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. While significant literature exists on the necessary adaptation logic, developers are left without proper support in materializing such a logic in a running system. 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 construct is negligible, and our verification techniques return results in a matter of seconds.

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 application. 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 allows the nodes to discover each other whenever they are within communication range, using a simple form of periodic radio beaconing. In fact, the radio doubles as a proximity sensor. A node logs the radio contacts to track an animal’s encounters with other animals, which allows biologists to study 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].

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 decide to 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 are commonly found in diverse WSN scenarios, including intelligent homes and buildings [Mattern et al. 2010], smart health-care [Jeonggil 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 and dependable 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, hampering the analysis by the domain experts.

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 and others 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, adaptation logic exist aplenty in the literature, yet proper developer support to implement these 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 18 to 23. This lies in the the same module as the adaptive processing itself from line 6 to 16. 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 7 and 10, 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, to maintain, and to 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
developers often deploy systems with little confidence in their correctness [Sasnauskas
et al. 2010; Iwanicki et al. 2014]. In WSNs, nevertheless, run-time errors are very
difficult to track and to recover from [Romer and Ma 2009].

Contribution. We tackle the problem with three interrelated contributions:

— We conceive dedicated design concepts, described in Section 2, to support develop-
ers in the early phases when identifying the possible system’s evolutions. In our
work, these emerge as a result of determining the different situations the WSN soft-
ware may find itself in, what environmental dimensions are responsible for these
situations, and the relations between different adaptation decisions.

— We extend nesC with notions of Context-oriented Programming (COP) [Hirschfeld
et al. 2008]. Section 3 describes the resulting language, called CONESC, which ame-
liorates the coupling between functionality, rendering implementations easier to
understand and to maintain. The design concepts of Section 2 map to the program-
ing constructs we introduce, easing the transition from design to implementation.

— We conceive automatic verification techniques to check the correctness of an ap-
plication’s design against the possible environment evolutions, as we describe in
Section 4. Our techniques operate before deployment, thus requiring reduced ef-
fort than most exiting approaches [Iwanicki et al. 2014; Romer and Ma 2009], and
quickly return counterexamples expressed with the same design concepts of Sec-
tion 2, facilitating the identification of issues.

The three contributions are tied together by dedicated tool support. Section 5 de-
scribes GREVECOM, a visual tool that allows developers to encode their designs using
the concepts in Section 2, and use this encoding to generate CONESC code templates
and automatically verify the designs using the techniques in Section 4. In the same
section, we also describe the compiler support we implement to automatically trans-
literate CONESC sources into plain nesC, which allows one to employ the latter tool-chain
to obtain the binary for deployment on WSN devices.

Our evaluation, described in Section 6, 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 evi-
dence of its practical use.

We conclude our paper by surveying efforts close to ours in Section 7, and with brief
concluding remarks in Section 8.

2. DESIGN

We illustrate design concepts to support developers in identifying the different situa-
tions that WSN software 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.

2.1. Concepts

We introduce two key concepts: i) individual contexts, and ii) context groups. A context
represents an individual situation the software may encounter. Whenever that situa-
tion occurs, the software changes its functioning accordingly, implementing an approp-
riate adaptation decision. For example, the reachability of the base-station based on
the physical location of a device represents an individual context coupled to a corre-
sponding 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 the physical location of a device.

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 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 situation the system finds itself in. 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 sufficiently expressive to cater for a significant fraction of adaptive WSN applications, as we demonstrate in Section 6. At the same time, they are largely decoupled from a concrete programming language, enabling their implementation in different WSN languages.

2.2. Patterns
Based on experience, we observe particular emerging patterns that provide structured ways to address specific types of adaptive functionality. These patterns, 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 depending on node’s mobility. The high-level packet relay functionality is expressed with a similar design, as we show in Section 6.

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 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 6 employs the same pattern to manage user preferences depending on time of the day.

This pattern’s 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 6 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 standalone, the context 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 6 we notice a similar pattern when dimming 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.

3. 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 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 3.1. Next, Section 3.2 illustrates the main programming constructs of our COP extension to nesC, called CONESC. The design concepts of Section 2 map directly to these constructs, facilitating the transition from design to implementation. Section 3.3 describes the
Fig. 6: Base-station context group in CONES C.

3.1. 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 component 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 et al. 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 and 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.

3.2. Context Groups and Individual Contexts

We map the notion of context group in Section 2 to an extended form of nesC configuration. The interface of a context group is used to declare the prototype of layered functions. Their behavioral variations are expressed by individual contexts in the group, whose definition extends that of nesC components.

Figure 6 shows the CONES C implementation of the Base-station group of Figure 2. The layered function report is declared in line 2 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 4 of Figure 6 using the contexts keyword. Of the three contexts in the example, the Unreachable one is defined as is default in line 5, that is, the context is active at start-up. The is error keyword in line 6 indicates an error context, whose semantics we illustrate next. Including an error context is 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 9 of Figure 7 deposits the message in the local data store; otherwise, if context Reachable is active, function report in line 15 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 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 2, 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 9 of Figure 8.
software adaptation in wireless sensor networks

module User {
  uses context group BaseStationG;
  implementation {
    event void Timer.fired() {
      call BaseStationG.report(msg);
    }
    event void BaseStationG.contextChanged(context_t con) {
      if(con == BaseStationG.Reachable) // DO SOMETHING...
    }
  }
}

Fig. 9: User component.

module BaseStationContextManager {
  uses context group BaseStationG;
  implementation {
    event msg_t Beacon.receive(msg_t msg) {
      activate BaseStationG.Reachable;
      call BSReset.stop();
      call BSReset.startOneShot(TIMEOUT);
      event void BSReset.fired() {
        activate BaseStationG.Unreachable;
      }
    }
  }
}

Fig. 10: Base-station context controller.

The event is automatically triggered when entering the context. The deactivated event, shown in line 10 of Figure 8, is dual.

3.3. Execution

Figure 9 shows an example component that relies on the functionality implemented by the layered function report. In line 5, when a timer fires, the call occurs without explicitly referencing either of the individual contexts that provide a concrete implementation of report. The binding corresponding to the active context happens dynamically. This is possible as our CONESC translator, described in Section 5, automatically generates the code that implements the necessary dynamic dispatching.

Figure 10 also shows the CONESC code that implements the necessary 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 9 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 9.

Notice how, in fact, Figures 6 to 10 implement the behavior control pattern, described in Section 2.2. Specifically, the nesC component of Figure 9 relies on a functionality whose behavior is determined by the context controller of Figure 10. The key benefit is that context-dependent functionality, as well as the logic driving context detection and activation, are completely decoupled and implemented in different software modules. We demonstrate in Section 6 that this renders implementations easier to understand, to debug, and to maintain.

3.4. 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`. Programmer may implement the corresponding event handler, as exemplified in line 7 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 2 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 transmissions, as described in Section 2.1. 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 2 on 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 11 of Figure 8. Should the check fail, the previous context remains active.
Finally, the design concepts illustrated in Section 2 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 in the Activity group. CONESC allows programmers to express this processing by using the triggers keyword, as shown on line 6 of Fig. 8, followed by the fully-qualified name of the other context that must be automatically activated when entering.

4. 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 little to no verification before deployment [Picco 2010].

Based on the concepts illustrated in Section 2, 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 reveal as unreachable or deadlock situations that may block the software in a specific configuration of active contexts. In addition, our techniques supports the verification of developer-provided 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 2. 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 5, 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.

Transformation. We choose to transform a context-oriented design defined according to Section 2 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.

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, Battery.C = \{Normal, Low\}. Each context \( c \) has a set \( O \) of outgoing transitions represented as tuples \( (o, e) \), where \( o \) is the target context and \( e \) is the label for that transition. In Figure 2, for example, Resting.O = \{ (Running, large GPS difference), (NotMoving, negligible GPS difference) \}. Triggers between contexts are defined as an attribute \( T \) of an individual context \( c \); for example, Reachable.T = 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 \( (c_1, c_2, \tau) \), where \( c_1 \) and \( c_2 \) are the originator and target contexts of a transition, respectively, and \( \tau \) is the context whose activation is necessary for this transition to be taken. Based on Figure 2, for example, both (Healthy, Diseased, Resting) and (Healthy, Diseased, NotMoving) belong to \( D \).
Fig. 13: FSM model obtained from a context-oriented design that only includes the groups Health conditions and Activity from Figure 2.

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 \( (c_1, c_2, \ldots, c_n) \), \( n = |G| \), 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 thus obtained by considering a single individual context for every group. A transition \( t \in T_{FSM} \) is a tuple \( (s_1, s_2, e) \) 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 exactly one individual context \( c' \in s_1 \) that changes to \( c'' \in s_2 \), that is, a transition in the FSM maps to a single transition in the original context-oriented design. A transition where \( c' \in s_1 \) changes to \( c'' \in s_2 \) 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, \( (c'', e) \in c'.O \) and \( c''.T = \bot \). Label \( e \) in transition \( t \) is the label of \( (c'', e) \in c'.O \).
2. For an individual context \( c' \) in \( s_1 \), a transition in the original context-oriented design exists in \( c'.O \) to an individual context \( c_i \) whose trigger points to \( c'' \), that is, \( (c_i, e) \in c'.O \) and \( c_i.T = c''.T \). Label \( e \) in transition \( t \) is the label of \( (c_i, e) \in c'.O \).
3. If a dependency \( (c', c'', \tau) \in \Delta \) exists, then \( \tau \) 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.

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 (3) 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, \( \Delta = \{(\text{Healthy, Diseased, Resting}), (\text{Healthy, Diseased, NotMoving})\} \). In the resulting FSM, the existing dependencies are satisfied for all states \( \langle \text{Healthy}, \ast \rangle \) but for \( \langle \text{Healthy}, \text{Running} \rangle \). As a result, the FSM does not include the transition from \( \langle \text{Healthy}, \text{Running} \rangle \) to \( \langle \text{Diseased}, \text{Running} \rangle \), 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 situ-
Software Adaptation in Wireless Sensor Networks

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

The FSM in Figure 14 reveals another potential issue, caused by the presence of triggers. The transition from \((Unreachable, Running)\) to \((Reachable, NotMoving)\), in fact, changes a state \((*, Running)\) to a state \((*, NotMoving)\). 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.

5. TOOL SUPPORT

We design and implement a complete tool-chain to support developers in designing, programming, and verifying adaptive WSN software using our approach.

Figure 15 illustrates the work-flow our tool-chain enables. The GRaphical Editor and VErifier for Context-oriented Models (GREVCOM), which we design and implement as an Eclipse plug-in, allows designers to graphically compose the context-oriented design as illustrated in Section 2. Based on this, GREVCOM 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 can also trigger a ded-
icated model generator that outputs a NuSMV model through the transformation of Section 4. 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.

**GREVECOM.** Figure 16 shows a screen-shot of the GREVECOM editor. The main area (A) operates as a canvas. From a dedicated palette, designers can drag and drop context groups and individual contexts (B) or even pre-canned patterns (C), described in Section 2.2. 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 of 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.

![Diagram](image1)

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

![Diagram](image2)

**Fig. 16:** GREVECOM editor.
A menu command Verify... is available to start 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 \( AG \lnot(Running \&\& Diseased) \) verifies that the application based on the design of Figure 2 cannot ever find itself simultaneously in the Running and Diseased contexts. Independently of the designer-provided constraints, the NuSMV model generator always 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 6.

**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 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 calls to layered functions 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 by a programmer for fine-grained optimization. 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 programmer to use CONESC with a great 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 about this aspect, in the benchmark applications we use in Section 6, the number of automatically-generated lines of nesC code is three times larger than the functionally equivalent CONESC sources. We discuss these aspects next.

6. EVALUATION
We consider three representative applications; one is the wildlife tracking application in the Introduction, the other two are illustrated in Section 6.1. We design and implement the three applications using either the design concepts of Section 2 and programming support of Section 3, or no specific design concept and plain nesC. The two implementations are functionality equivalent, yet the latter approach arguably represents common practice in WSN software [Mottola and Picco 2011]. We consider that as a baseline for comparison.

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 6.2, as the ease of changing the implementations against varying requirements, which we discuss in Section 6.3, and in the code complexity, which we study both qualitatively and quantitatively in Section 6.4. 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 6.5. 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 6.6, and ii) concretely run the verification, as reported in Section 6.7.

6.1. Applications
In addition to the wildlife tracking application shown in Figure 2, we consider two applications with different requirements. The resulting 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, that relies 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 the week. The context transitions within the Light and Temperature groups are driven by thresholds found in such parameter set, compared against current temperature and light readings. 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 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 alternative periods of significant mobility with periods
Software Adaptation in Wireless Sensor Networks

Fig. 18: Smart-home controller design.

Fig. 19: Adaptive protocol stack design.

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 gossip protocol, which allows the node to relay data opportunistically [Fotouhi 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 2.2 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 also found
Table I: Coupling types among software components.

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

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

<table>
<thead>
<tr>
<th>Application</th>
<th>Content</th>
<th>Common</th>
<th>External</th>
<th>Control</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wildlife tracking – nesC</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Wildlife tracking – ConesC</td>
<td>–</td>
<td>–</td>
<td>yes</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>Smart-home – nesC</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Smart-home – ConesC</td>
<td>–</td>
<td>–</td>
<td>yes</td>
<td>–</td>
<td>yes</td>
</tr>
<tr>
<td>Adaptive stack – nesC</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Adaptive stack – ConesC</td>
<td>–</td>
<td>–</td>
<td>yes</td>
<td>–</td>
<td>yes</td>
</tr>
</tbody>
</table>

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.

6.2. Design and Programming → Coupling

The coupling among components generally determines the ease of maintenance of the software [Koopman 2010]. According to Stevens et al. [1979], seven types of coupling exist, summarized in Table I. The tightest is 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. They both ultimately extend the C language that relies on typed interfaces, and different components need to use common data formats.

6.3. 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 are to 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 developers possibly 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 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 the corresponding 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, an entirely new global variable is necessary to switch between these states, further increasing the coupling among components.

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.

6.4. 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
Table III: Complexity comparison. The context-oriented design and CONESC yield simpler implementations that are easier to debug and to reason about.

<table>
<thead>
<tr>
<th>Application</th>
<th>Variables</th>
<th>Functions</th>
<th>Per-function states (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wildlife tracking – nesC</td>
<td>6</td>
<td>8</td>
<td>12567.3</td>
</tr>
<tr>
<td>Wildlife tracking – ConesC</td>
<td>3</td>
<td>2</td>
<td>6231.2</td>
</tr>
<tr>
<td>Smart-home controller – nesC</td>
<td>2</td>
<td>2</td>
<td>18654.2</td>
</tr>
<tr>
<td>Smart-home controller – ConesC</td>
<td>0.8</td>
<td>1.9</td>
<td>5678.3</td>
</tr>
<tr>
<td>Adaptive stack – nesC</td>
<td>2.5</td>
<td>3.25</td>
<td>9830.3</td>
</tr>
<tr>
<td>Adaptive stack – ConesC</td>
<td>0.4</td>
<td>1.6</td>
<td>3451.8</td>
</tr>
</tbody>
</table>

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. It 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.

**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 5, 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.

### 6.5. Programming → MCU and Memory Overhead

The use of CONESC comes at the cost of run-time overhead. Compared to a handcrafted implementation, for example, the code our translator automatically generates is likely less optimized in terms of memory occupation or processing time, with the latter possibly affecting energy consumption as well. To assess 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 widespread 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 as 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 3.4.

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.

6.6. Verification → NuSMV Model Generation

The automatic verification technique we conceive involves two steps: i) generating the semantically equivalent FSM as explained in Section 4 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 4 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 6.1, 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%.
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.

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.

6.7. Verification → Running Time

We measure the time taken by NuSMV to verify the semantically equivalent FSM. We consider both the context-oriented designs of the representative applications in Section 6.1 and the synthetic ones of Section 6.6. Because of the way NuSMV works internally, its running times change depending on whether the model is correct or includes different types of flaws. Whether NuSMV is required to generate a detailed counter-example also affects the running time. To cater for these aspects, we configure the input models and NuSMV differently in different experiments.

For the representative applications of Section 6.1, we consider the cases where the model is correct and where it contains flaws such as deadlocks, unintended transitions as defined in Section 4, and violations of developer-provided CTL properties. To trigger the latter, we artificially introduce selected 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 generate a counter-example whenever one is found.

In contrast, for the synthetic designs of Section 6.6, we limit ourselves to the case of a correct model or a model with a deadlock. In addition, we check the impact of generating the counter-example whenever one is found. We fix $N_C = 5$ and vary $N_{CG}$ to investigate how NuSMV scales with the size of the input models.
Fig. 22: NuSMV running time with the FSM equivalent to the context-oriented designs of Section 6.1. The absolute values remain below 50 ms. Verification of a flawed model requires more time because of the generation of counter-examples.

Fig. 23: NuSMV running times with the FSM equivalent to the synthetic designs of Section 6.6. The running times remain practical up to $N_{CG} = 8$, and are always below one minute for $N_{CG} \leq 7$.

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 6.6.

**Results.** Figure 22 shows the NuSMV running time with the FSM equivalent to the context-oriented designs of the applications of Section 6.1. 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-oriented design, discussed in Section 6.6, the total running times remain practical. The running time when the model is correct is constantly lower than when the model includes a flaw. This is mainly due to the cost of generating the counter-example when found. There is no significant difference in the additional running time 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 6.6. 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 in Figure 22, a flawed model requires slightly increased running times. However, the chart provides evidence that this overhead comes from the generation of the counter-example. When this functionality is disabled in NuSMV, 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 violation is found.
7. 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. By the same token, 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 in these cases 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. In applications such as wildlife tracking, this approach 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, as we demonstrated by investigating diverse applications in Section 6.

Fleurey et al. [2011] propose a model-driven approach to develop adaptive WSN firmwares. They model an application as a single 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 leave the door open to fine-grained optimizations.

Programming support. Three main programming approaches typically provide support for adaptation: Context-Oriented Programming (COP), Meta-programming, and Aspect-oriented programming (AOP) [Salvaneschi et al. 2013]. We choose to borrow concepts from COP in that meta-programming requires modifications of the binary at run-time, which is hardly feasible on resource-constrained devices. In contrast, AOP is usually applied for large software systems; hence, it would likely be overkill in WSNs.

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 verification tool that checks ANSI-C sources against safety properties such as correctness of pointer constructs. The SLAM toolkit developed by Ball and Rajamani [2002] determines whether a C program violates programmer-provided correctness rules. The BLAST tool of Beyer et al. [2007] formally proves that a C program satisfies predefined 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 and others 2004] operating system. Anquiro [Mottola and others 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.

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 we design and implement, 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.

REFERENCES

A. Dunkels and others. 2004. Contiki - a lightweight and flexible operating system for tiny networked sen-
Proc. of Simultools.
N. Finne and others. 2010. Improving Sensornet Performance by Separating System Configuration from
System Logic. In Proc. of EWSN.
SEAMS.
H. Fotouhi and others. 2012. Smart-HOP: A Reliable Handoff Mechanism for Mobile Wireless Sensor Net-
works. In Proc. of EWSN.
based Context Transition. In Proc. of AOSD.
C. Magerkurth and others. 2005. Pervasive Games: Bringing Computer Entertainment Back to the Real
F. Mattern and others. 2010. ICT for green: how computers can help us to conserve energy. In Proc. of ICCECN.
L. Mottola and others. 2006. Pervasive Games in a Mote-enabled Virtual World Using Tuple Space Middle-
ware. In ACM NETGAMES.
Proc. 2010 ICSE/SENSEA.
B. Piaztor and others. 2010. Selective Reprogramming of Mobile Sensor Networks Through Social Commu-
nity Detection. In Proc. of EWSN.
Divorce?. In Proc. of FSE/SDP FOSER.
works Before Deployment. In Proc. of IPSN.
S. Sehic and others. 2011. COPAL-ML: A Macro Language for Rapid Development of Context-aware Appli-
A. Wood and others. 2008. Context-aware wireless sensor networks for assisted living and residential moni-
M. Zimmerman and others. 2012. pTunes: Runtime parameter adaptation for low-power MAC protocols. In
Proc. of IPSN.