levels. The individual contexts, shown as the inner boxes in every group, are described by a name and by actions taken when *entering* or *leaving* a context, and by processing executing as long as the context is *active*, that is, the context corresponds to the current situation. Context and context groups provide structure and help factor out the adaptation necessary to deal with independent environment dimensions.

At most one context is active in each context group at any point in time. However, multiple contexts belonging to different groups may be active at the same time. Contexts within the same group are tied with transitions that express the conditions triggering a change of the current context. In Figure 2, for example, a change in the battery voltage below a threshold triggers a change from the *Normal* to the *Low* context in the *Battery* group. The evolution of active contexts in different groups thus mimics the semantics of parallel state machines, but for the following features:

— Context transitions may contain *dependencies*. For example, if a body sensor reads an abnormal temperature, it might indicate that the animal is *Diseased*, and require a transition to the corresponding context. In this situation, however, an animal is most probably moving slightly or not at all; therefore, the active context in the *Activity* group should not be *Running*.
— Context activation may also *trigger* a transition in a different context group, as is the case in the *Reachable* context of Figure 2. Because the base-station is deployed at a known location, its reachability indicates the device is nearby. Therefore, we trigger a transition to the *NotMoving* context in the *Activity* group to disable GPS tracking and assume the base-station location as the one of the device.

The concepts we described are largely decoupled from a concrete programming language, enabling their implementation in different WSN languages. On the other hand, we do not provide explicit support for distribution. Notwithstanding this limitation, we demonstrate in Section 7 that our design concepts, together with their concrete realization in CONESC, apply to a significant fraction of adaptive WSN applications.

3.2. Patterns

Based on experience, we observe distinct patterns emerging that provide structured ways to address specific types of adaptive functionality. These patters, discussed next, allow developers to express complex functionality with only a handful of concepts.

Behavior control. Different *behaviors* of the same high-level functionality are often represented in a single context group. Figure 2 shows one such example in the *Base-station* group, which includes two different behaviors for the same high-level functionality of processing the collected logs. The same pattern is found also in other applications. For example, an adaptive protocol stack [Gnawali et al. 2009; Fotouhi et al. 2012] uses different protocols for the same underlying physical layer depending on node's mobility. The high-level packet relay functionality is expressed with a similar design, as we show in Section 7.

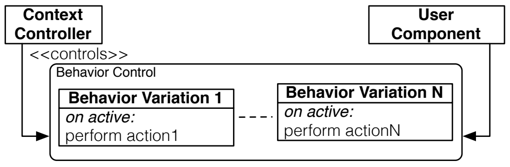


Fig. 3: Behavior control pattern.

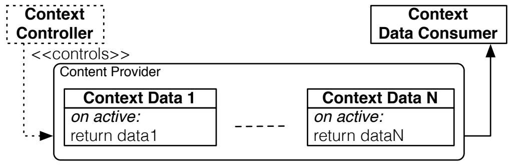


Fig. 4: Content provider pattern.

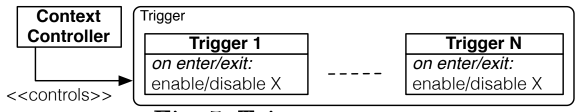


Fig. 5: Trigger pattern.

Figure 3 shows an abstract view of the behavior control pattern and its characterizing elements. Developers define a single context group to export a functionality whose behavior depends on the active context. An external context "controller" drives the transitions between the contexts in the group. In the wildlife tracking application, for example, the context controller checks if beacons are received indicating a nearby base-station, and accordingly activate a specific context in the *Base-station* group.

Content provider. We also observe cases where context-dependent *data* is offered to other functionality with little to no processing involved, differently from the behavior control pattern that provides non-trivial context-dependent processing. An example is in the *Health conditions* group of Figure 2. Depending on the active context, the periodic beacon is generated differently. The actual processing that involves the beacon happens elsewhere in the system; in this case, throughout the network stack responsible for transmitting the beacon over the air. We notice this pattern in other applications as well. For example, the smart-home application we describe in Section 7 employs the same pattern to manage user preferences depending on time of the day.

The characterizing elements, abstractly shown in Figure 4, differ from those of behavior control. The "controller" component is often fairly trivial. For example, the "controller" in the smart-home application of Section 7 simply checks the time of the day. Differently, the component consuming the context-dependent data plays a key role. While functionality structured as behavior control can be considered stand-alone, the content provider needs to be tailored to the data consumer.

Trigger. We also recognize designs where contexts are used only to *trigger* specific operations, especially on hardware components, without any significant context-dependent processing or data offered. An example is the *Battery* group in Figure 2. The contexts in the group are used to enable or disable the GPS sensor depending on the battery level. In the smart-home application of Section 7, we notice a similar pattern when tuning lights in a room. Depending on the amount of natural light, different contexts are activated that tune the artificial lighting accordingly.

As shown in Figure 5, the "controller" drives context transitions similar to behavior control. However, unlike the other patterns, there is no external components that either uses context-dependent functionality or consumes context-dependent data.

4. PROGRAMMING SUPPORT

To support programmers in implementing adaptive WSN software, we borrow concepts from Context-oriented Programming (COP) [Hirschfeld et al. 2008] and embed them in the nesC language. We choose nesC because of the widespread adoption and stable tool-chain. However, as we discuss next, we are not tied to nesC; designs similar to the

```
context group BaseStationG {
 layered command void report(msg_t msg);
}implementation {
 contexts Reachable,
          Unreachable is default,
          MyErrorC is error;
 components Routing, Logging;
 Reachable.Collection -> Routing;
 Unreachable.DataStore -> Logging;
}
```

Fig. 6: *Base-station* context group in CONESC.

one we explain next can be obtained in a range of WSN programming languages, for example, functional ones [Newton et al. 2007; Mainland et al. 2008].

We provide the necessary background on COP and nesC in Section 4.1. Next, Section 4.2 illustrates the main programming constructs of our COP extension to nesC, called CONESC. The design concepts of Section 3 map directly to these constructs, facilitating the transition from design to implementation. Section 4.3 describes the concrete use of these constructs in implementing adaptive functionality. Section 4.4 describes how CONESC programmers control context transitions.

4.1. nesC and COP

nesC is a component-based event-driven programming framework for WSNs, 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; for example, querying a sensor. Commands are non-blocking and return immediately. Events are used to collect the results asynchronously. Because of the duality between commands and events, nesC interfaces are bidirectional: data flows both ways between components connected through the same interface. Component *configurations* specify the wirings among components; these are components themselves, can provide interfaces, and be wired to other configurations or components.

COP is a programming paradigm often employed to implement adaptive software. Central to COP is the notion of *layered function*, that is, a function whose behavior changes depending on the current situation and *transparently* to the caller. COP already proved effective in creating adaptive software in mainstream applications, such as user interfaces [Keays and Rakotonirainy 2003] and text editors [Kamina et al. 2011]. In these settings, programmers rely on COP extensions of popular high-level languages, such as Java [Sehic et al. 2011].

Realizing a COP extension of nesC is non trivial. Because of the resource constraints of the underlying platform, nesC itself is quite limited. For example, nesC programmers cannot create run-time instances of components, while component wirings in nesC are statically defined; therefore, they cannot change at run-time. Further, the use of dynamic memory is discouraged; typical microcontrollers on WSN devices offer no memory protection, so bugs in memory handling may have disastrous effects. These features are often employed to realize COP extensions of existing programming languages [Salvaneschi et al. 2012], yet they are not available in nesC.

4.2. Context Groups and Individual Contexts

We map the notion of context group in Section 3 to an extended form of nesC configuration. The interface of a context group is used to declare the prototype of layered func-

```
 1 context Unreachable {
❷  transitions Reachable iff ActivityG.Running;
 3  uses interface DataStore;
 4 }implementation {
 5  event void activated(){//...}
 6  event void deactivated(){//...}
 7  command bool check(){//...}
 8  layered command void report(msg_t msg){
❾   call DataStore.deposit(msg);
10  }
11 }
```

Fig. 7: *Unreachable* context.

```
 1 context Reachable {
 2  uses interface Collection;
 3  uses interface DataStore;
 4  transitions Unreachable;
 5  uses context group BatteryG;
❻  triggers AcitivityG.NotMoving;
 7 } implementation {
 8  event void activated(){
❾   call DataStore.dump_log();}
❿  event void deactivated(){//...}
⓫  command bool check(){
12   return call BatteryG.getContext() == BatteryG.Normal;
13  }
14  layered command void report(msg_t msg){
⓯   call Collection.send(msg);
16  }
17 }
```

Fig. 8: *Reachable* context.

tions. Their behavioral variations are expressed by individual contexts in the group, whose definition extends that of nesC components.

Figure 6 shows the CONESC implementation of the *Base-station* group of Figure 2. The layered function **`report`** is declared in line ❷ using the **`layered`** keyword. The actual behavior of such a function, in fact, depends on the active context in the group. If the base-station is reachable, messages must be transmitted over the radio; otherwise, they should be locally stored until the next encounter with a base-station.

The individual contexts in the group are specified in line ❹ of Figure 6 using the **`contexts`** keyword. Of the three contexts in the example, the *Unreachable* one is defined as **`is default`** in line ❺, that is, the context is active at start-up. The **`is error`** keyword in line ❻ indicates an *error* context, whose semantics we illustrate next. Including an error context is not mandatory; our translator generates an empty one if not declared. The remainder of the context group follows the normal nesC syntax and semantics for configurations.

Figure 7 and 8 illustrate an excerpt of the CONESC implementations of the contexts *Reachable* and *Unreachable* in Figure 2. Individual contexts extend the notion of nesC component by providing different implementations of the layered function defined in the corresponding group, for example, function **`report`** defined in Figure 6. In context *Unreachable*, the implementation of **`report`** in line ❾ of Figure 7 deposits the message in the local data store; otherwise, if context *Reachable* is active, function **`report`** in line ⓯ of Figure 8 transmits the message over the air using a data collection protocol. The caller of function **`report`**, however, only owns a reference to the context group and

```
 1 module User {
 ❷  uses context group BaseStationG;
 3 }implementation {
 4  event void Timer.fired() {
 ❺   call BaseStationG.report(msg);
 6  }
 ❼  event void BaseStationG.contextChanged(context_t con) {
 8   if(con == BaseStationG.Reachable) // DO SOMETHING...
 9  }
10 }
```

Fig. 9: User component.

```
 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;
10  }
11 }
```

Fig. 10: Base-station context controller.

does not need to know what context is currently active therein; changing to the most appropriate behavior happens transparently, as we illustrate next.

Individual contexts may specify actions to take when entering or leaving a context; for example, to initialize variables or to perform the necessary clean-up. As discussed in Section 3, on entering context *Reachable*, a programmer may want to off-load the contact traces, since the base-station is within reach. Programmers place the necessary functionality as the body of a predefined **`activated`** event, as in line ❾ of Figure 8. The event is automatically triggered when entering the context. The **`deactivated`** event, shown in line ❿ of Figure 8, is dual.

4.3. Execution

Figure 9 shows an example component that relies on the functionality implemented by the layered function **`report`**. In line ❺, when a timer fires, the call occurs without explicitly referencing either of the individual contexts that provides a concrete implementation of **`report`**. The binding corresponding to the active context happens dynamically. This is possible as our CONESC translator, described in Section 6, automatically generates the code that implements the necessary dynamic dispatching.

Figure 10 also shows the CONESC code that implements context detection and correspondingly activates a context in the *Base-station* group. Programmers trigger transitions between contexts using the **`activate`** keyword. For example, in line ❺ of Figure 10, a transition to context *Reachable* is triggered as soon as a radio beacon from the base-station is received. In the same event handler, a timer is started to keep track of the time since the last radio beacon. When the timer fires before being reset by the next radio beacon, the base-station is considered unreachable and a transition to the corresponding context is triggered in line ❾.

The context detection functionality in Figure 10 is intentionally simplified for illustration purposes. However, nothing prevents developers from employing more sophisticated criteria to trigger a context change, based on application requirements, platform

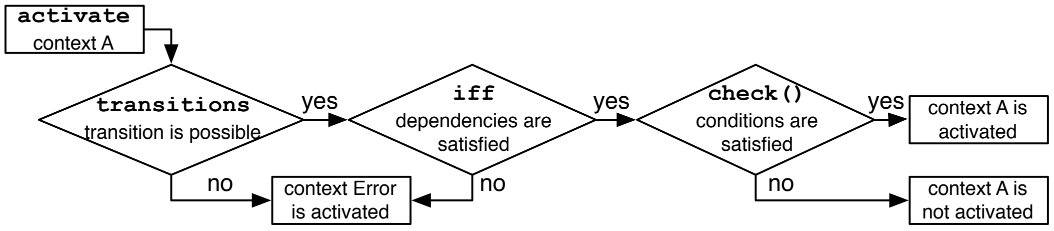


Fig. 11: Context activation rules.

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

Fig. 12: *NotMoving* context.

characteristics, and the vast literature on context detection [Schmidt et al. 1999; Kang et al. 2008; Kern et al. 2003].

For example, consider how to detect when an animal's body temperature becomes abnormal to activate the *Diseased* context of Figure 2. Given application requirements may dictate this situation to be detected when the sensor readings surpass a threshold for a prolonged period. Different application requirements may prescribe the same to be recognized based on fluctuating patterns over the evolving time series of sensor readings. Using CONESC, both can be implemented to execute the **`activate`** instructions for the *Diseased* context. Similar observations apply to implement functionality to tame the inaccuracies of sensor readings, including those due to calibration, occlusion, and noise [Wu et al. 2002; Gustafsson 2010; Olfati-Saber 2007].

Notice how Figures 6 to 10 implement the behavior control pattern, described in Section 3.2. The component of Figure 9 relies on a functionality whose behavior is determined by the context controller of Figure 10. The benefit is that context-dependent functionality, as well as the logic driving context detection and activation, are completely decoupled and implemented in different modules. We demonstrate in Section 7 that this renders implementations easier to understand, debug, and maintain.

4.4. Context Transitions

Programmers need to take extra care of context transitions, as they may drastically change the application behavior. CONESC provides specific features to this end.

Every time the active context in a group changes, user components are notified through a predefined event **`contextChanged`**. Programmers may implement the corresponding event handler, as exemplified in line ❼ of Figure 9, to react to the corresponding context change with specific actions. The name of the newly activate context is provided as a parameter to the event.

When transitioning to a new context, the processing transparently traverses several checking stages, shown in Figure 11. If all checks succeed, the new context is activated; otherwise, either the transition is canceled or the *Error* context is activated, depending on the kind of failure:

(1) Not all transitions are feasible between contexts. For example, within the *Activity* group of Figure 2, it is only possible to move from context *NotMoving* to *Resting*. This is encoded in CONESC with the keyword **`transitions`**, exemplified in line ❷ of Figure 12, followed by the list of target contexts. A violation of this specification

may indicate a significant design or hardware/software issue; the *Error* context is thus activated where programmers may take dedicated countermeasures.

(2) The context-oriented design may indicate dependencies among transitions, as described in Section 3.1. For example, within the *Base-station* group, a transition from *Unreachable* to *Reachable* depends on context *Running* being active, indicating that an animal is actually moving when approaching the base-station. Such dependencies are specified in CONESC with the keyword **`iff`**, as shown in line ❷ of Figure 7, followed by the fully qualified name of the context this transition depends on. A dependency violation is treated similarly to the case above.

(3) Programmers may express "soft" requirements whose violation may not necessarily indicate a serious issue. For example, before activating the *Reachable* context, programmers may check that sufficient energy is available to transfer the collected logs to the base-station. If not, they may defer the activation of the *Reachable* context until sufficient energy is harvested. To specify these checks, programmers implement a predefined command **`check`** in individual contexts, which returns a Boolean value that grants permission for context activation. An example is shown in line ⓫ of Figure 8. Should the check fail, the previous context remains active.

Finally, the design concepts illustrated in Section 3 allow one to express triggers between transitions. This is the case, for example, when transitioning to the *Reachable* context in the *Base-station* group. As the base-station is nearby at a known location, we assume the node location to be the same and spare the energy consumption of GPS sampling by triggering a transition to the *NotMoving* context. CONESC allows programmers to express this processing by using the **`triggers`** keyword, as shown on line ❻ of Fig. 8, followed by the fully-qualified name of the context that must be automatically activated when entering.

5. VERIFICATION

Checking the correctness of WSN software is difficult in general. Worse is the case when WSN software needs to be adaptive to environment dynamics that are, in general, unpredictable. Traditional testing approaches struggle in exhaustiveness [Iwanicki et al. 2014]. Further, as the number of relevant environment dimensions grows, the number of possible situations the software may encounter increases exponentially; thus, scalability also becomes a hampering factor. As a result, many WSN software implementations undergo very limited verification before deployment [Picco 2010].

Based on the concepts illustrated in Section 3, we conceive a technique to automatically, yet exhaustively check the correctness of the application's design. We use model checking techniques to identify issues such as contexts that may become unreachable or deadlock situations that may block the software in a specific configuration. In addition, our techniques support the verification of developer-provided arbitrary properties expressed in Computation Tree Logic (CTL) [Clarke et al. 1986].

Using model checking to verify the correctness of the context-oriented design is, however, not immediate. Existing tools do not directly support the semantics of the design concepts described in Section 3. For example, the notion of dependency across transitions or the existence of triggers between contexts do not directly translate into the abstractions used by most existing model checkers.

Rather than building a new tool from scratch, we illustrate next an algorithm that translates a context-oriented design into a form compatible with that of modern model checking tools. As described in Section 6, our tool-chain uses the latter to run the actual verification process. Next, it translates back the results of the verification into the original form, easing their interpretation by developers. Leveraging an existing tool to check the correctness of the context-oriented design allows us to rely on the robustness and maturity of the latter. Particularly, the verification engine is re-used from the existing tool, which reduces the risk of introducing bugs in the verification mechanisms as compared to developing a new tool.

Transformation. We choose to transform a context-oriented design defined according to Section 3 into a semantically equivalent finite state machine (FSM), that is, one whose state space features a one-to-one mapping with the state space of the original context-oriented design. An FSM representation allows us to employ many state-of-the-art model checkers [Clarke et al. 1999].

Let us consider a generic context-oriented design with a set G of context groups. We call $g.C$ the set C of contexts in group $g \in G$; for example, in the context-oriented design of Figure 2, $\mathit{Battery}.C = \{\mathit{Normal}, \mathit{Low}\}$. Each context c has a set O of outgoing transitions represented as tuples $\langle o, e \rangle$, where o is the target context and e is the label for that transition. In Figure 2, for example, $\mathit{Resting}.O = \{\langle \mathit{Running}, \mathit{large\ GPS\ difference} \rangle, \langle \mathit{NotMoving}, \mathit{negligible\ GPS\ difference} \rangle\}$. Triggers between contexts are defined as an attribute T of an individual context c; for example, $\mathit{Reachable}.T = \mathit{NotMoving}$ in Figure 2, or $\perp$ if no trigger is defined.

We call D the set of dependencies for context transitions in the original design. An element in D takes the form $\langle c_1, c_2, \overline{c} \rangle$, where c_1 and c_2 are the originator and target contexts of a transition, respectively, and $\overline{c}$ is the context whose activation is necessary for this transition to be taken. Based on Figure 2, for example, both $\langle \mathit{Healthy}, \mathit{Diseased}, \mathit{Resting} \rangle$ and $\langle \mathit{Healthy}, \mathit{Diseased}, \mathit{NotMoving} \rangle$ belong to D.

The corresponding FSM has a set S_{FSM} of states and a set T_{FSM} of transitions. A state $s \in S_{FSM}$ is a n-tuple $\langle c_1, c_2, \ldots, c_n \rangle$, $n = |G|$, where each element c_i in the tuple represents an individual context in a *distinct* group $g_i \in G$. Each state $s \in S_{FSM}$ is obtained by considering each active context in every group. A transition $t \in T_{FSM}$ is a tuple $\langle s_1, s_2, e \rangle$ representing a transition from $s_1 \in S_{FSM}$ to $s_2 \in S_{FSM}$ with label e. States s_1 and s_2 differ by at least one individual context $c' \in s_1$ that changes to $c'' \in s_2$.

A transition where *exactly* one $c' \in s_1$ changes to $c'' \in s_2$, and no other c''' changes between s_1 and s_2, exists when it satisfies the following constraints:

(1) For an individual context c' in s_1, a transition in the original context-oriented design exists in $c'.O$ to an individual context c'' in s_2 with no triggers, that is, $\langle c'', e \rangle \in c'.O$ and $c''.T = \perp$. Label e in transition t is the label of $\langle c'', e \rangle \in c'.O$.

(2) If a dependency $\langle c', c'', \overline{c} \rangle \in D$ exists, then $\overline{c}$ is also found in s_1, that is, if a dependency exists on this transition, the dependent context is part of s_1 and thus the dependency is satisfied.

Differently, a transition where $c' \in s_1$ changes to $c'' \in s_2$, with other c''' possibly changing between s_1 and s_2, exists when for an individual context c' in s_1, a transition in the original context-oriented design is found in $c'.O$ to an individual context c_t whose trigger points to c''. This means, $\langle c_t, e \rangle \in c'.O$ and $c_t.T = c''$. Label e in transition t is the label of $\langle c_t, e \rangle \in c'.O$.

After pruning unreachable states, the FSM represents the feasible combinations of active contexts in different groups and their transitions.

Examples. Figure 13 shows the FSM obtained for a context-oriented model that, for simplicity, only includes group *Health conditions* and *Activity* from Figure 2. This slice of the model does not include triggers, so only constraints (1) and (2) above apply. In group *Health conditions* of Figure 2, the transition from *Healthy* to *Diseased* has label *abnormal temperature* and a dependency on either *Resting* or *NotMoving*; therefore, $D = \{\langle \mathit{Healthy}, \mathit{Diseased}, \mathit{Resting} \rangle, \langle \mathit{Healthy}, \mathit{Diseased}, \mathit{NotMoving} \rangle\}$. In the resulting FSM, the existing dependencies are satisfied for all states $\langle \mathit{Healthy}, * \rangle$ but

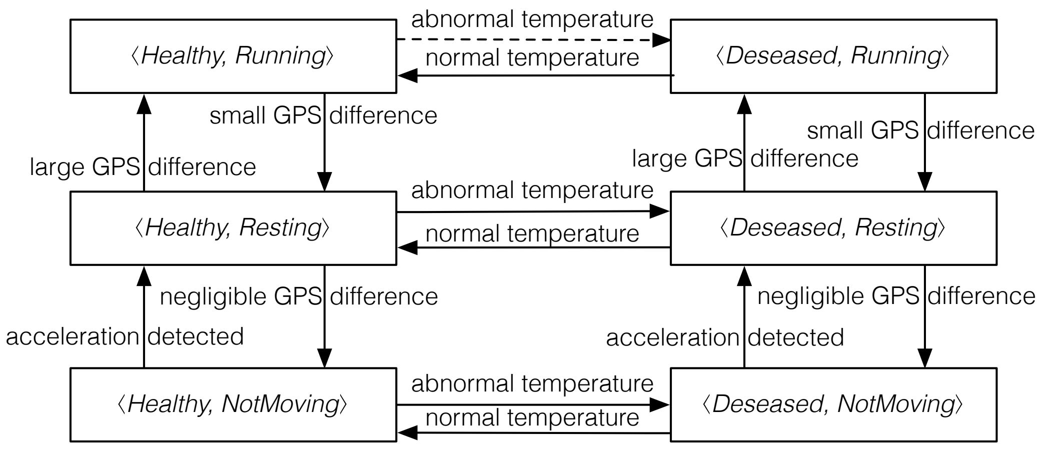


Fig. 13: FSM model obtained from a context-oriented design that only includes the groups *Health conditions* and *Activity* from Figure 2.

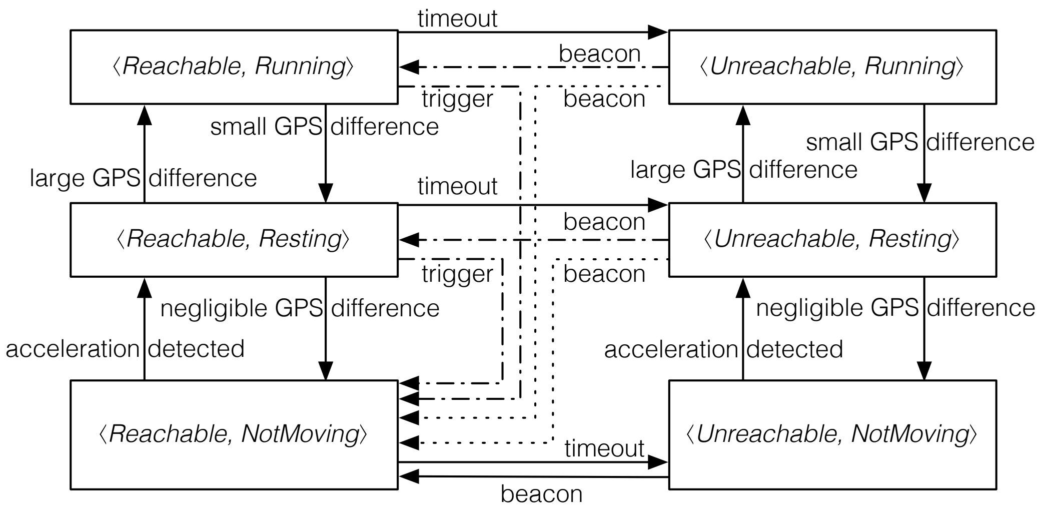


Fig. 14: FSM model obtained from a context-oriented design that only includes the *Base-station* and *Activity* groups from Figure 2.

for $\langle Healthy, Running\rangle$.[2] As a result, the FSM does *not* include the transition from $\langle Healthy, Running\rangle$ to $\langle Diseased, Running\rangle$, shown in Figure 13 with a dashed line.

Interestingly, the FSM in Figure 13 already reveals a potential issue. Based on the informal description of the application in the Introduction, it may appear that a situation where both context *Diseased* and *Running* are active may make little sense. It would, in fact, represent an animal that is very active when sick. However, Figure 13 shows that state $\langle Diseased, Running\rangle$ is still reachable, thus the evolution of active contexts in different groups may eventually lead the application to that situation.

[2]The $*$ notation is a wildcard to indicate a subset of the FSM states, regardless of the specific value of the tuple element.

Figure 14 shows an example based on a different slice of the context-oriented design of Figure 2, only including the *Base-station* and *Activity* groups. This time, the slice of the model includes a trigger $Reachable.T = NotMoving$ and no dependencies. In the FSM, the transitions from a state $\langle Unreachable, *\rangle$ to a state $\langle Reachable, *\rangle$ are subject to a trigger that "redirects" the transition to state $\langle Reachable, NotMoving\rangle$. The dotted lines in Figure 14 show the resulting transition in the FSM, whereas the dashed lines show the transitions ahead of the triggers and the triggers themselves.

The FSM in Figure 14 reveals another potential issue, caused by the presence of triggers. The transition from $\langle Unreachable, Running\rangle$ to $\langle Reachable, NotMoving\rangle$, in fact, changes a state $\langle *, Running\rangle$ to a state $\langle *, NotMoving\rangle$. In the original context-oriented design of Figure 2, however, the *Running* context in the *Activity* group has no outgoing transitions that points to *NotMoving*, that is, the trigger forces the contexts to evolve in a way the designer did not foresee. This may indicate a design flaw. We indicate this kind of occurrences as *unintended transitions*.

6. TOOL SUPPORT

We design and implement a complete tool-chain to support developers in designing, programming, and verifying adaptive WSN software using our approach. The complete tool-chain is publicly available, including a pre-configured virtual machine image to ease the installation process [Grevecom 2017].

Figure 15 illustrates the work-flow our tool-chain enables. The GRaphical Editor and VErifier for Context-oriented Models (GREVECOM), which we design and implement as an Eclipse plug-in, allows designers to graphically compose the context-oriented design as illustrated in Section 3. Based on this, GREVECOM allows one to automatically generate CONESC templates later completed with application-specific functionality. The resulting implementations are handed over to a CONESC translator and then to the nesC compiler to produce a deployment-ready binary.

Designers may also trigger a dedicated model generator that outputs a NuSMV model through the transformation of Section 5. This is input to NuSMV, along with predefined as well as developer-provided properties for running the actual verification. The NuSMV results are then parsed to express them using the same concepts of the initial context-oriented design. We choose NuSMV as it is a mature and robust model-checking tool featuring a multitude of success stories against complex problem instances [NuSMV 2017]. Nevertheless, we are not tied to NuSMV. As we are ultimately generating an FSM representation, we may as well employ different model checkers, depending on the nature of the problem [Clarke et al. 1999].

GREVECOM. Figure 16 shows a screen-shot of the GREVECOM editor. The main area (**A**) operates as a canvas. From a dedicated palette, designers drag and drop context groups and individual contexts (**B**) or even pre-canned patterns (**C**), as described

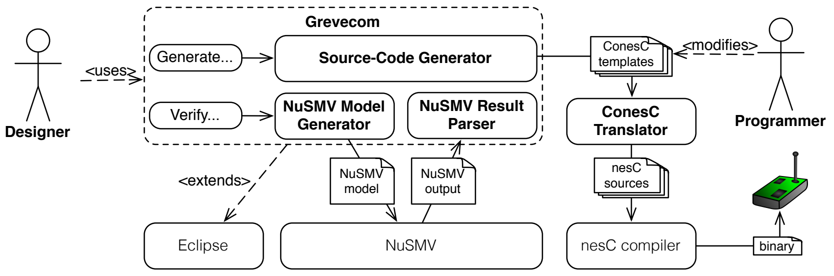
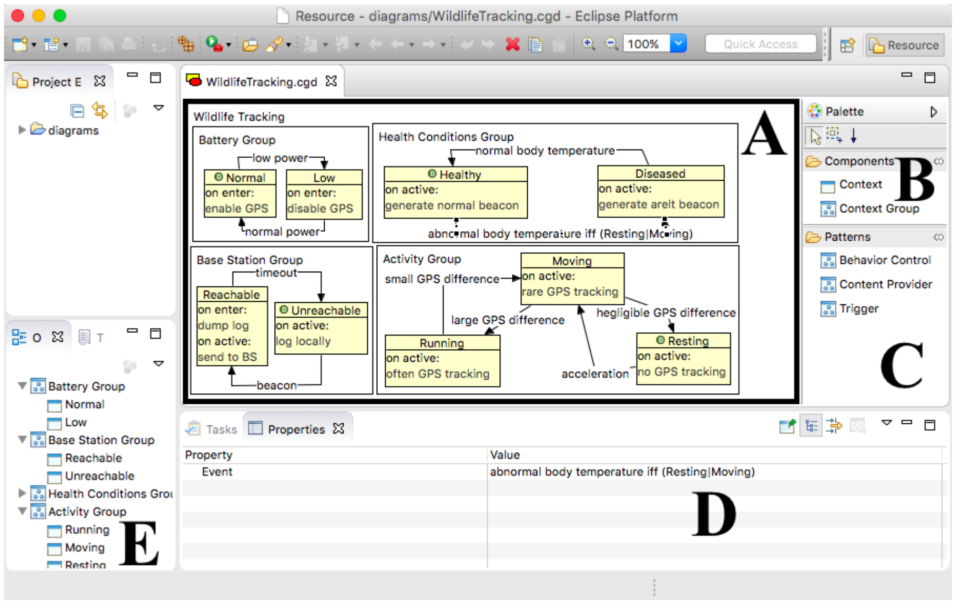


Fig. 15: Tool-chain work-flow. *The components in bold are those we developed.*



Property	Value
Event	abnormal body temperature iff (Resting|Moving)

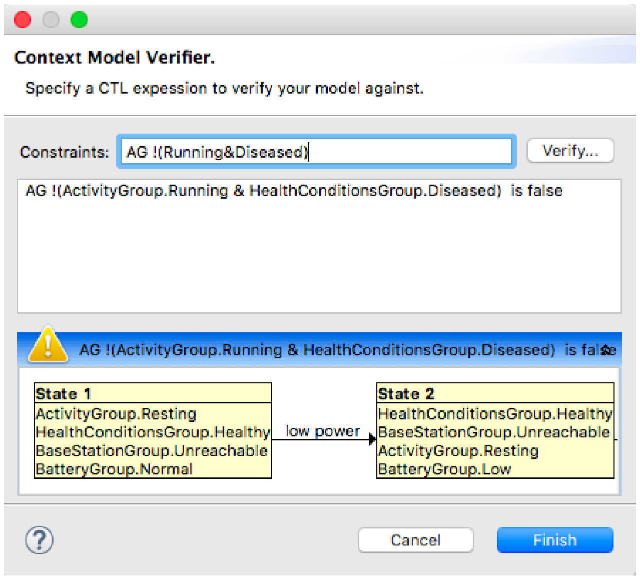
Fig. 16: GREVECOM editor.

in Section 3.2. Whenever doing so, a wizard pops up to guide the designer in expressing the key properties of the object; for example, in the case of individual contexts, a designer is asked for the name, actions on entering/exiting, a description of the processing when active, whether it is to be tagged as "default" or "error", and the possible triggers to other contexts. Individual contexts are linked by labeled transitions. Each label contains a representation of the environmental event triggering the transition and an optional dependency. The property tab (**D**) allows designers to change the properties of any object on the canvas. The hierarchy tree (**E**) helps navigate large models.

A menu command *Verify...* starts the automatic verification of the design, including the translation to a NuSMV model, the actual verification process, and the translation of the results back into the context-oriented design. A window such as Figure 17 pops up asking the designer to type a list of CTL constraints divided by a semicolon. In this example, a constraint *AG !(Running&Diseased)* verifies that the design of Figure 2 cannot ever find itself simultaneously in the *Running* and *Diseased* contexts. In addition to the designer-provided constraints, the NuSMV model generator inserts specifications to check the reachability of individual contexts and deadlocks. Any found counterexample is graphically represented as a sequence of activated contexts, as shown at the bottom of Figure 17. We investigate the efficiency of this process in Section 7.

CONESC translator. When the designer triggers the *Generate...* command, GREVECOM uses the context-oriented design to automatically generate CONESC templates to aid programmers in the implementation phase. The templates already include the CONESC code defining context groups, individual contexts with their properties, and nesC components possibly used in patterns. The labels describing context transitions and processing within individual contexts are translated into comments to guide programmers in filling the templates with application-specific functionality.

To obtain a deployment-ready binary, we develop a translator from CONESC to plain nesC, which allows us to rely on the compiler and platform support of the latter. Our translator performs two passes through the source code. First, it reads the main `Makefile` to scan the component graph. Based on this, it parses every input file to


Context Model Verifier.
Specify a CTL expession to verify your model against.

Constraints: AG !(Running&Diseased)

AG !(ActivityGroup.Running & HealthConditionsGroup.Diseased) is false

AG !(ActivityGroup.Running & HealthConditionsGroup.Diseased) is false

State 1
ActivityGroup.Resting
HealthConditionsGroup.Healthy
BaseStationGroup.Unreachable
BatteryGroup.Normal

low power

State 2
HealthConditionsGroup.Healthy
BaseStationGroup.Unreachable
ActivityGroup.Resting
BatteryGroup.Low

Fig. 17: GREVECOM verification wizard.

convert any CONESC constructs into plain nesC and to generate support functionality. The resulting sources can be compiled with the standard nesC tool-chain.

Specifically, the CONESC translator generates a custom nesC component for each context group that takes care of dynamically dispatching layered function calls to the implementation corresponding to the active context. This component is a part of an automatically-generated nesC configuration that exports the layered functions declared in the context group. Each individual context is translated into a nesC component with the necessary interfaces for wiring. CONESC constructs such as **activate** are also translated into standard function calls that determine the context to be considered for dispatching calls to layered functions.

Our translator is implemented using JavaCC [JavaCC 2016]. Three aspects are worth noticing. First, the generated code is human-readable and can be modified for fine-grained optimizations. Second, the resulting binary is completely hardware independent; since CONESC is translated into plain nesC, it enjoys the same compatibility as the original nesC tool-chain, allowing programmers to use CONESC with a variety of WSN platforms. Third, despite the apparent simplicity of the translation process, the semantics gap from CONESC to plain nesC is significant. To give an intuition on this aspect, in the benchmark applications we use in Section 7, the number of automatically-generated lines of nesC code is three times larger than the functionally equivalent CONESC sources. We discuss these aspects next.

7. EVALUATION

We consider three representative applications; one is the wildlife tracking application described in the Introduction, the other two are illustrated in Section 7.1. We design and implement the three applications using either the design concepts of Section 3 and programming support of Section 4, or no specific design concept and plain nesC. The two implementations are functionally equivalent, yet the latter approach arguably represents common practice in WSN software [Mottola and Picco 2011]. We consider that as a baseline for comparison.

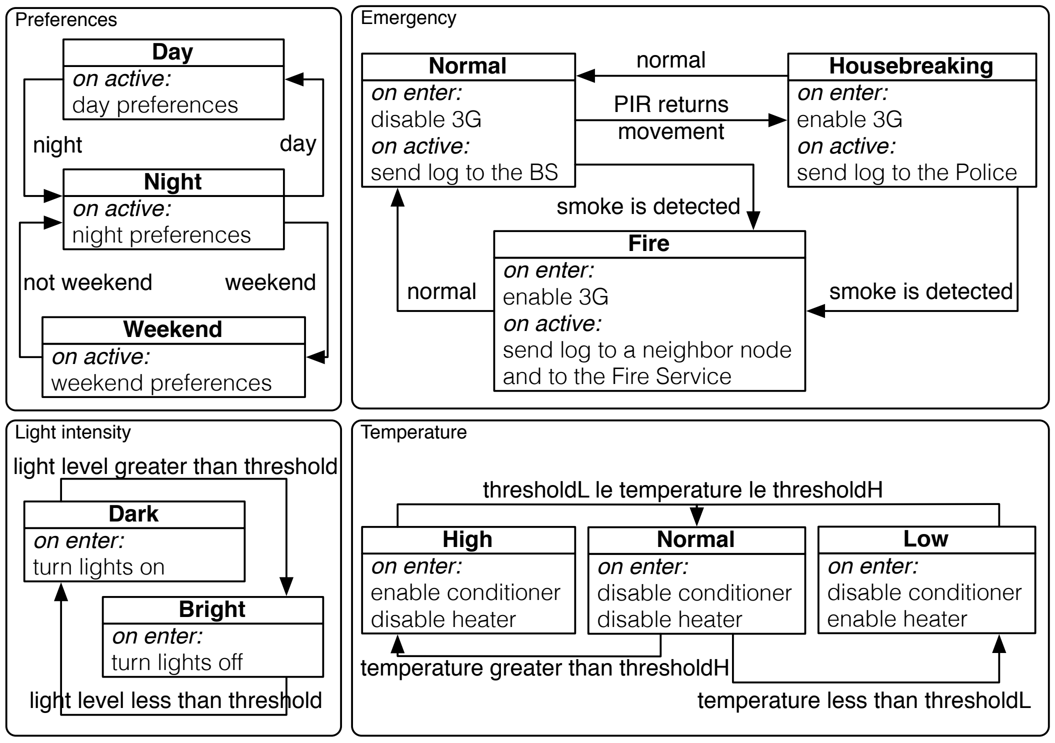


Fig. 18: Smart-home controller design.

We assess our approach along several dimensions. First, we analyze the impact of the design concepts and programming support on the actual implementations. This materializes in terms of the resulting *coupling* among components, which we investigate in Section 7.2, as the ease of changing the implementations against varying requirements, which we discuss in Section 7.3, and in the code complexity, which we study both qualitatively and quantitatively in Section 7.4. The benefits we demonstrate for our design and programming approaches bear a run-time cost, mainly in terms of MCU and memory overhead, which we measure in Section 7.5. Finally, we investigate the effectiveness of the automatic verification for context-oriented designs in terms of the time to *i)* generate the NuSMV model, discussed in Section 7.6, and *ii)* concretely run the verification, as reported in Section 7.7.

7.1. Applications

In addition to the wildlife tracking application shown in Figure 2, we consider two applications with different requirements. The diversity among the benchmarks we employ provides evidence of the generality of our approach.

Adaptiveness to different situations is one of the key requirements in smart-home applications [Mattern et al. 2010]. We consider a smart-home controller, whose design is shown in Figure 18, relying on environment 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 situation. The preferences are managed within the *Preferences* group, whose contexts provide different operating parameters depending on day/night patterns and day of

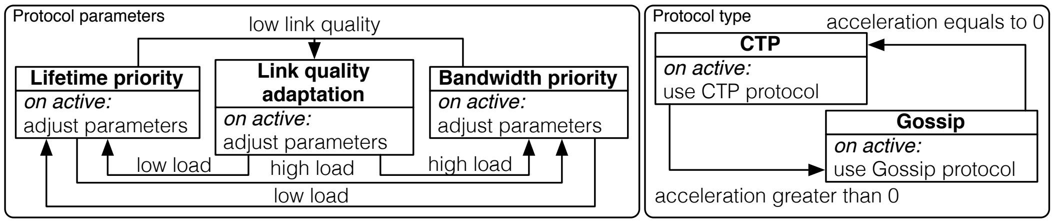


Fig. 19: Adaptive protocol stack design.

Table I: Coupling types among software components.

Type	Description
Content (tightest)	The internal working of two components depend on each other.
Common	Two or more components share some global state, for example, a variable.
External	Two or more components share a common data format.
Control	One component controls the flow of another.
Stamp	Two components share a common data format; each of them uses a different part.
Data	Two components share data through a typed interface, for example, a function call.
Message (loosest)	Two components share data through an untyped interface, for example, via messages.

the week. The context transitions within the *Light* and *Temperature* groups are driven by thresholds found in such a parameter set, compared against current temperature and light readings. The controller exploits image, fire, and smoke sensors to detect housebreaking and fire situations, as specified in the *Emergency* group.

Adaptive functionality is also required at system level, for example, when developing network stacks able to change the protocol logic depending on the situation. An example context-oriented design is shown in Figure 19. It realizes dynamic protocol switching in situations where a node may alternate periods of significant mobility with periods of static operation, as specified in the *Protocol type* group. Whenever a node remains static, it runs the Collection Tree Protocol (CTP) [Gnawali et al. 2009], which employs tree routing to funnel data to the destination. As soon as the on-board accelerometer detects a significant movement, the node switches to a route-less protocol, which allows the node to relay data efficiently in mobile settings [Ferrari et al. 2012]. In addition, the *Protocol parameters* group specifies three parameter sets, depending on whether lifetime or bandwidth is to be favored, and based on current link qualities.

Worth noticing is that the patterns we discussed in Section 3.2 emerge in these designs as well. For example, the *Preferences* group in Figure 18, in fact, operates as a *content provider* with respect to the *Light* and *Temperature* groups. The individual contexts in the *Preferences* group do not include any significant functionality, yet provide context-dependent parameters to other components. The same pattern is found in the *Protocol parameters* group of Figure 19. A *behavior control* pattern is found in both applications. In Figure 18, the functionality to handle emergency information changes between the individual contexts in the *Emergency* group, as well as the behavior of network API changes depending on the active protocol in the *Protocol type* group of Figure 19. Finally, the *trigger* pattern is found in multiple places; for example, in Figure 18 when operating the HVAC systems in the *Temperature* group or when enabling/disabling the 3G modem in the *Emergency* group.

7.2. Design and Programming $\rightarrow$ Coupling

The coupling among components generally determines the ease of maintenance of the software [Koopman 2010]. According to Stevens and Yourdon [1979], seven types of

Table II: Coupling comparison. *The context-oriented design and* CONESC *save most types of coupling that are present when using plain nesC.*

Application	Content	Common	External	Control	Data
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

coupling exist, summarized in Table I. The tighter is the coupling among these types, the more difficult is extending, evolving, and debugging the software. We manually inspect the CONESC and nesC implementations of the three benchmark applications to identify the types of coupling therein.

Results. Table II reports our findings. In general, the context-oriented design and CONESC foster increased decoupling among components compared to plain nesC.

For example, *Content* coupling is avoided in CONESC implementations, in that different behavioral variations of the same layered function are encapsulated in different contexts. Contrary, nesC programmers are forced to expose internal component information to bind command or events to different components depending on the situation. Similarly, nesC developers use global variables to switch between functionality; this is actually the case of variable **`base_station_reachable`** in Figure 1. This creates *Common* coupling, which developers avoid in CONESC implementations because the necessary functionality is automatically generated by our translator. *Control* coupling is avoided in CONESC implementations as well. This is a result of the ability to dynamically dispatching calls to layered functions transparently to the programmer; this functionality must be manually coded using nesC.

Data and *External* coupling are found in both CONESC and nesC implementations. Both ultimately extend C, which relies on typed interfaces, and different components need to use common data formats.

7.3. Design and Programming → Software Evolution

The need of evolving the software is common to many application domains. In WSN software, this need exacerbates as new requirements often emerge once domain experts gain increased understanding of the environment they study [Gaura et al. 2010]. Because of the complexity of designing and implementing WSN software, the question is how disruptive such changes may be.

To investigate this aspect, we assess the effort required to perform three example modifications in the CONESC and nesC implementations of the three benchmark applications. We extend the wildlife tracking application to the case where domain experts need to track the spread of a disease. To this end, a new type of beacon needs to be generated for an animal who was in contact with a diseased one, but shows no symptoms yet. In the smart-home application, we consider the case when a periodic run-time check of the controller execution is to be added. Should a potential failure be discovered, the controller needs to change its behavior. Finally, we modify the adaptive protocol stack by removing one of the parameter sets, which was found to be inefficient.

Results. Generating a new type of beacon in the wildlife tracking application requires modifying the design by adding a *Carrier* context in the *Health conditions* group of

Table III: Complexity comparison. *The context-oriented design and* ConesC *yield simpler implementations that are easier to debug and to reason about.*

Application	Average per-module Variables	Average per-module 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

Figure 2. This context corresponds to the generation of the new type of beacon, handed over to the network stack for the actual transmission according to the *content provider* pattern. Let apart the functionality to concretely gather the information to embed in the new type of beacon—equally required in ConesC and nesC—the modifications in the former only amount to 5 lines of code. To implement the same extension in nesC, besides the necessary code modifications, programmers need additional global states, further complicating the control flow.

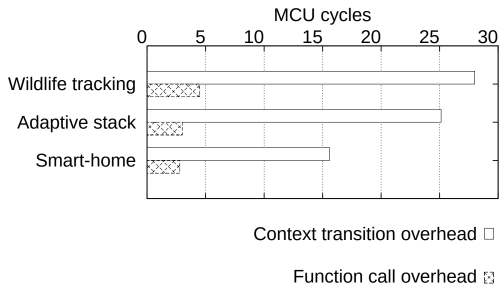
Extending the smart-home controller with a run-time check of the execution requires modifying its design by adding a new context group with two individual contexts: *Normal* or *Faulty*. This change in the design corresponds to about 40 lines of code in ConesC, besides the implementation of the new contexts. Using nesC, in addition to the individual functionality depending on the state of the controller, a new global variable is necessary to switch between these states, further increasing the coupling.

Finally, removing a context in the ConesC implementation of the adaptive protocol stack only requires modifying 3 lines of code, whereas in nesC developers must modify several lines of code scattered throughout different components. This is a paradigmatic benefit brought by the increased decoupling of ConesC implementations.

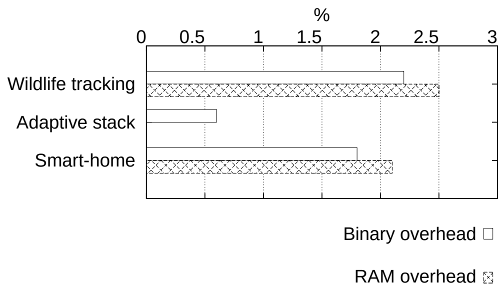
7.4. Programming → Complexity

The complexity of implementations bears an impact on the readability as well as the ease of debugging and understanding. We estimate this aspect by measuring the number of variable and function declarations in each component, which are generally considered as intuitive indicators of complexity [Koopman 2010]. As debatable it may be to measure the effectiveness of a programming abstraction [Mottola and Picco 2011], we also measure the number of lines of code when using ConesC or nesC.

Complexity is also a function of the number of states in which a program can find itself [Koopman 2010]. A state here is considered 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 of variables for every possible execution. We use SATABS [Clarke et al. 2005], a model-checking tool for C programs, to perform this analysis. The tool performs off-line verification against user-defined assertions. To do so, it searches through the program executions where a given assertion holds. As a by-product of this process, SATABS returns the number of states it explores in the program. With a specific configuration, we force SATABS to explore *all* possible program executions and thus to return the total number of distinct states in the program. We use SATABS on a per-function basis, implementing empty stubs wherever we cannot process the code with SATABS, for example, in the case of hardware drivers.



(a) MCU overhead.



(b) Memory overhead.

Fig. 20: MCU and memory overhead. *The additional resources necessary for* CONESC *are very limited.*

Results. Table III illustrates our results. CONESC shows a significant reduction in both declared functions and variables. This comes from the ability to dynamically bind function calls to a corresponding behavioral variation transparently to the caller. In nesC, achieving the same functionality requires to define a set of global variables to check what variation needs to be employed depending on the situation. As a result, the number of per-function states that programmers have to deal with almost halves when using CONESC, making implementations easier to understand.

The number of lines of code when using CONESC or nesC turns out to be roughly comparable. More interesting is the size of the plain nesC code output by our CONESC translator. As mentioned in Section 6, this is three times larger than the original CONESC code, on average. Besides indicating the extent of the semantics gap between CONESC and nesC, this observation also demonstrates that the few simple concepts we conceive for CONESC do capture a significant portion of processing.

7.5. Programming → MCU and Memory Overhead

The use of CONESC comes at the cost of run-time overhead. Compared to a hand-crafted implementation, for example, the code our translator automatically generates is likely less optimized in terms of memory occupation or processing time. The latter possibly affects energy consumption as well. To asses this aspect, we investigate the memory overhead when using CONESC as compared to nesC, as well as the MCU overhead for context transitions and calls to layered functions. We determine the former using the nesC and GNU-C tool-chains, whereas we measure the latter using the MSPSim cycle-accurate emulator [Eriksson et al. 2009]. MSPSim models the MCU of popular WSN devices such as the TMote Sky [Polastre et al. 2005].

Results. Figure 20 shows the results. The MCU overhead for layered function calls, shown in Figure 20a, varies from 2 to 5 MCU cycles depending on the application. This emerges because of the dynamic dispatching to the active context. In absolute terms, and thus also in terms of the corresponding energy consumption, the overhead is negligible: the simplest operation on a WSN node, such as turning an LED on or off, takes 8 MCU cycles. The MCU overhead for performing context transitions is slightly larger, but in the same order of magnitude. This is mainly caused by the checks performed when executing a transition, described in Section 4.4.

Figure 20b indicates that the memory overhead is 2.5% in the worst case. The complexity of the application, however, largely dictates the relative memory penalty. For example, the wildlife tracking application, being the most complex in terms of contexts, context changes, and data processing, shows the highest memory overhead. In contrast, the memory overhead for the adaptive protocol stack is negligible. In this case, interestingly, the CONESC translator generates almost the same set of variables that a nesC programmer would define by hand.

7.6. Verification → NuSMV Model Generation

The automatic verification technique we conceive involves two steps: *i)* generating the semantically equivalent FSM as explained in Section 5 along with its encoding in an NuSMV model, and *ii)* running the actual verification using the latter as input. We evaluate the two separately.

To measure the time to generate the NuSMV model, we place ourselves in the worst situation and employ synthetic designs conceived to yield the highest running times. A procedural implementation to generate the NuSMV model according to the specification in Section 5 is quadratic in both the number of context groups $N_C = |G|$ and in the number of individual contexts $N_{CG} = |g.C|$ in a group $g \in G$. Because additional processing is required for each transition in a group, the highest running times correspond to a design where all groups include the largest number of individual contexts and each of these is bound with a transition to every other context in the group.

We perform measurements with $N_C \in [2, \ldots, 10]$ and $N_{CG} \in [1, \ldots, 10]$. Note how these designs are, in fact, quite unrealistic. The representative applications in Section 7.1, for example, include fewer context groups and fewer individual contexts than most of these configurations. Moreover, it is rarely the case that so many transitions are defined for each context. Typical WSN devices are severely resource-constrained; the context-oriented designs would rarely reach these degrees of complexity.

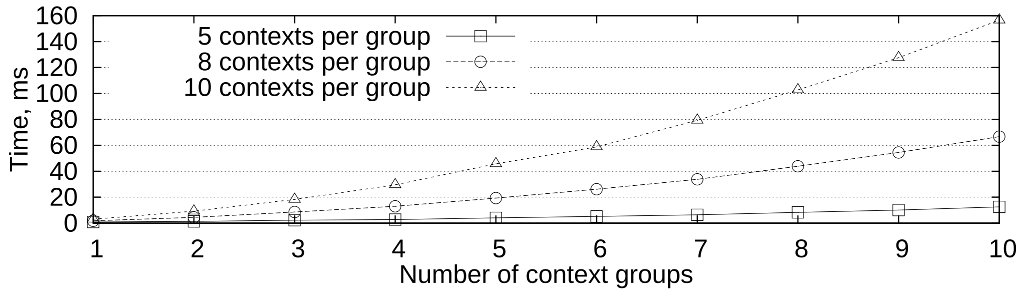
Our implementation of the transformation procedure is written in Java and runs on an Intel Core2Duo machine at 1.4 GHz with 4 GBytes RAM. We measure the CPU time using the standard Java library *ThreadMXBean*. To account for random effects that may alter a measurement, we repeat the measurements multiple times, until the standard deviation across different repetitions falls below 5%.

Results. Figure 21 depicts the trends at stake against varying either N_C or N_{CG}. The curves confirm the quadratic trends. Most importantly, however, the absolute values are extremely limited. Even in an unrealistic configuration with $N_C = N_{CG} = 10$, the time to generate the NuSMV model rests well below 200 ms.

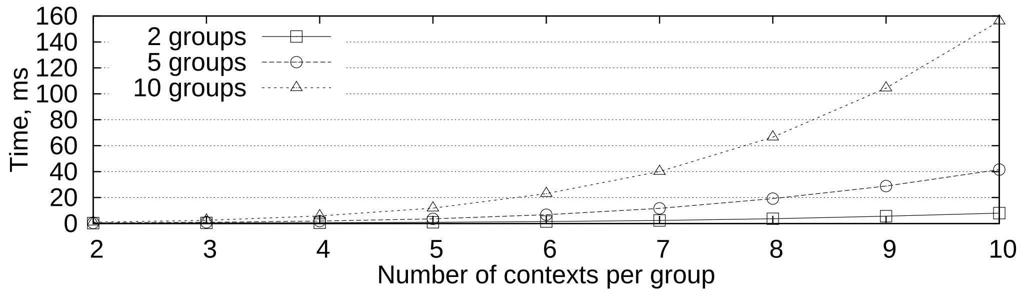
7.7. Verification → Running Time

We measure the time taken by NuSMV to verify the semantically equivalent FSM. To verify the input model efficiently, NuSMV converts the FSM into an equivalent binary decision diagram (BDD) [Clarke et al. 1999]. The counter-example possibly indicating a property violation is found on the BDD representation of the model, yet it might not be useful to the user until it is translated back in terms of transitions in the original FSM model. This process, called “generation of counterexample traces” is optionally disabled through a command-line parameter in NuSMV [NuSMV 2017], as users may not necessarily be interested in detailed information on a property violation.

We consider both the context-oriented designs of the representative applications in Section 7.1 and the synthetic ones of Section 7.6. For the former, we consider the cases where the model is correct and where it contains flaws such as deadlocks, unintended transitions as defined in Section 5, and violations of developer-provided CTL properties. To trigger the latter, we artificially introduce specific flaws in the context-oriented design; for example, to insert a deadlock, we add two mutually-exclusive dependencies in the original design. In these experiments, NuSMV is configured to always generate counter-example traces whenever a property violation is found. In contrast, for the synthetic designs of Section 7.6, we limit ourselves to the case of a correct model or



(a)



(b)

Fig. 21: Time for generating the NuSMV model. *The curves confirm the quadratic trend in the number of context groups and individual contexts; the absolute values remain below 200 ms.*

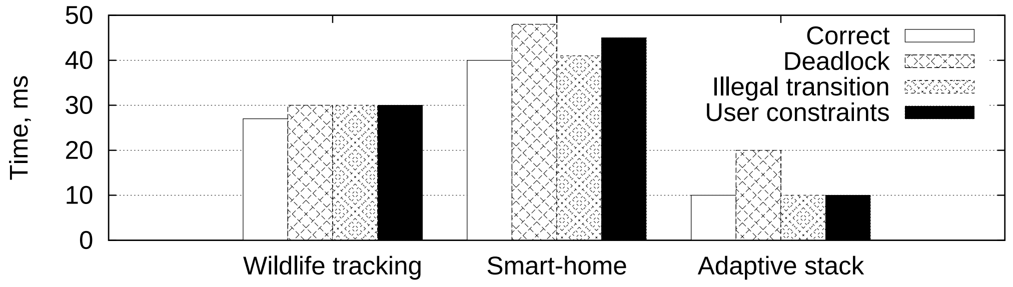


Fig. 22: NuSMV running time with the FSM equivalent to the context-oriented designs of Section 7.1. *The absolute values remain below 50 ms.*

a model with a deadlock. In addition, we check the impact of generating a counter-example trace whenever a property violation is found. We fix $N_C = 5$ and vary N_{CG} to investigate how NuSMV scales with the size of the input models.

We measure the CPU time taken by NuSMV with UNIX command `time`. The rest of the setup, including the hardware, is the same as in Section 7.6.

Results. Figure 22 shows the NuSMV running time with the FSM equivalent to the context-oriented designs of the applications of Section 7.1. The absolute values are all very limited, and lower than 50 ms in all cases. Even by considering these values in addition to the times to generate the equivalent FSM from the original context-

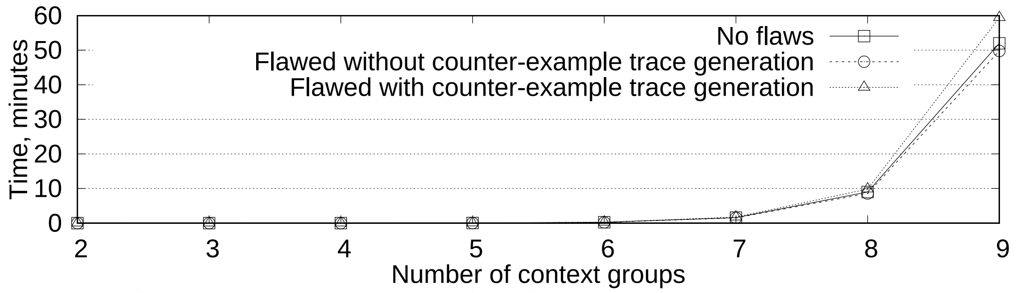


Fig. 23: NuSMV running times with the FSM equivalent to the synthetic designs of Section 7.6. *The running times remain practical up to $N_{CG} = 8$, and are always below one minute for $N_{CG} \leq 7$.*

oriented design, discussed in Section 7.6, the total running times remain practical. There is no significant difference in the running times depending on the type of flaw, or on the type of CTL property that fails.

Figure 23 shows the results of the experiments with the synthetic models of Section 7.6. Because of how we generate the equivalent FSM, adding one context group in the original design yields an exponential increase in the size of the state space NuSMV needs to explore, which is reflected in the trends of Figure 23. For realistic configurations, such as those with $N_{CG} \leq 7$ and $N_C = 5$, the running times are below one minute. They start becoming impractical only for $N_{CG} = 9$.

As seen already in Figure 22, a flawed model requires slightly increased running times also in Figure 23. A comparison between the two plots provides evidence that this overhead comes from the generation of the counter-example trace. When this functionality is disabled in NuSMV as explained above, the running times are slightly below those of a correct model. Reason for this is that NuSMV verifies the model incrementally, and stops as soon as a property violation is found.

8. CONCLUSION

We presented design concepts, programming constructs, and automatic verification techniques to support developers in realizing adaptive WSN software. Our design concepts help factor out the adaptation necessary to deal with independent environment dimensions and understand their relations. These concepts map directly to the constructs of CONESC, our context-oriented extension of nesC, which greatly simplifies the implementation of adaptive WSN software. The automatic verification techniques we conceive complement the design and programming support by providing a means to check the correctness of the design prior to the actual deployment. The three contributions are tied together by dedicated tool support, which supports developers from design to implementation and verification.

Our evaluation, based on three diverse representative applications, indicates that our design concepts and CONESC result in higher-quality implementations that are simpler to reason about, structurally more decoupled, and easier to modify compared to functionally-equivalent implementations written in plain nesC. On the other hand, the run-time overhead for CONESC implementations, again compared to plain nesC, turns out negligible. Finally, we quantitatively demonstrate that our verification techniques work efficiently on practical instances, returning results in a matter of seconds.

Acknowledgments. This work was partly supported by projects "Zero-energy Buildings in Smart Urban Districts" (EEB), "ICT Solutions to Support Logistics and Transport Processes" (ITS), and "Smart Living Technologies" (SHELL) of the Italian Ministry for University and Research.

REFERENCES

G. Abowd, A. Dey, P. Brown, N. Davies, M. Smith, and P. Steggles. 1999. Towards a better understanding of context and context-awareness. In *International Symposium on Handheld and Ubiquitous Computing*.

T. Ball and S. K. Rajamani. 2002. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of POPL*.

J. E. Bardram. 2005. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-aware Applications. In *Proc. of PERVASIVE*.

D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. 2007. The Software Model Checker Blast: Applications to Software Engineering. *Int. J. Softw. Tools Technol. Transf.* (2007).

N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Trans. on Wireless Sensor Networks (TOSN)* (2016).

T. Bourdenas, D. Wood, P. Zerfos, F. Bergamaschi, and M. Sloman. 2011. Self-adaptive routing in multi-hop sensor networks. In *Proc. of CNSM*.

D. Bucur and M. Kwiatkowska. 2009. Bug-Free Sensors: The Automatic Verification of Context-Aware TinyOS Applications. In *Proc. of AmI*.

B. Cheng and others. 2009. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. Springer.

E. Clarke, O. Grumberg, and D. Peled. 1999. *Model checking*. MIT press.

E. Clarke, D. Kroening, and F. Lerda. 2004. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 168–176.

E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. 2005. SATABS: SAT-based Predicate Abstraction for ANSI-C. In *Proc. of TACAS*.

E. M. Clarke, E. A. Emerson, and A. P. Sistla. 1986. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Trans. Program. Lang. Syst.* (1986).

J. Coutaz, J. Crowley, S. Dobson, and D. Garlan. 2005. Context is Key. *Commun. ACM* 48, 3 (2005).

A. Dey. 2001. Understanding and using context. *Personal and ubiquitous computing* 5, 1 (2001).

J.-P. Diguet, Y. Eustache, and G. Gogniat. 2011. Closed-loop-based self-adaptive hardware/software-embedded systems: Design methodology and smart cam case study. *ACM Trans. on Embedded Computing Systems (TECS)* (2011).

A. Dunkels, B. Gronvall, and T. Voigt. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*.

J. Eriksson, F. Oesterlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marroon. 2009. COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In *Proc. of Simutools*.

F. Ferrari, M. Zimmerling, L. Mottola, and L. Thiele. 2012. Low-Power Wireless Bus. In *Proc. of the ACM Conference on Embedded Network Sensor Systems (SenSys)*.

R. Filman, T. Elrad, S. Clarke, and M.ehmet Akşit. 2004. *Aspect-oriented software development*. Addison-Wesley Professional.

N. Finne, J. Eriksson, N. Tsiftes, A. Dunkels, and T. Voigt. 2010. Improving Sensornet Performance by Separating System Configuration from System Logic. In *Proc. of EWSN*.

F. Fleurey, B. Morin, and A. Solberg. 2011. A Model-driven Approach to Develop Adaptive Firmwares. In *Proc. of the 6th SEAMS*.

H. Fotouhi, M. Zuniga, M. Alves, A. Koubaa, and P. Marron. 2012. Smart-HOP: A Reliable Handoff Mechanism for Mobile Wireless Sensor Networks. In *Proc. of EWSN*.

E. Gaura, L. Girod, J. Brusey, M. Allen, and G. Challen. 2010. *Wireless sensor networks: Deployments and design frameworks*. Springer Science & Business Media.

D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. 2003. nesC language: A holistic approach to networked embedded systems. In *Proc. of PLDI*.

O. Gnawali, R Fonseca, K. Jamieson, D. Moss, and P. Levis. 2009. Collection Tree Protocol. In *Proc. of SENSYS*.

Grevecom 2017. GrEVeCOM: Software Adaptation in Wireless Sensor Networks. (2017). `bitbucket.org/neslabpolimi/conesc/wiki/`.

F. Gustafsson. 2010. *Statistical sensor fusion*. Springer.

R. Hirschfeld, P. Costanza, and O. Nierstrasz. 2008. Context-oriented Programming. *Journal of Object Technology* (2008).

K. Iwanicki, P. Horban, P. Glazar, and K. Strzelecki. 2014. Bringing Modern Unit Testing Techniques to Sensornets. *ACM Trans. Sen. Netw.* (2014).

M. Jackson. 1995. The World and the Machine. In *Proc. of ICSE*.

JavaCC 2016. JavaCC - The Java Compiler Compiler. (2016). `javacc.java.net`.

T. Kamina, T. Aotani, and H. Masuhara. 2011. EventCJ: A Context-oriented Programming Language with Declarative Event-based Context Transition. In *Proc. of AOSD*.

S. Kang, J. Kee, H. Jang, Y. Lee, S. Park, and J. Song. 2008. SeeMon: Scalable and Energy-efficient Context Monitoring Framework for Sensor-rich Mobile Environments. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services (MOBISYS)*.

R. Keays and A. Rakotonirainy. 2003. Context-oriented Programming. In *Proc. of MobiDe*.

N. Kern, B. Schiele, and A. Schmidt. 2003. Multi-sensor activity context detection for wearable computing. In *European Symposium on Ambient Intelligence*.

G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. *European Conference on Object-oriented Programming (ECOOP)* (1997).

J. Ko, C. Lu, M. B. Srivastava, J. A. Stankovic, A. Terzis, and M. Welsh. 2010. Wireless sensor networks for healthcare. *Proceedings of IEEE* (2010).

P. Koopman. 2010. *Better Embedded System Software*. Carnagie Mellon Press.

P. Li and J. Regehr. 2010. T-check: Bug Finding for Sensor Networks. In *Proc. of IPSN*.

C. Magerkurth, A. D. Cheok, R. L. Mandryk, and T. Nilsen. 2005. Pervasive Games: Bringing Computer Entertainment Back to the Real World. *Comput. Entertain.* 3, 3 (2005).

G. Mainland, G. Morrisett, and M. Welsh. 2008. Flask: Staged Functional Programming for Sensor Networks. In *Proc. of ICFP*.

F. Mattern, T. Staake, and M. Weiss. 2010. ICT for green: how computers can help us to conserve energy. In *Proc. of ICEECN*.

L. Mottola, A. L. Murphy, and G. P. Picco. 2006. Pervasive Games in a Mote-enabled Virtual World Using Tuple Space Middleware. In *ACM NETGAMES*.

L. Mottola and G.P. Picco. 2011. Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Comp. Surveys* (2011).

L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi. 2010. Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In *Proc. 2010 ICSE/SENSEA*.

R. Newton, G. Morrisett, and M. Welsh. 2007. The Regiment Macroprogramming System. In *Proc. of IPSN*.

NuSMV 2017. NuSMV: A New Symbolic Model Checker. (2017). `nusmv.fbk.eu`.

R. Olfati-Saber. 2007. Distributed Kalman filtering for sensor networks. In *IEEE Conference on Decision and Control*. IEEE.

B. Pásztor, L. Mottola, C. Mascolo, G. P. Picco, S. Ellwood, and D. Macdonald. 2010. Selective Reprogramming of Mobile Sensor Networks Through Social Community Detection. In *Proc. of EWSN*.

G. P. Picco. 2010. Software Engineering and Wireless Sensor Networks: Happy Marriage or Consensual Divorce?. In *Proc. of FSE/SDP FOSER*.

J. Polastre, R. Szewczyk, and D. Culler. 2005. Telos: enabling ultra-low power wireless research. In *Proc. of IPSN*.

K. Romer and Junyan Ma. 2009. PDA: Passive distributed assertions for sensor networks. In *Proc. of IPSN*.

G. Salvaneschi, C. Ghezzi, and M. Pradella. 2012. Context-oriented Programming: A Software Engineering Perspective. *J. Syst. Softw.* (2012).

G. Salvaneschi, C. Ghezzi, and M. Pradella. 2013. Towards language-level support for self-adaptive software. *ACM Trans. Auton. Adapt. Syst.* (2013).

R. Sasnauskas, O. Landsiedel, M. Hamad Alizai, C. Weise, S. Kowalewski, and K. Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *Proc. of IPSN*.

B. Schilit, N. Adams, and R. Want. 1994. Context-aware computing applications. In *First Workshop on Mobile Computing Systems and Applications (WMCSA)*.

A. Schmidt, M. Beigl, and H.-W. Gellersen. 1999. There is more to context than location. *Computers & Graphics* 23, 6 (1999).

S. Sehic, F. Li, and S. Dustdar. 2011. COPAL-ML: A Macro Language for Rapid Development of Context-aware Applications in Wireless Sensor Networks. In *Proc. of SESENA*.

W. Stevens and E. N. Yourdon. 1979. Classics in Software Engineering. Chapter Structured Design.

L. Subramanian and R.H. Katz. 2000. An architecture for building self-configurable systems. In *Proc. of MobiHOC*.

TinyOS 2016. TinyOS 2.1.2. (2016). `www.tinyos.net`.

E. Visser. 2002. Meta-programming with concrete object syntax. In *International Conference on Generative Programming and Component Engineering*.

A. D. Wood, J. A. Stankovic, G. Virone, L. Selavo, Z. He, Q. Cao, T. Doan, Y. Wu, L. Fang, and R. Stoleru. 2008. Context-aware wireless sensor networks for assisted living and residential monitoring. *Network, IEEE* (2008).

H. Wu, M. Siegel, R. Stiefelhagen, and J. Yang. 2002. Sensor fusion using Dempster-Shafer theory [for context-aware HCI]. In *IEEE Instrumentation and Measurement Technology Conference*.

M. Zimmerling, F. Ferrari, L. Mottola, T. Voigt, and L. Thiele. 2012. pTunes: Runtime parameter adaptation for low-power MAC protocols. In *Proc. of IPSN*.