POLITECNICO DI MILANO
DEPARTMENT OF ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

SOFTWARE LEVEL ADAPTATION IN CYBER PHYSICAL SYSTEMS

Doctoral Dissertation of:
Mikhail Afanasov

Supervisor:
Prof. Luca Mottola

Tutor:
Prof. Luciano Baresi

The Chair of the Doctoral Program:
Prof. Carlo Fiorini

2015 – XXVIII

Abstract

CYBERPHYSICAL systems (CPSs) are a category of engineered systems that combines physical processes with computational control. The key property of these systems is that their functionality is emerging from the intense interactions between computational devices and the real world.

Consider a typical CPS application: a Wireless Sensor Network (WSN) for the environmental monitoring. The environment exhibits multiple dimensions that are changing continuously and independently. As the WSN monitors these changes, its functionality heavily depends on the environmental dynamics, which leads to the need for the WSN software to be *adaptive*.

Another large class of CPSs are Unmanned Aerial Vehicles (UAVs). These are representative of time-critical CPSs. The typical UAV control board also consists of a number of sensors that are used to calculate the modulation of the motors and to keep the flight stable. Similarly to WSNs, UAVs monitor the environment through sensors to control the flight. Crucially, UAVs rely on the sensoric input, and generate the control decisions in *real-time*.

In this thesis we focus on the adaptive software for such systems. Our goal is to provide language independent concepts that can help developers to design, verify and implement the adaptive software for time-critical systems. Unlike most existing work, we do not present the mechanisms or algorithms for adaptation. Our aim is to make the ways these mechanisms and/or algorithms are designed, programmed, and verified more effective.

In the first part of this thesis we introduce our language-independent design concepts to organize the WSN operating modes, decoupling the abstractions from their concrete implementation in a programming language. Our language CONESC natively supports the dedicated adaptation mechanisms and allows developers to implement adaptive WSN software. To verify this software we provided a dedicated verification algorithm. The latter is integrated with our tool GREVECOM that delivers the seamless way to build the model of the software, to exhaustively verify it against the environmental evolutions, and to build the CONESC templates. Finally, CONESC sources are compiled with the dedicated translator yielding the binary that is ready to be deployed. The evaluation have shown that our concepts increase the ease of the developing and verifying of the adaptive WSNs software with a very little price, such as less than 2.5% memory overhead and less that 200ms verification time.

In the second part of the thesis we focus on the time aspect in enforcing adaptation decisions. Our concepts deliver different activation types for the CPS operation modes, trading off latency at run-time vs. programming efforts. Each activation type has also time boundaries that can be optionally enabled by the programmer. We show the usefulness of our concepts in a prototype built for the Cortex-M3 micro-controller. Our early evaluation has revealed that with the cost of a small MCU performance overhead, we provide an additional functionality and guaratees that do not exists in current approaches.

Contents

CHAPTER 1

Introduction

CPSs are defined as systems, where physical processes are tightly integrated with the computational processes. This integration make CPS to interact with and possibly taking actions on the real world.

1.1 Motivation

The real world exhibits multiple dimensions that influence the behavior of CPS software continuously and independently. These require the CPS software to cope with uncertainty of sensoric input and to produce corresponding reactions. This close interaction between the real world and CPSs reveals several challenges.

In CPSs, the events that occur in the physical world have to be reflected in the CPSs software, and the decisions taken by the software influence the physical world [37]. Due to this tight integration, the CPSs software is continuously confronted with a range of largely unpredictable environment dynamics and changing requirements. This demands CPSs software to *adapt* to a range of different situations.

The adaptation problem is even more difficult under time constraints. CPSs also include such systems as: aerial drones, and sensor networks for automobiles or railways. All these systems have soft or hard real-time requirements, and in all these cases CPSs are required to handle both periodic and aperiodic tasks within *time constraints*.

There are two major classes of systems we focus in this thesis. *Wireless Sensor Networks* are typical platforms where adaptivity is required, since the very functionality of WSNs is entangled with the highly dynamic environment. *Adaptive Time-Critical Systems* are also highly adaptive, but in addition to that, the adaptation is performed with time boundaries.

Taking into explicit account every possible situation in the design and implementation of CPSs software is a challenge. Crucially, multiple combined dimensions concurrently determine how the software should adapt its operation. Moreover, these operations may have to be preformed under time constraints. The challenge increases even more when developers battle against the resource limitations of many existing CPS platforms. Using available approaches, this typically results in entangled implementations that are difficult to debug, to maintain, and to evolve. As the number of dimensions affecting the execution (and their combinations) grows, the implementations quickly turn into "spaghetti code" [24].

1.2 Contribution and Roadmap

We address this challenge by providing a handful of concepts that provide a time-critical adaptation mechanisms and help developers to implement adaptive CPSs software under resource and time constraints.

The **first part** of this thesis is intended to solve the *adaptivity* problem that is described in Section 3.1. To do so, in Section 3.2, we adapt the Context-Oriented Programming [64] to WSNs – a paradigmatic example of resource constrained CPSs. In doing so, we provide full support for developing adaptive software for WSNs. In Section 3.3 we outline a handful of concepts that greatly simplify the design of the adaptive software for WSNs. We argue that two main notions in the WSN software are: *i) context* that represents a single environmental situation the software executes in, and *ii) context group* – a collection of the contexts sharing common characteristics. These concepts are implemented in our own language CONESC that is described in details in Section 3.4. As the software model needs to be continuously verified during the development, in Section 3.5 we elaborate on a verification algorithm for the context-oriented models. Our tools described in Section 3.6 allow a designer and a programmer to utilize our concepts in a real development process. With our tool GREVECOM the designer can build a model of the adaptive software and exhaustively verify it against environmental evolutions. Based on this model, the ConesC templates are automatically generated allowing the programmer to implement the actual functionality of the application. Finally, our dedicated translator generates the binary based on ConesC sources.

In Section 4.1, we describe our early experience in developing the adaptive WSN software using our concepts and tools. We also noticed particular recurring patterns that are used in some application. Our evaluation in Section 4.2 has shown that our concepts make the software components much more decoupled and more simple, which directly influences the ease of debugging, maintaining, and evolving the code. These benefits come with a cost of memory overhead and verification time. We have shown, however, that our approach has a little price: less that 2.5% memory overhead and up to 200ms of the verification time.

The **second part** is devoted to time-critical systems. In Section 5.1 we show that the adaptation problems are similar to the ones we observed in WSNs, but in this part we focused on the context activation time aspect, as described in Section 5.2. In our solution, described in Section 5.3, in addition to the design concepts described in Section 3.3, we provided different types of context activation: *i) fast* and *ii) lazy*. The main difference between the two is that the *fast* activation requires less time, but more efforts from a programmer's perspective, while the *lazy* one allows the programmer not to spend much efforts with the cost of increase activation time. As in time-critical systems tasks may have deadlines, in our solution, the programmer can also add a deadline to the activation command: whenever the deadline is not met, the programmer will be notified about the failure. Our prototype described in Section 5.4 implements these concepts and shows how our concepts can be used in an implementation of adaptive software for time-constrained CPSs. Our measurements in Section 5.5 reveal that our concepts cover a significant part of the timing aspect of the adaptation, while the overhead is fairly modest.

CHAPTER 2

State of the Art

Cyber-Physical Systems (CPSs) are a class of systems that continuously interacts and possibly affects the real world. Due to the dynamic nature of the real world, CPSs should always adapt their functionality according to the changes in the environment they operate in. In this work we focus on a large group of CPSs: *Wireless Sensor Networks* (WSNs). In the section 2.1 we show the need of adaptivity in WSNs, and explore the existing approaches towards the adaptivity in these platforms. Then, we investigate another big group of CPSs: *Time-Critical Embedded Devices*, for example, aerial drones, railways, robots, etc. The common characteristic of the platforms from this group is that the time required for adaptation is crucial, and we show it in the section 2.2. In the section 2.3 we discuss different approaches and programming paradigms, which provide language independent concepts and a language support for implementing adaptation mechanisms for the adaptive software.

2.1 Adaptivity in Wireless Sensor Networks

Wireless Sensor Networks (WSNs) are embedded platforms that are wirelessly connected to each other forming a network. Each node in the network is typically equipped with a processing unit, set of sensors and/or actuators, and a wireless communication interface. Most of the applications for WSNs utilize tiny, battery powered devices that are deployed over some area. Such deployment provides a fine-grained interaction with the world, a low-cost re-deployment, and an increased flexibility in scenarios. The advantage of this technology would not be fully achieved without a proper programming support.

WSNs operate in a highly dynamic and rapidly changing environment. In order to keep a good quality of service and a functionality that is adequate to the current conditions, a software for WSN must react to these changes and adapt to them accordingly. As it is shown in figure 2.1, there are two main dimensions in the approaches towards adaptive WSNs: *i)* adaptation type; and *ii)* adaptation scope. Along the latter dimension existing approaches achieve the adaptivity by modifying either a network topology, an application's global state, or protocol's parameters. Approaches along the first dimension achieve the adaptivity in a different way: either through the re-programming, or by applying different configurations depending on the situation. These dimensions are orthogonal, though, so every approach can be evaluated in terms of this two-dimensional space.

The common approach to the adaptation is through the *reprogramming* the *application* logic at run-time, as we can seen in figure 2.1. The less usual approach to the adaptation is the *parametric* adaptation either on the *network* level, the *application* level, or the *protocol* stack level. Through our examination we found out that the adaptation in the *protocol* stack is only achieved through the *parametric* type of adaptation. The same situation can be observed in the adaptation in the *network* level: the *reprogramming* is not used in this case.

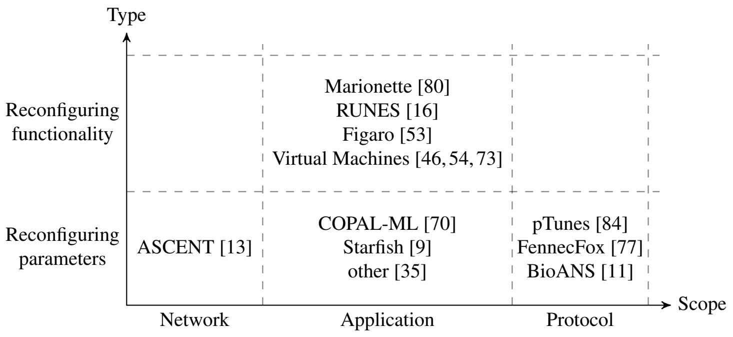


Figure 2.1: *Adaptation in WSN.*

In section 2.1.1 we review typical applications for WSNs. Then we discuss the approaches – e.g., that are displayed in figure 2.1 – in terms of two dimensions: *Scope* (section 2.1.2), and *Type* (section 2.1.3).

2.1.1 Paradigmatic Applications

WSNs are widely used in different applications such as monitoring the man's health state, or controlling the climate system in a house. For example, the goal of WSN for health monitoring [42] is to continuously monitor an individual's physical conditions. Body-worn sensors track quantities such as heart rate, blood pressure, body temperature, and so on. WSNs are also used in a tunnel that is instrumented with networked embedded sensors [16]. The latter are reporting the status of the tunnel and allow one to detect and to react to the emergency situations. The list of the application can be continued even further, but here we focus only on two representative and diverse examples of WSN applications, which require adaptivity.

Wildlife tracking. WSNs are widely used to track the animals [58]. In these applications, sensor nodes are attached to the collars of animals to study their movements, e.g., using GPS and accelerometers, and health conditions, e.g., based on body temperature. A low-power short-range radio allows nodes to detect social interactions of animals based on periodic radio beaconing. A node logs acquired data on a local memory, until the opportunistic encounter with a fixed base-station. The radio is used in this case to offload the readings. Small solar panels harvest energy to prolong a node's lifetime.

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

Smart home. The sensor network is deployed in a house and equipped with temperature, lighting, smoke and movement detection sensors. Based on the gathered data, the system controls the temperature and the lighting levels in a house. If a housebreaking is detected, e.g., by using movement detection sensors, the system sends the message to the police. If the system recognizes fire, e.g., using smoke detection sensors, it sends the alert message to firemen.

The need in adaptivity arises when the system needs to tune the temperature and the lighting. The latter depends on the level of natural light and on the time of the day. For example, in the evening natural light is not enough and the system turns the lights on. However, at night the lights should be switched off. Desired temperature also depends on time and the user's preferences. Moreover, the user's preferences may vary depending on a day of a week. Thus, the user may prefer the higher temperature at night, and the lower one at working days. However, during a weekend, the user may prefer the higher temperature for both day and night. Emergency situations are even less predictable, and the system must rapidly adapt its functionality to respond to a housebreaking or a fire.

Adaptive protocol stack. This application only affects the system level of the software. In the WSNs where the nodes have periods of significant mobility interleaved with periods of static operation. Within the network of static nodes the node joins to the network by running an instance of CTP [30]. As soon as the on-board accelerometer detects a significant movement, the node switches to a route-less gossip based protocol, which allows to relay data opportunistically [27]. In addition to the protocol switching functionality, the node can change the parameters, based on the performance priority, which determine whether bandwidth, latency, or lifetime is favored. In this application, the need in adaptivity arises, when orthogonal to the switch between different protocols, the software must tune the protocols according to the situation.

The applications we described above are using WSNs in very different scenarios. However, all of the application require the adaptivity. The latter is achieved with different approaches that we describe in the next sections.

2.1.2 Adaptation Scope

As it is depicted in figure 2.1, the adaptation scope determines whether the whole network or only a single configuration of a node is involved in the adaptation process. The adaptation can also occur entirely in the application logic, without any changes in the network or node's configuration.

Network level. Usually, in WSNs every node has its own role. Adaptation at the network level involves a dynamic changing of the roles of the nodes in the network. Approaches at this level focus mainly on such properties of the network as topology, scaleability, components' roles, and fault-tolerance.

For example, Subramanian and Katz [76] proposed a set of roles, which are forming an adaptive architecture for WSN:

- *Specialized sensors* monitor different physical entities.
- *Routing sensors* provide a data routing and a fault tolerance of the network.
- *Aggregator nodes* combine routing and sensing functionalities to provide a better network flexibility.
- *Sink nodes* have a high storage capacity to store and to process received data.

All the components mentioned above are sufficient to build a wide range of applications, where the infrastructure includes addressing, routing, broadcasting and multicasting mechanisms. Given the mentioned architectural components, the authors also propose four steps, which should be performed by the network to self-organize:

- *Discovery phase.* Each node discovers its neighbors.
- *Organizational phase.* Nodes organize groups, allocate addresses, build routing tables and construct a broadcast tree.
- *Maintenance phase.* Each node keeps a track of its energy; constantly updates its routing table, broadcast trees and graphs; and sends *"I am alive"* message and its routing table to the neighbors.
- *Self-Reorganization phase.* If the node detects the failure of a neighbor, it updates its routing table, or starts the Discovery phase in case of the failure of all of the neighbors.

The analysis of the approach shows that the hierarchy of the network is strictly balanced; the complexity of the routing is *O(log n)*; and the network is extremely tolerant to either node or link failures. However, this approach is not applicable for mobile WSNs, such as one for wildlife tracking that is discussed in section 2.1.1.

A similar approach is proposed in ASCENT [13]. This work proposes an adaptive network topology for networks with high density of the nodes. The authors outline two types of nodes: *i)* active and *ii)* passive. The latter just passively listen and periodically check if they have to be involved in the routing process. Nodes of the first type always stay awake, sample data, and provide a routing functionality. In the case of low quality channel, the active nodes send a help message to the passive nodes in order to activate them.

The self-configuring process starts by turning a random number of the passive nodes on. The latter enter in a test mode, and if there are too many neighbor nodes active, the passive node will be switched off. Otherwise, the node switch to the active mode after a predefined time. If the number of active neighbor nodes is less than a prefixed parameter, or the data loss rate is greater than a predefined threshold, the re-configuration process will start again.

Application level. Another way to achieve the adaptivity is to vary the application's behavior depending on the environmental situations. For example, COPAL-ML [70] provides language abstractions that tackle changes in the environment and vary the behavior of the software accordingly. By using this framework, developers can specify the way the data will be collected, processed and used by the application, depending on the conditions the network is working in. This approach, however, uses the WSN as a source of a raw-data, while the main application and its adaptation are executed solely on the base station. Thus, the approach is inappropriate for applications without a central control, or where the adaptation should occur on the device.

Another example of the adaptation at application logic is Marionette [80]. The main concept of that tool is a *remote procedure call function* (RPC-function) – a function that runs on the embedded device, but can be called remotely from within the Marionette tool. With this tool a programmer is able to split the software into two parts: *i)* a node-specific functionality, such as a data sampling, messages relay, etc.; and *ii)* the application logic, such as data processing and network control. The main application logic runs on a personal computer (PC), while a node-specific functionality is implemented as a set of rpc-functions and deployed on the nodes. This approach allows a programmer to arbitrarily call RPC-functions, depending on the needs of the application. The application logic, however, can be changed at any point of the execution, because it's not on the node. With this approach a single RPC-function implements only a fraction of the application logic, thus, a sensing node should always be connected to a PC. The latter calls RPC-functions in the appropriate order. In a majority of applications, the connection between the network and the base station is not guaranteed, hence, this approach is hardly applicable there.

Virtual Machines (VMs) for WSNs [46,54,73] also provide tools that allow programmers to change the application logic at run-time. The common feature of VMs is that the system consists of two major parts: *i)* instructions that are written by a programmer and sent to the VM, and *ii)* the VM itself, which executes these instructions.

For example, Mate [46] is a bytecode interpreter that runs on a single node. The code for Mate is broken into *capsules*, each of 24 instructions. Mate hides the asynchrony of TinyOS programs, as the execution of each code-capsule is performed in a synchronous way. For example, to measure the light intensity, the programmer sends a code-capsule that reads the values and stores them into non-volatile memory. Then, the next code-capsule instructs Mate to read the value and to send it over the network. While providing a very flexible and expressive way of programming WSNs, Mate also limits the programmer by code-capsules, each of which can only contain up to 24 instructions. Moreover, for more complicated applications, nodes have to be always connected to the Base Station. For example, the programmer has to send one code-capsule to sample the light readings and to store them into the memory. The processing of the data is only possible from within the other code-capsule that is sent from the base station.

Another way to dynamically change the logic of the application is to deploy these capabilities into a component model of the software. This class of approaches define how the software components have to be reconfigured, and how do they interact with each other. For example, OpenCom [17] introduces a component model that provides such information as: the graph of the components, interaction between the components, and reconfiguration policies. By modifying the system graph, a programmer can modify the logic of the applications. Similar capabilities are offered by Starfish [9], Figaro [53], and RUNES [16] that are described in section 2.1.3. These frameworks can also be utilized to build adaptive applications.

A slightly different approach is proposed by Huebscher et. al [35], where they proposed a framework to build a component model that is tailored to a specific application. The framework includes two main abstractions: *i* the Context Provider and *ii)* the Context Service. The Context Provider collects the raw sensoric data and interprets it to produce context types for the further usage by the higher level, where Context Services operate. The latter hide the Context Providers from the application and provide the contextual data to the application. In doing so, Context Service examines the metadata of Context Providers to select the most appropriate one based on such metrics as: availability, precision, refresh rate, etc. Developers implement Context Providers for each type of the raw sensoric data, and Context Services for each context type. The application itself is built on top of the Context Services. The adaptation occurs whenever the new Context Provider appears, or the existing one fails. In this case, Context Service re-examines the Context Providers transparently to the user. This approach along with others mentioned above are designed, however, to provide solutions to specific problems. In our work we provide a general adaptation mechanism.

Protocol level. At this level programmers often consider the reconfiguration of MAC protocols. One of such examples is pTunes [84]. The proposed framework allows one to adjust the parameters of the protocol stack to adapt to link, topology, and traffic dynamics. To this end, authors break a performance model into three levels: *i)* an application level; *ii)* a protocol independent node-specific level; and *iii)* a protocol dependent level. At the application level, such parameters as *reliability*, *latency*, and *network lifetime* are defined for the whole network as functions of node-specific reliability, latency and lifetime. The latter are defined on the node-specific, yet protocol independent level. On the protocol dependent level, the authors provide a mechanism that can be adopted to optimize protocol parameters and, thus, to achieve the required values of node-specific metrics. One of the main advantages of this approach is that only the protocol-dependent layer must be changed to adapt the performance model for another MAC protocol.

By using the model described above, the authors implemented a Java-based control application. It retrieves the network state, and starts the optimization process based on the performance model. The optimization process in pTunes can be triggered by three dedicated *triggers*. The *TimedTrigger* optimizes parameters periodically; the *ConstraintTrigger* launches the optimization only if particular constraints – for example, a maximum latency – are violated; and the *NetworkStateTrigger* fires the optimization if the network state requires the MAC parameters to be updated. For example, a high traffic volume requires a higher bandwidth. The framework addresses such challenges as: minimum disruption, timeliness, consistency and energy efficiency. However, the control application and the solver are run on the base station and the network should always be able to communicate with the base station to adapt to the network dynamics.

Another work at this level is BioANS [11] – a self-configurable network protocol for WSNs. Being based Autonomic Networked Systems (ANS) the protocol dynamically chooses the best node for each request. Every sensor within the communication range responds with the random delay. If quality of service of the node is sufficient, the other nodes stop responding. The adaptation occurs when the message is lost or some nodes are failed. If the requester receives no answer, it simply repeats the request and the selection procedure repeats. If any of the nodes are failed, they do not participate in the procedure.

Another example is the Fennec Fox framework proposed by Szczodrak et. al [77] that provides *reconfiguration policies* – high-level programming abstractions for WSN reconfiguration. Similarly to the component-based approach, the authors split the protocol stack into four layers: radio, MAC, network, and application. Each level contains one or more *modules* that provide an implementation of the service for the each layer. Depending on the layer, modules can be: *i)* an application; *ii)* a network protocol, such as Collection Tree Protocol (CTP) or Parasite Network Protocol (PNP); *iii)* a MAC protocol such as Carrier Sense Multiple Access (CSMA) or Time Division Multiple Access (TDMA); and *iv)* a driver of the particular device. A single *configuration* in this framework is a set of four modules, one for each layer. The reconfiguration is a process of switching from one configuration to another.

Fennec Fox is running on each node and provides a high-level programming abstractions – Swift Fox. It allows one to define configurations, priorities, events, and reconfiguration policies. The latter defines which configuration will be loaded whenever a dedicated event occurs. If a single event invokes several configurations, the configuration with the highest priority will be chosen.

The adaptation process at this level is very narrow and allows one only to reconfigure software components related to the protocol stack. Moreover, while providing the adaptation in a very specific aspect, these approaches do not allow programmers to implement their own adaptation mechanism.

2.1.3 Type of Adaptation

Orthogonal to the scope, the type of the adaptation determines the mechanism of the adaptation. As it is depicted in figure 2.1, we outline two types: *i)* parametric; and *ii)* re-programming.

Parametric adaptation. This type of the adaptation focuses on the tuning of the parameters of the algorithm, software component, or hardware module. This type of adaptation affects not only the protocol stack, but also other aspects of the system.

For example, Starfish [9] provides a policy driven adaptation of WSN. Starfish is a part of Finger2 framework, which provides the most commonly used functions in WSNs. For example, the *Sensor* module provides a standard routine for periodic sampling through the *Sense()* function, the *Get()* function immediately retrieves the value from a sensor, and etc. The module *Buffer* provides storage facilities on the node. The *Timer* module provides an interface for scheduling of future or periodic tasks. The communication interface is provided by the *Network* module. Starfish extends Finger2 by providing the *Policy* module for the policy management. This library allows programmers to *Enable()* or *Disable()* particular policies. The *Install()* operation deploys a new policy, and *Remove()* deletes the policy. These operations can be performed at run-time providing a basic mechanism for self-adaptation.

The authors proposed the notion of the *mission* – a set of policies to accomplish a specific task. The notion of *role* allows one to dynamically assign policies to a specific node, while the *configuration* is a piece of a source code that is loaded on a node. The framework allows programmers to build a reconfiguration strategy for dealing with sensor errors, component failures and the appropriate reconfigurations. Similarly to the *Policy*, the *Mission* module provides a manipulation of missions by loading, adding, or removing them. The same functionality with respect to the roles is implemented in the *Role* module. By managing the policies, missions, and roles at run-time, programmers are able to create effective reconfiguration strategies depending on the environment dynamics.

A similar approach is used in pTunes [84], where the system automatically optimizes parameters of the protocols stack, which is already described in previous section. Another reconfiguration-based approach is proposed by Fleurey et. al [26] where they developed a framework, which allows programmers to develop an adaptive firmware for sensor nodes. With this framework the programmer specifies the model and all possible variations of that model. Optimal configurations for these models are found through the exhaustive simulation. Then a state machine is created based on these configurations. This state machine is used then to generate a source-code.

The adaptation mechanisms in ASCENT [13], COPAL-ML [70], and Fennec Fox [77] (see section 2.1.2) are also built on top of the parametric paradigm. The common idea of these approaches is that the adaptation only occurs in a set of parameters of the software or hardware component. For example, in ASCENT, the adaptation is performed by changing the role; COPAL-ML changes the global state according to the context; and Fennec Fox changes the configuration. However, in our work we provide concepts, which allow programmers to define where exactly the adaptation should take a place.

Re-programming. Run-time reprogramming is widely used as an effective adaptation type. This approach allows programmers to replace an old component, or even to add a new one without interrupting the execution flow.

For example, Contiki [20] is an operating system, which allows programmers to distribute code updates and to reprogram sensor nodes at run-time. Despite an ability of complete reprogramming of a node, the transfer of binary images over the network requires a high bandwidth. Together with an unreliable communication medium, this transfer increases energy consumption. Furthermore, for updates to work, a node has to be rebooted, and its operation has to be interrupted.

Contiki consists of the kernel, the program loader, and a set of processes and libraries. A process may be either an application or a service. The latter provides a functionality that is used by several processes. Both the application program and the process can be replaced at run-time. The whole systems is partitioned into two parts: the core and the loaded programs. The latter are loaded by the program loader. The core contains the kernel, the program loader, the most commonly used functionality, and the support libraries.

Project RUNES [16] proposes a language-independent component model that can be used to implement adaptive heterogeneous systems. In this model, the *component* is a single unit of deployment that can be instantiated at run-time. Each component provides a functionality through the *interface*. Moreover, the component contains a set of dependencies that are expressed in terms of one or more *receptacles*. By using these, the system links the components to each-other through *connectors*. All the meta-data can be stored in *attributes*. The collection of attributes is a *capsule*. This component model allows programmers to dynamically change the hierarchy of the components.

RUNES framework is implemented in three parts: *i)* a Java-based implementation contains the code needed for the components instantiation and destruction; *ii)* a C/Unix-based gateway; and *iii)* a Contiki-based run-time environment for sensor nodes. Dynamically generated components are sent to the nodes through the gateway, and deployed on the nodes using the Contiki tool-chain.

Being built on top of Contiki, Figaro [53] provides programming abstractions, which provide more fine-grained updates of the software components. Being a components model, this approach makes it possible to re-program sensor nodes on the fly by updating old, or deploying new software components. In Figaro, the run-time environment automatically manages the reconfiguration process, based on the dependencies declared by a programmer. When components are instantiated, the systems keeps track of their versions, interfaces they implement, and dependencies they have. If the component is not able to be instantiated because of the missing mandatory dependency, it is buffered in hope to receive necessary components later on. Moreover, in Figaro, programmers are able to specify a single part of the WSN where the reconfiguration takes a place. It is achieved by two different ways: *i)* by specifying the attributes that characterize the node; or *ii)* by addressing a subset of nodes directly.

Similarly to Contiki, Dynamic TinyOS [55] enables dynamic reprogramming of the components in TinyOS. At the compilation phase, Dynamic TinyOS preserves the parts selected by a programmer. The result is an executable with multiple replaceable objects that can be replaced at run-time over the air. The code update is done in three phases: *i)* the components of the application and the system are compiled into multiple objects; *ii)* updates – e.g., binary objects – are transferred over the air; and *iii)* updates are stored and linked to the application or to the OS at run-time. The last phase is done by the *Tiny Manager* – a run-time system that is executing on the node and handling storage and integration of the new components. Thus, the application logic and the software architecture can be modified at specific points, defined by a programmer.

As we described in section 2.1.2, Marionette [80] and virtual machines for WSNs [46, 54, 73] also provide re-programming facilities that are used to build adaptive applications. These approaches, again, provide a very generic mechanisms that can be utilized to implement the adaptive functionality. The implementation of the adaptation mechanism, however, is completely on the programmers shoulders.

2.2 Adaptivity in Time-Critical Systems

Time-critical systems are such systems where tasks have deadlines. Should the deadline be violated, the system treats it as a failure. These systems are used in automobiles, robots, aerial drones, railways, etc. The problem of the adaptation in these systems has the same background as in WSNs. Indeed, these platforms are using time-critical embedded sensing devices that are also tightly coupled to the environment they operate in, and, thus, have to adapt to the changes in this environment, as we have shown in section 2.1. An additional challenge in such systems arises from time constraints: the system has to change its behavior within the specific time requirements.

In section 2.2.1 we describe representative applications of time-critical embedded systems. A common approach that addresses the problem of the task execution at the real-time is an adaptive scheduling that is described in section 2.2.2. Apart from scheduling, several approaches for real-time systems are based on the dedicated component models, their architecture and interaction patterns. These approaches are described in Section 2.2.3. Finally, real-time reconfiguration for the whole system, which is described in section 2.2.4, is also widely used.

2.2.1 Paradigmatic Applications

Among the great number of different applications of time-critical systems, we focus on two diverse and the most representative applications, where the adaptation has to be performed at the real-time.

Gas Leak Localization. In this application, a swarm of aerial drones is used to monitor a 3-dimensional area in order to find a high gas concentration – a probable gas leak [10]. Each drone is equipped with a gas sensor that allows the drone to measure a concentration of the gas; a low-range radio that is used to discover the neighboring drones via periodic beaconing; and a high-range radio to receive coordinations from the Base Station. Periodic beaconing is also used to exchange the data – e.g. gas concentration samples – between the neighboring drones.

At the beginning, each drone moves to the predefined location in the area. The map is prealoaded into the memory of drones, and each drone knows the path to its location. Upon arriving, drones start sampling the data and exchanging this data with each other. Whenever a single drone, or a group of drones in the swarm detects a high gas concentration, drones perform more fine-grained sampling. To do that, they reduce the distance to the neighbor that sampled the highest value of the gas concentration, and continue sampling. The gas leak is considered localized, when the further reduction of the distance between the drone will lead to the collision between them.

Orthogonal to the main task, drones with low batteries return to the charging station. If the swarm is partially covered by the base station communication range, drones switch from the direct communication with the base station to the Collection Tree Protocol.

Drones are representatives of time-critical systems. Indeed, the control loop of a drone runs with the period of 10-100ms, or even less. Each iteration, the control loop gathers the information from the sensors, analyses it, and issues commands to the motors. Should some internal task of drone execute longer than expected, the control loop will not be able to update the parameters for motors, and drone will crash. Thus, in these systems, adaptation should occur within the time-constrains.

Automotive Embedded Systems. WSNs are widely used in automobiles [82]. The latter consist of a great number of electronic devices that realize the functionality of the car. This networked embedded system controls the engine; provides the safety features – e.g., anti-lock breaking system; keeps the vehicle running; provides convenient driving assistance; and entertains passengers.

Consider the networked embedded system in the automobile that has to adapt to the changes in the environment, as well as to the internal changes. For example, based on the fuel level, the system may dynamically switch the engine from the performance to the save mode. Orthogonally, the ambient light directly affects on the headlights' brightness: it's zero when the weather is clear; if it's cloudy or foggy, the brightness is slightly higher; the maximum brightness is in the evening, or at night; in case of incoming cars, the brightness should be decreased. In the same time, the system controls the climate depending on the user preferences and the temperature outside.

The similar adaptation problem we also observed in the *Smart Home Controller* before. Here, however, the system has to react quickly to the changing conditions, either from inside, or outside the system. For example, the system has to handle hardware or software failures at real-time, since the car is moving. In such systems, tasks have deadlines, for example, in the critical situation – e.g. hardware or software failure – the system has to activate breaks before the collision occurs. For the same reason, any changes in the environmental conditions – e.g., an obstacle appearance or an incorrect trajectory – have to be tackled immediately. The natural way to handle the real-time task execution is scheduling. In the adaptive systems, an *adaptive scheduling* is widely used.

2.2.2 Adaptive Scheduling

Most of the works in time-critical systems assume that the task execution has fixed time parameters, e.g., periods, deadlines, etc. Schedulers play an important role in planning the tasks execution to meet the deadlines, and to balance the CPU load. Kuo et al. [44] discussed how to adjust the load in order to handle periodic tasks. The authors analyze existing scheduling algorithms and show how the load could possibly adjusted there. They conclude, that the load adjustment problem is application-specific, and, moreover, it opens another problem of the adaptive resource allocation.

Naturally, scheduling algorithms are divided into two approaches: under the non-overload conditions, the time-based deadline-monotonic scheduling (DMS) [5] algorithm schedules the tasks with an optimal processor load; and during the overload, a value-based scheduling (VBS) [32] is performed to execute the most *valuable* tasks first. In real-time systems it is important to detect which scheduling algorithm to use under current conditions. Richardson et al. [63] provide an adaptive deadline-monotonic (ADM) scheduling algorithm. This algorithm provides a fault-tolerance by detecting possible overload of CPU and switching from DMS to VBS. Possible overload is detected by monitoring the execution of each task, and by calculating the time before the deadline. In order to find the best moment to switch to VBS and to avoid the overload, authors introduce a *sliding-window* parameter – e.g., the time before the deadline that is considered critical. Should the time that is remained before the deadline be lesser than the size of the sliding-window, ADM switches to VBS.

Zhang et al. [83] developed an adaptive algorithm, which calculates an optimal timeout for saving the task execution state to tolerate possible faults. To this end, the authors introduce an adaptive check-pointing – a preservation of the task execution state. Whenever a failure occurs, the system performs a rollback to recover the last correct state. The goal of the algorithm, is to make the last checkpoint as close to the failure as possible and to allow the task to finish the execution within the time constraints. Every time the rollback occurred, the proposed algorithm evaluates a new check-pointing interval to reduce the number of rollbacks, while keeping the execution within the deadline.

A work similar to Richardson et al. [63] towards an adaptive scheduling for real-time systems has been done by Abeni et al. [1]. They proposed an adaptive real-time scheduling algorithm that is based on a feedback loop: Legacy Feedback Scheduler (LFS). The algorithm scans all the scheduled tasks and calculates the scheduling error for each time-sensitive task. Then the algorithm updates the time that is reserved for each task. This approach allows the programmer to adjust time-specific scheduling parameters for the tasks that are not designed for time-critical systems. Its performance, however, is lower than the performance of other dedicated schedulers for time-critical systems.

These works are complementary to ours. We do not focus on the particular scheduling problem, but provide a simple paradigm and programming tools, that can be used to implement an adaptive software. Currently, these tools are not provided by any of the existing approaches.

2.2.3 Component Models

The adaptation in dedicated component models is widely adopted in both Wireless Sensor Networks and Time-Critical Systems. The adaptation in the latter is achieved by a dedicated hierarchy of software components.

These models are developed to continuously monitor and calculate the parameters of the software components of the system.

For example, Zeller et al. [82] proposed a multi-layered control approach for self-adaptive systems. In this approach, the authors use a hierarchy of layers. A single layer has a number of control loops, each of them provides an adaptation of a single parameter (variable). Each control loop on the higher layer requests the decision from the lower layer, and then decides itself. The last decision is taken by the main control loop on the top layer. This kind of adaptation only focuses on a reconfiguration of a single set of parameters. While this approach is proven to be effective in automotive embedded systems, it is quite an overkill for resource constrained systems. Moreover, this approach considers the large networks with a great number of sensors, while we focus on a single device and its adaptivity.

Prechofer et al. [61] formalized the adaptation process for time-critical and resource-constrained embedded systems. In their work the authors consider a heterogeneous sensor network, where a software component can be transferred from one hardware platform to another. Typically, such systems are used in automotive embedded systems, which are described previously. In this case, the component should adapt to different platforms. In order to preserve the functionality, the software calculates the best configuration for the transferred component. Contrary, in our work we focus on a single device, where adaptation requires to react to the environmental changes. These changes result into a necessity to vary the functionality of the software component on withing a single device.

Design patterns that are commonly used in typical scenarios of real-time embedded systems are described by Loyall et al. [47]. The authors address to the issue, which arises during the transmission of the information within the time and resource – e.g., limited bandwidth and processing power – constraints. The proposed dedicated pattern for this issue – *QoS contract* – implies a definition of the requirements, which trigger the execution of a particular task. Another problem that is tackled in this work appears when the user wants to see the state of the whole system. Some values can be collected fast, other values require some time to be calculated. This situation disrupts the integrity of the received state: while the values from the one sensors will be calculated, values from the other may change already. Addressing this issue, the authors propose another pattern – *snapshot* – where the dedicated software agents make snapshots of the system and move it in the storage. The user will receive the snapshots from the storage. While describing these patterns in details, the authors do not provide any abstractions or tools, which can help a programmer implement an adaptive functionality. Our concepts do not rely on the specific scenarios, instead, they can be used to implement an adaptation mechanism regardless of language or scenario.

2.2.4 Re-configurable Systems

A re-configurable hardware brings the adaptivity in time-critical systems to the lower level. Differently from re-configurable software, in this class of systems it is possible to change the whole circuit. This hardware, such as Programmable Logic Arrays (PLAs) or Field-Programmable Gate Arrays (FPGAs), can be reconfigured by downloading a different configuration data. Thus, the design issues can be corrected, and physically damaged parts can be avoided. These serve for the better fault tolerance and adaptivity in real-time systems. In such systems, a circuit configuration is stored in memory cells.

By design, FPGAs allow developers to implement a high-performance computational logic, which is crucial in real-time systems. These chips, however, are subjected to cosmic radiation that can cause Single Event Upset [71] and alter the logic [33]. Moreover, the adaptation on these chips is performed via the reconfiguration of the whole chip, which takes the considerable amount of time. For this reason, the majority of the work is performed towards the error tolerance and the reconfiguration time.

Conde at al. [15] addressed the memory corruption issue by introducing a controller that compares the FPGA configuration with the configuration stored in a flash memory. In case the controller detects a difference, the original configuration from the flash memory is loaded into FPGA.

Alfke et al. [2] proposed to read a configuration from several FPGAs and then vote for the most appropriate one. Xilinx FPGAs offer a *readback* capability that can monitor the chips without interfering with the operation of the device in the system. The authors used this feature to build a triple-redundancy system: three FPGAs are connected in parallel and configured to operate synchronously. A small controller initiates a readback from all the three chips simultaneously, and compares the outputs. If the outputs are different, then the error correction is launched. The error is corrected in two ways: *i)* all three chips are reconfigured, or *ii)* only the faulty device is reconfigured based on the feedback of the others.

To reduce a reconfiguration time, vendors designed a chip that can be reconfigured only partially, one of such examples is Xilinx Virtex-II Pro [81]. Moreover, some types of FPGAs can be reconfigured without affecting system operation on the chip. Precompiled reconfigurations are stored in non-volatile memory, so when faults occur, system just reload a new configuration [34, 45]. It also positively affects on the adaptivity: whenever the adaptation is required, FPGA can be partially or completely reprogrammed without stopping the operation of the system.

The approaches described above provide a hardware support for adaptation. These tools, however, do not provide a software support for the adaptation. Moreover, it is not possible neither to control the adaptation process, nor to tune the adaptation mechanism. In this work we provide a flexible, yet simple mechanism that can be utilized by a programmer to handle, to control, and to tune the adaptation process of the application.

2.3 Language Support for Adaptive Mainstream Systems

In this section we outline a language support for implementing adaptive systems. This area has been exhaustively investigated in traditional computation platforms. Naturally, an adaptive software is implemented via dedicated software architecture styles or a middleware. This approach is the one of the most common approaches to the software adaptivity in WSNs, since it provides an architecture that is tailored to the specific problem and platform. However, a number of dedicated programming techniques provide more flexible adaptation mechanisms, such as Context-Oriented Programming [64], Aspect-Oriented Programming [41], and Metaprogramming [74], which are widely used to create self-adaptive software. These techniques are based on such features of the language as dynamic memory allocation, computational mirroring, and a source-code modification at run-time, which are generally not available on the resource and time-constrained CPSs. Moreover, the solutions based on these techniques are designed for the application where the adaptation time is not critical. Hence, these solutions are hardly applicable in CPSs as is.

2.3.1 Architecture-Based Adaptation

In the area of self-adaptive systems, researches mostly focus on the adaptation from an architectural viewpoint. For example, Oreizy et al. [57] proposed a framework that allows the user to change the architecture of the software dynamically. The architecture of the software is represented in the framework as the *Architectural Model*. The *Architecture Evolution Manager* (AEM) maintains the correspondence between the implementation and the model. Every time the model is changed, the AEM determines if the modification is valid. The AEM uses an explicit knowledge of the architecture constraints. If the modification violates these constraints, the

AEM rejects the change. Otherwise, the model is altered and the dedicated mapping is used to modify the implementation correspondingly.

The Rainbow framework [28] provides an infrastructure to support the architecture-based self-adaptation. The framework provides two elements to support a self-adaptivity: the *adaptation operator* and the *adaptation strategy*. The adaptation operators determines a set of actions that are performed to make desirable adaptations in the system, while the adaptation strategy prevents the system from the undesirable behavior. Offering a general approach, the framework can also be tailored to a specific class of systems. For example, in web-based client-server systems, the adaptation strategy focuses mostly on the analysis of the client’s requests and the time needed for the response. Differently, in the videoconferencing system, an adaptation strategy optimizes a video quality according to the available bandwidth.

A number of middleware systems has been developed to support the implementation of self-adaptive systems. For example, Sadjadi et al. [65] proposed ACT (Adaptive CORBA Template) that allows a programmer to modify the behavior of the CORBA applications at run-time. In CORBA the interaction between components occurs via requests. ACT intercepts and adapts these requests in order to adapt the behavior of the application. Another middleware, Madam [51] is able to detect context changes and to make decisions about the adaptation necessity. This middleware adopts *utility functions* that are used to establish general goals, and to implement the architectural changes.

Even though an architecture-based adaptation is widely applied, the architectures are usually tailored to specific problems. The more general approaches include the support of the adaptation on the language level, providing developers with more flexible adaptation mechanisms, as we show in the next section. Nevertheless, these approaches do not consider the limitations of resource-constrained cyber-physical systems.

2.3.2 Context Oriented Programming

Context Oriented Programming (COP) [64] was recently introduced to provide language abstractions for adaptive software. In traditional languages, the adaptation would be achieved by *if then else* structures, making the application increasingly complex. COP addresses this issue by modularizing behavioral variations. Additionally, COP provides explicit mechanisms for enabling these variations. Two main notions of COP are *i) a context* that represents the information about the current state of the environment, and *ii) a layered function* that changes its behavior depending on the current environmental state.

COP has been implemented in a great variety of languages, such as Java, Python, Ruby, JavaScript, Common List and Scheme [3]. Since these languages lack a dedicated COP support, researches borrowed COP concepts to better address specific design issues. Despite the variety, these solutions share common concepts: *i) an abstraction for behavioral variations*: typically a first-class entities, which can be assigned to parameters, or returned by functions; *ii) an activation* mechanism defines how the behavioral variations are enabled or disabled from within the application; and *iii) a behavior combination* supports the reaction to the several activated behavioral variations. Commonly, the behavioral variation is enabled through the activation of the corresponding context.

ContextJ [64] is one of the implementations of COP paradigm. In ContextJ, the first-class entity is the *Layer* – an object that represents a single context. Every software component that provides a context-dependent functionality should define a context-specific implementations of each layered function. The dedicated key-word *with* activates layers within the scope of this key-word. Thus, context activation affects only the current thread, and the behavioral variation propagates its effect only within the activation block.

A different approach, however, is used in Ambience language [31], where the context spreads its effect across the whole application. The context in this approach is represented by a hierarchy tree that consists of smaller *subcontexts*. Every path in this tree reflects a contextual information. Whenever an environment is changed, the topology of the tree changes as well. The context in Ambience is passed as an additional argument to every method definition. Thus, every method is aware of the global context of the application and behaves accordingly. With this approach the activation of the variation can be triggered for the whole program. Yet, the activation is completely asynchronous and conflicting variations can be activated.

ContextErlang [69] is another example of COP implementation, yet with a different context activation mechanism. Being build on top of Erlang language, ContextErlang uses messages to activate contexts. Thus, the latter are activated asynchronously. Moreover, behavioral variations are activated on per-object basis, so each object can be controlled individually. This also prevents an unintended adaptation propagation.

Embedded Domain-Specific Language (EDSL) [49] seamlessly embed abstractions for building context-aware application in Haskell. Every context-aware computation in EDSL is represented as a function with implicit arguments and inference rules. These functions automatically derives the contextual dependencies from the *Global Knowledge Base* component. The latter can be updated by using the *parametrized mondas* that inject a new context to the knowledge base. This approach does not only hides the context management from the programmer, but also helps to achieve the static verification of the context-aware application.

Generally, COP offers a better support for modularity [67], and is specifically designed to allow programmers to implement their own adaptation mechanisms. These languages, however, are designed for traditional platforms and can not be applied in resource-constrained systems like WSNs or time-critical systems, as we argued above.

2.3.3 Aspect Oriented Programming

Aspect-Oriented Programming (AOP) is a programming technique that focuses on providing mechanisms for modularization of orthogonal functionality called crosscutting concerns. This technique allows a programmer to change the behavior of the software at run-time by activating the corresponding *aspects*. There are several AOP frameworks, such as CaesarJ [36], Prose [60], JAC [59], and AspectWerkz [8]. These approaches offer general adaptation mechanisms that can be used to implement the adaptive software.

For example, J-EARS [6] is a Java-based framework for developing and managing the application that perform the adaptation autonomously using AOP. The framework supports a deployment of a control-loop to the application. Firstly, the *Application Manager* allows a programmer to define adaptation policies. Secondly, the *Aspect Manager* provides the interface to add, to remove and to configure the software components, which are responsible for the adaptation process, such as monitors, decision makers and executors. Finally, the *Communication layer* binds the application with the managing tool.

TOSKANA [21] framework is devoted to the run-time adaptation of the UNIX system kernel. A developer uses TOSKANA to deploy the aspects, which can be activated within the operating system kernel to modify its behavior while the system is running. The framework extends the standard C language with the *pointcut*, *before*, *after*, and *around* statements. These statements allow the programmer to define a crosscutting concern – e.g. by using *pointcut* key-word – and then specify actions before, after, or both before and after the pointcut functionality. This framework can be used not only for the adaptation, but also for the monitoring of the running applications. In this case, the monitoring functionality crosscuts the main kernel logic.

CaesarJ [36] provides dedicated *wrappers*: dynamic extensions of other classes in the system. Wrappers are instantiated by the *WrapperConstructor*, which takes the objects as parameters and returns the corresponding wrapper object. The latter includes and extends the functionality of both objects. Moreover, the wrapper object is a singleton – i.e. whenever a *WrapperConstractor* is used with the same arguments, it returns the same wrapper object. This approach does not only provide the adaptation mechanism by dynamically extending the functionality of the objects, but also guarantees the uniqueness of such extensions.

These frameworks, however, are designed for large and complex systems like OS kernels, client-server applications, or web-applications. Despite the usage of AOP for resource-constrained CPSs would be quite an overkill, it has to be *ported* to the CPSs because of the reasons we described before.

2.3.4 Metaprogramming

Metaprogramming is a programming technique that exploits such tools for dynamic adaptation as: computational reflections, dynamic link libraries (DLL), and source-code modifications at run-time. Dowling et al. [18] have concluded that computational reflections offer a significant advantage in implementing the adaptive paradigm, however, with a non-negligible performance overhead. The metaprogramming is an extremely general mechanism, hence, it has to be tailored to the application-specific purposes.

Some languages natively offer a metaprogramming mechanism. Otherwise, developers implement such abilities manually. For example, TRAP/J [66] allows the developer to add the dynamic adaptation to every (even to an existing one) Java application without modifying the original source-code. The tool wraps the original classes at instantiation time. Then, using a meta-object protocol, method calls are redirected to the delegated objects. Thus, the adaptive behavior is achieved at run-time. TRAP/C++ [25] is an implementation of the same concepts in C++ language.

Metaprogramming is delivered not only by language abstractions, but also by the dedicated middleware systems. OpenORB [7] and CARISMA [12] are such examples, as they do not provide a meta-object protocol, but do provide an API for the already implemented software to make it adaptive at run-time. For example, CARISMA exploits a meta-object model to provide the application engineers with customizable services. The resulted applications are described as a set of customized *services* that are provided by the middleware; *policies* that define how to deliver the services; and *configurations* – the requirements that must be hold in order for the policy to be applied.

In our work we focus on the devices with a very limited amount of memory and without memory protection. Hence, the dynamic memory allocation (and, thus, the run-time instantiation) is not safe, and metaprogramming technique can not be applied.

Part I

Adaptive Software in Wireless Sensor Networks

CHAPTER 3

Design Concepts and Programming Support

The software for WSNs, as shown in Section 2.1, is continuously confronted with unpredictable environmental dynamics and changing requirements. These demand WSN software to *adapt* to the situations exhibited by the real world. Programmers, however, are missing a dedicated support for implementing adaptive software for WSNs. Lack of this support makes the implementation of adaptive software difficult in general, and even more under resource constraints, which are common characteristics of WSNs.

In this chapter we describe our work towards the full support for the developing the adaptive WSN software. In the following Section 3.1, we investigate the problem in the details, while discussing our solution in Section 3.2. We mapped our solution into a handful of design concepts described in Section 3.3. These concepts are implemented in our own language CONESC, as we described in Section 3.4. Moreover, our concepts allow the WSN software to be verified even before the actual execution, as we show in Section 3.5. In Section 3.6, we provide a support of all these features in a set of tools that we developed for application designers and software engineers.

```
1 module ReportLogs {
2   uses interface Collection;
3   uses interface DataStore;
4 }implementation {
5  int base_station_reachable = 0;
❻  event message_t Beacon.receive(message_t msg) {
❼   if (call Battery.energy() <= THRESHOLD)
8     return;
9   base_station_reachable = 1;
10  call GPS.stop()
11  call BaseStationReset.stop();
12  call BaseStationReset.startOneShot(TIMEOUT);}
⓭  event void BaseStationReset.fired() {
⓮   base_station_reachable = 0;}
⓯  event void ReportPeriod.fired() {
16  switch (base_station_reachable){
17   case 0:
18    call DataStore.deposit(msg);
19   case 1:
⓴    call Collection.send(msg);}}}
```

Figure 3.1: *Example nesC implementation of adaptive functionality:* several orthogonal functionality become entangled an need to share global data.

3.1 Problem

Consider the wildlife tracking application [58] described in Section 2.1.1. The challenge arises when a programmer and a designer take into explicit account every possible situation in the design and the implementation of the WSN software. *Multiple combined dimensions* – e.g., physical location, battery level, and health conditions – concurrently determine how the software should adapt its functionality. Typical approaches often result in entangled implementations that are difficult to debug, to maintain, and to evolve. With the growing number of dimensions and their combinations that affect the software operation, the software model grows exponentially, and the implementation quickly turns into "spaghetti code" [24].

To illustrate this problem, we implemented the wildlife tracking application in one of the most popular languages for WSNs called nesC [29]. Figure 3.1 shows a greatly simplified example of how the adaptive functionality is implemented. The code implements only one aspect of the adaptive functionality in the wildlife tracking application: to send readings to the base station whenever it is reachable, or to store them locally otherwise.

The base station continuously broadcasts beacons to signal its presence. Whenever the node receives this beacon, the event on line ❻ is fired, and the global state **`base_station_reachable`** is changed. If the node has not received the beacon for a considerable amount of time, the event on line ⓭ is fired and the global state is reset. The adaptive behavior is implemented on line ⓯, where the programmer uses the **`DataStore`** interface to deposit the readings if the base station is not reachable. Otherwise, the readings are sent directly to the base station over the **`Collection`** interface on line ⓴.

However, multiple orthogonal concerns are overlapping the main functionality, making different aspects dependent on each other. For example, the adaptation process between the lines ⓯ and ⓴ rests in the same module as the operating mode on lines ❻ and ⓮. Both parts share the global state **`base_station_reachable`**, managing of which is entirely on programmer's shoulders. Moreover, if the battery is low, it is better to delay the transmission of the data, until the battery is charged from the solar panels. This check has to be performed before changing the operating mode, as on line ❼, and the check is mixed with the code that changes the mode itself. Finally, the specific implementation of adaptive functionality, such as the external interfaces **`Collection`** and **`DataStore`**, are visible from the caller module, further coupling the two.

In this situation it is difficult to maintain, to debug, and to evolve the software. Modification of the code in one place would likely lead to the need of changing the code in several other places. It is of course possible to partly ameliorate the problem by alternative implementations. However, we found many similar implementation patterns by looking at publicly available implementations, e.g., within the TinyOS codebase [78].

3.2 Solution

We address this problem by providing Context-Oriented Programming (COP) [64] in resource-constrained WSN software. COP brings a strict separation of adaptive functionality. This is achieved by two key notions: *i)* different situations that software must adapt to are mapped to different *contexts*, and *ii)* context-dependent behaviors are encapsulated in *layered functions*, that are the functions whose behavior changes – transparently to the caller – depending on the context.

COP has already proven its effectiveness in creating context-aware software, such as text editors [39], and user interfaces [40], using COP extensions of popular high-level languages [68]. Currently, COP remains far from being applicable in WSNs, since most implementations of COP relay on such capabilities of the languages as run-time instantiation, polymorphism, and inheritance [68]. The absence of such abilities in WSN languages prevents COP to be applied there as is.

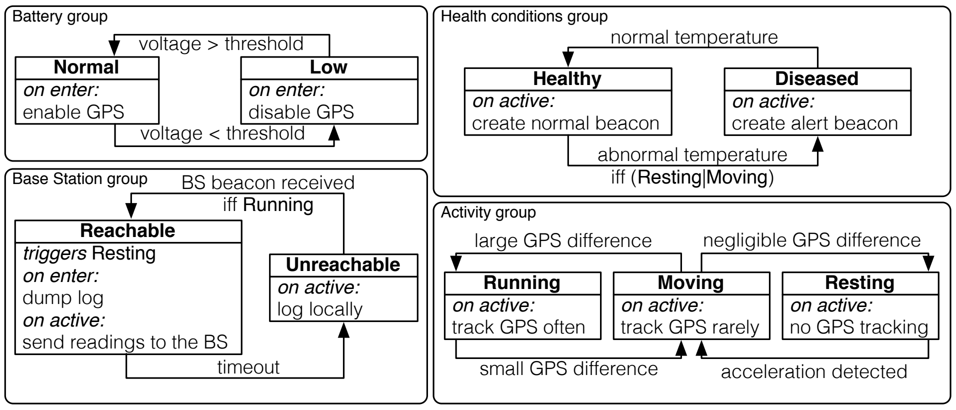


Figure 3.2: *Context-oriented model of wildlife tracking application.*

To address this issue, we borrow concepts from COP and provide context-oriented programming abstractions for WSN software. In Section 3.3 we illustrate design concepts, which are independent of a specific programming language. We aim to decouple the abstractions form their implementation in a concrete language, thus facilitating their application in any language. One such realization is CONESC, that we describe in Section 3.4. This is our own context-oriented extension to nesC. We have chosen nesC because of its popularity, stable toolchain, and node-centric view [52]. The latter will most likely be used in the applications, such as the wildlife tracking application, where the adaptation process is local to the individual nodes. In Section 3.5 we provide an approach to the verification of the adaptive WSN software model.

In order to cover the whole development process from the early design to the real deployment, we provide a set of tools that are usable by both: a designer and a programmer. For the early software design, in section 3.6 we provide our own editor where the designer can use our concepts to manually build and to automatically verify the model of the WSN software. Finally, our translator seamlessly converts the CONESC sources to the plain nesC sources and compiles them, yielding a binary that is ready for deployment.

3.3 Design

We illustrate language-independent design concepts and provide a foundation for importing COP on target languages. We refer to the wildlife tracking application, while explaining how our concepts can improve a design process.

Figure 3.2 exemplifies how our concepts can be applied to the software design for the wildlife tracking application. The diagram reflects variations in functionality depending on the battery level, the base station communication range, and the animal's health conditions and activity.

We introduce two core notions: *i)* a context, and *ii)* a context group. A context represents an individual situation the software must adapt to, and contains a behavioral variation corresponding to this situation. As environment changes, the software adapts by activating a corresponding context. A context group is a collection of contexts sharing common characteristics – e.g. belonging to the same environmental dimension.

The contexts within the group define an individual behavioral variation. For example, in the wildlife tracking application, the software must report readings from sensors, and the report function has to behave differently depending on whether the base station is reachable. If so, a programmer activates the *Reachable* context within the *Base Station* group, and data is sent directly to the base station, as shown in the diagram in Figure 3.2. Otherwise, the programmer activates the *Unreachable* context, as the software must save readings locally.

The contexts are tied with transitions that express context triggering conditions. For example, as it is displayed in Figure 3.2, within the *Base Station* group the transition from the *Reachable* to the *Unreachable* context is triggered if there is no beacon from the base station received. In that case the software must adapt the behavior, that is, save reports locally, instead of relaying them to the base station over the radio.

A designer can also add dependencies to transitions. For example, if the sensor reads an abnormal body temperature, it might indicate that an animal is *Diseased*. But if the animal is diseased, it is most probably moving slightly or not moving at all. In any case, the *Diseased* animal should not be *Running*. If it is not the case, the situation indicates that the model is incorrect, and probably has some flaws in the design. To this end, the designer adds a dependency *iff(Resting|Moving)* to the transition from the context *Healthy* to the context *Diseased* in Figure 3.2. With this dependency we are sure that the transition will be triggered only if the animal is either *Resting* or *Moving*.

Context activation can also trigger a transition in another context group. For example, in our example the base station is always static. Having this knowledge, a programmer may want to reduce the power consumption by disabling GPS module whenever the node is nearby a base station. The module, however, is used in the *Activity* group, as it is shown in Figure 3.2, and should not be disabled outside the group to avoid access conflicts. Hence, the programmer activates the *Resting* context, as shown in Figure 3.2, to disable the GPS module whenever the base station is *Reachable*. Indeed, if the animal is within the range of the static base station, it is most likely *Resting*, since the base station is deployed close to nests.

Contexts must not necessarily provide a complete behavior variation by their own, but they can serve to other functionality; for example, by proving context-dependent data. One such example is the *Health Conditions* group displayed in Figure 3.2. Using a body temperature sensor, the software detects if the animal is *Healthy* or *Diseased*. The behavioral variations generates two types of beacons used as "proximity" sensor to detect contacts between animals. If the animal is *Diseased*, additional information is added to the beacon to track the disease spreading. Both types of beacons are then broadcast via the radio stack.

The outlined concepts are sufficient to implement an environment-dependent functionality in a large class of applications of WSNs, as we show next. At the same time, unlike the majority of similar approaches [52], our approach is largely decoupled from a concrete language implementation, which emphasizes its generality. Although, we use nesC language as a host for our approach, our concepts can be easily applied to other languages as well. For example, within functional languages such as Regiment [56] or Flask [48], variations of programmer-defined functions can be simply enabled through a proper syntax.

3.4 Programming

We illustrate how our concepts are rendered within CONESC: a context-oriented extension to nesC. We describe the notions of a context module and a context configuration in Section 3.4.1, and show how a programmer can use these notions to specify an adaptive behavior of the application in Section 3.4.2. Finally, in Section 3.4.3 we describe how the programmer can gain a fine-grained control on the adaptation process in CONESC.

3.4.1 Context Groups and Contexts

A context group in CONESC is an extension of a nesC configuration. Programmers use context groups to declare layered functions, which include behavioral variations. Figure 3.3 shows the *Base Station* context group, where the layered function **`report`** is declared on line ❷ with the key-word **`layered`**. The key-word **`contexts`** on line ❹ is followed by the contexts included in the group. There are three such contexts, where the **`is`**

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

Figure 3.3: *Context Group in* CONESC.

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

Figure 3.4: Unreachable *context.*

```
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(message_t msg){
❾    call DataStore.deposit(msg);}}
```

Figure 3.5: Reachable *context.*

`default` modifier on line ❺ points to the context that will be activated at start-up. The modifier `is error` on line ❻ indicates an *Error* context, the purpose of which we discuss later in Section 3.4.3. This modifier is not mandatory, and an *Error* context is generated automatically if not indicated.

A context in CONESC extends a nesC module by providing a context-dependent implementation of a layered function declared in the context group. Only one context can be *active* at a time providing a corresponding implementation of the layered function. For example, Figures 3.4 and 3.5 show CONESC snippets for the contexts *Reachable* and *Unreachable*, which

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

Figure 3.6: *Base-station context manager.*

are displayed in Figure 3.2. They provide different implementations of the layered function **`report`**. If the base station is *Reachable*, and, thus, the corresponding context is active, the layered function on line ⓫ of Figure 3.4 transmits messages over the radio. Differently, the code on line ❾ of Figure 3.5 deposits messages in internal memory.

Optionally, a programmer can specify additional instructions, such as initialize variables or enabling/disabling hardware components upon entering/exiting contexts. For example, on entering the context *Reachable*, the programmer may want to disable the GPS-module, since the location can be obtained from the static base station. To do so, the programmer puts this functionality into the body of the predefined **`activated`** event, as on line ❺ of Figure 3.4. Similarly, the programmer can put clean-up code that will be executed on exiting the context by implementing the event **`deactivated`**, as on line ❼ of Figure 3.4.

3.4.2 Execution

Figure 3.6 shows how a programmer can detect and activate a proper context. Anywhere in the code the programmer can trigger a transition between the contexts in the group by using the key-word **`activate`**. For example, according to the diagram displayed in Figure 3.2, on line ❺ of Figure 3.6 the programmer activates the *Reachable* context as soon as a beacon is received, and activates the *Unreachable* context whenever the timeout fires on line ❾. Each time the active context is changed, the context-specific implementation of the layered function **`report`** is used.

The calls to the layered functions occur transparently to the caller w.r.t the available contexts and independently of which context is active. Figure 3.7 shows one such example: a programmer just declares the usage of the *Base Station* group on line ❷, and uses then the function **`report`** on line ❺. The net advantage here is that the context's activation/deactivation

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

Figure 3.7: *Caller module.*

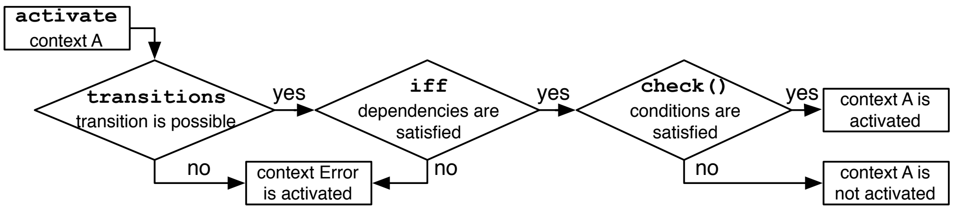


Figure 3.8: *Context activation rules.*

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

Figure 3.9: Resting *context.*

logic is decoupled from the actual usage of the context-dependent functionality, and can be implemented even in a separate module.

Every time the active context in the group is changed, the predefined event **`contextChanged`** is fired. A programmer can implement the body of this event, as on line ❻ of Figure 3.7, to add additional instructions depending on the activated context. Within the body of this event the programmer uses the parameter, which holds the current active context as on line ❼.

3.4.3 Transition Rules

A programmer needs to take care of context transitions, as the latter can drastically change the application's behavior. To better support programmers in doing so, ConesC provides several checking stages, as shown in Figure 3.8. A successful check allows the transition to continue, while a failure leads either to the activation of the *Error* context or to the cancellation of the transition.

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

Figure 3.10: Low *context*

The first check in Figure 3.8 allows only *possible* transitions to pass. Let us refer to the context diagram in Figure 3.2: within the *Activity* group, it is only possible to transit from the *Resting* to the *Moving* context. In CONESC, this also can be specified by the programmer by using the key-word **`transitions`**, as on line ❷ in Figure 3.9. An attempt to initiate a transition from the context to the one that is not explicitly mentioned in the list of possible transitions means a significant hardware or software failure, and leads to the activation of the *Error* context. Within the latter a programmer can implement the run-time exception handling.

There may also exist relations between several context groups. For example, within the *Base Station* group in Figure 3.2, a transition from the *Unreachable* to the *Reachable* context is only meaningful if the context *Running* is active, indicating that an animal was actually moving when the node received a beacon. These inter-group relations are covered by context dependencies in our design, as shown on line ❷ of Figure 3.5. The key-word **`iff`** is optionally used to indicate the full name of the required context. This rule is verified by the second check in Figure 3.8, leading to the *Error* context in case of violations, and giving programmers a chance to handle the exception.

The last check in Figure 3.8 considers violations to "soft" requirements, which are not necessarily indicating a design or an implementation flaw. For example, before activating the *Reachable* context displayed in Figure 3.2, a programmer may want to check that sufficient energy is available to transfer the data stored in the local memory to the base station. Should this not be the case, the activation of the *Reachable* context will be deferred until the next attempt to activate this context. To implement this check, the programmer has to implement the body of a predefined command **`check`** and put necessary conditions there, as shown on line ❽ of Figure 3.4. If **`check`** returns false, the transition does not occur, and the system remains in the previous context.

On the other hand, the programmer may want to proactively initiate a context transition as a result of another context transition. Let as assume, for example, that the system is in the *Reachable* context, and the transition to the context *Low* has been initiated in the *Battery* group of Figure 3.2

meaning the energy is running low. Since the nodes are equipped with solar panels, as we described in Section 2.1.1, it is probably better to suspend the radio communication until the battery is charged again. Our design allows programmers to express this processing by using the **`triggers`** key-word, as shown on line ❷ of Figure 3.10. The **`triggers`** key-word indicates the list of contexts that have to be activated as a result of an enclosed context being activated. The same checks shown on Figure 3.8 are applied to this type of transitions.

3.5 Verification

As the WSN software continuously adapts to the environmental dynamics, it is crucial to exhaustively verify this software. The increasing complexity of the software leads to the increase in the number of possible situations the software must adapt to. These situations are difficult to recreate, hence, it is not feasible to test the software by running it on the device. Off-line model-checking, however, can help find the flaws of the model even before the real deployment. With this technique, the model can be exhaustively verified against different combinations of environmental situations.

Our concepts allow a designer to create a high-level model of the software for WSNs, but the semantics of the resulted models is not directly supported by any of the existing verification and model-checking tools. The transformation from a context-oriented model to an equivalent state-machine could help here, but it will be a fairly time consuming process if performed manually. Indeed, the context-oriented model consists of the context groups – a set of state machines that operate concurrently. Hence, the state machine that is equivalent to the context-oriented model will be a Cartesian product of the context groups, and an increase in the number of contexts leads to an exponential explosion of states. Moreover, the context groups interact via dependencies and triggers, further complicating the equivalent state machine. Thus, even a fairly simple context-oriented model will require a significant amount of time when transformed into an equivalent state-machine manually.

We address this problem by providing an approach that can be utilized to automatically verify the context-oriented models. We describe the general rules of transformation, as well as a mapping algorithm from a context-oriented model to a state-machine.

Algorithm 1: Mapping algorithm.

```
   Input: $Model$
   Output: $States$
1  create list $States$
2  for each $Group$ in $Model$ do
3      $States \leftarrow$ CartesianProduct($Group, States$)
4  end
5  for each $State$ in $States$ do
6      for each $transiton$ in $State.outgoingTransitions$ do
7          if $transitions.target.triggers$ is not empty then
8              Retarget($transition, States$)
9          end
10     end
11 end
12 for each $State$ in $States$ do
13     if $State$ has no incoming transitions then
14         $States$ remove $State$
15     end
16 end
17 Function CartesianProduct($Group, States$)
18     create list $newStates$
19     for each $Context$ in $Group$ do
20         for each $State$ in $States$ do
21             create $newState$
22             $newState.name \leftarrow (Context.name + State.name)$
23             create list $Transitions$
24             Build($Transitions, Context, State, newState.name$)
25             Build($Transitions, State, Context, newState.name$)
26             $newState.outgoingTransitions \leftarrow Transitions$
27             create list $Triggers$
28             $Triggers$ append $Conetxt.triggers$
29             $Triggers$ append $State.triggers$
30             $newState.triggers \leftarrow Triggers$
31             $newStates$ append $newState$
32         end
33     end
34     return $newStates$
35 end
36 Procedure Build($Transitions, SourceState, TargetState, newSource$)
37     for each $transition$ in $SourceState.outgoingTransitions$ do
38         $target \leftarrow transition.target$
39         $dependency \leftarrow transition.dependency$
40         $newTarget \leftarrow (target.name + TargetState.name)$
41         if $newTarget$ satisfies $dependency$ then
42             $Transitions$ append NewTransition($newSource, newTarget, transition$)
43         end
44     end
45 end
46 Function NewTransition($source, target, parent$)
47     create $transition$
48     $transition.source \leftarrow source$
49     $transition.target \leftarrow target$
50     $transition.event \leftarrow parent.event$
51     return $transition$
52 end
53 Procedure Retarget($transition, States$)
54     for each $State$ in $States$ do
55         if $State.name$ contains all $transition.target.triggers$ then
56             $transition.target \leftarrow State$
57         end
58     end
59 end
```

3.5.1 Mapping Algorithm

In our transformation algorithm 1, we take a *Model* as an input. The *Model* is a set of *Groups*, where every group is a set of *Contexts*. Within the single group, contexts are connected by *labeled transitions*, where a label represents an environmental event and an optional dependency. Only one context in the group can be active at a time. Whenever the event occurs and the dependencies are satisfied, the transition executes and another context within the group becomes active. Each context also contains optional *triggers* – whenever the context is activated, it can trigger a transition in another context group. Thus, each *Context* in the algorithm 1 has a set of properties:

- *Context.name* holds a name of the context.
- *Context.triggers* holds a list of names of contexts from other groups, that have to be activated whenever current *Context* is activated.
- *Context.outgoingTransitions* holds a list of outgoing transitions.

For example, in the context *Reachable* in figure 3.2, *Reachable* is a context name; *Resting* is a trigger; and *Reachable→Unreachable* is an outgoing transition.

Each *transition* in the algorithm 1 has the following properties:

- *transition.source* holds the source of the transition.
- *transition.target* holds the target of the transition.
- *transition.event* holds the environmental event that triggers the transition.
- *transition.dependency* holds an optional dependency: transition can only be executed if the dependency is satisfied.

For example, in the transition *Healthy→Diseased* displayed in Figure 3.2, *Healthy* is a source and *Diseased* is a target; *abnormal temperature* is an event; and *iff(Resting|Moving)* is a dependency.

The output of the algorithm 1 is a list of *States* of the resulted state-machine. For example, the state-space of the *Health Conditions* and the *Activity* groups is displayed in figure 3.11. We outline a set of rules that are used to build this state-space.

Rule 1. In a single context group only one context can be active at a time. Hence, a single state of the system is a combination of the contexts, which

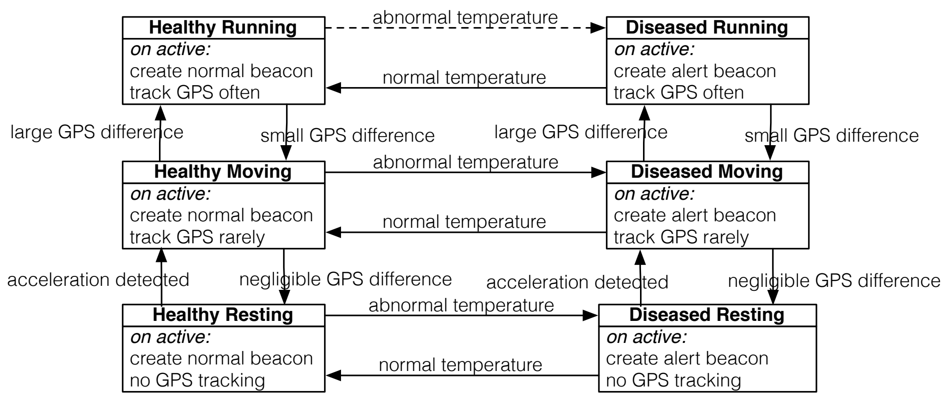


Figure 3.11: *State-space of the* Health Conditions *and the* Activity *groups.*

are currently active. Thus, the name of each resulting *State* is a concatenation of the context names.

The concatenation in the algorithm 1 is represented by the operation +. For example, the name of the state *HealthyRunning* in figure 3.11 is a result of the operation *Healthy+Running*, where *Healthy* is a context from the *Health Conditions* group and *Running* is a context from the *Activity* group. Applying the *Rule 1* to these groups, we obtain the Cartesian product of these groups, as shown in figure 3.11.

The state-space in the algorithm 1 is built incrementally: at each step, line ❸ , we calculate a Cartesian product of the next context group and the current state-space, according to the *Rule 1*. Thus, a name of each new state defined at line ㉒ is a concatenation of the names of the context and a state from the current state-space. At the end of the algorithm, the names of the resulting states will be the combinations of the names of the contexts involved, as in figure 3.11.

Rule 2. Each resulting *State* inherits all the outgoing transitions of the contexts involved in the concatenation process in the *Rule 1*. The names of targets of the inherited transitions are modified by appending the names of the involved contexts.

Rule 3. The dependencies of the transition prevent some of the inherited transitions from being added to the list of the outgoing transitions of the resulting *State*.

For example, the *HealthyRunning* state, which is shown in figure 3.11, inherits both outgoing transitions: *Healthy* $\rightarrow$ *Diseased* and *Running* $\rightarrow$ *Moving*. Based on these, we correspondingly create the transitions *Heal-*

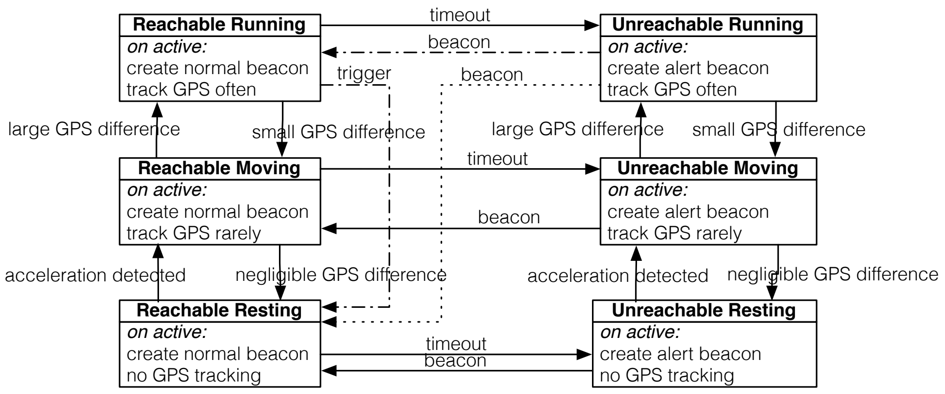


Figure 3.12: *State-space of* Base Station *and* Activity *groups.*

thyRunning → *DiseasedRunning* and *HealthyRunning* → *HealthyMoving*. The dependency *iff (Moving|Resting)* of the transition *Healthy* → *Diseased* means that the target state of any inherited transition should strictly contain either *Moving* or *Resting*. As the transition *HealthyRunnig* → *DiseasedRunning* is inherited from the transition *Healthy* → *Diseased* and its target state does not contain neither *Resting* nor *Moving*, the transition *HealthyRunning* → *DiseasedRunnig* is not added to the set of the outgoing transitions of the *HealthyRunning* state. This transition is shown as a dashed line in figure 3.11.

In the algorithm 1, the *Rule 2* and the *Rule 3* are implemented in the procedure BUILD on line 36 . This procedure is called twice: to scan transitions from the *Context* at line 24 and from the *State* at line 25 . This procedure *i)* scans outgoing transitions from the *SourceState* at line 37 ; *ii)* builds a new target at line 40 ; and *iv)* builds and appends a new transition to the list of $Transitions$ at line 42 if the *dependency* is satisfied at line 41
.

Rule 4. Each resulting *State* inherits all the triggers of the contexts involved in the concatenation process in the *Rule 1.* The new target's name is modified by appending the name of the context that contains the trigger. The trigger of the context implicitly removes some of the inherited transitions and adds a new one to the list of the outgoing transitions of the resulting *State*.

For example, the state-space of the *Base Station* and the *Activity* groups is shown in figure 3.12. The target of the transition *UnreachableRunning* → *ReachableRunning* is a concatenation of the *Running* and the *Reachable* contexts. As shown in figure 3.2, the latter has the *trigger Resting*, which is inherited by the *ReachableRunning* state. Hence, whenever the transition *UnreachableRunning* → *ReachableRunning* is executed, it should result into the state *ReachableResting*, as shown by the dash-dotted lines in figure 3.12. To do that we retarget this transition to the *ReachableResting* state instead of the *ReachableRunning* state, and the transition becomes *UnreachableRunning* → *ReachableResting*, as shown by a dotted line.

In the algorithm 1 the *Rule 4* is implemented on lines 28 and 29 . Every new state inherits the *triggers* from both: the context and the state. Then on line 5 , the triggers are applied to the state-space. There, we look for a transition, which has a target with a non-empty list of *triggers*. Each found transition is retargeted then to a state, the name of which contains all the context names from the *triggers* list, as shown on line 56 . After this procedure, the old target may happen to be without any incoming transitions. Hence, at line 12 of our algorithm 1 we scan the state-space and remove unreachable – i.e., with no incoming transitions – states.

The algorithm 1 is fairly straight-forward, but it gives an intuition how a context-oriented model is mapped to a state-machine. It also helps one to make some flaws in the model more evident. For example, in Section 3.3, we argued that the *Diseased* animal should not *Running*. To prevent both the *Diseased* and the *Running* contexts to be active in the same time, we added a dependency *iff(Resing|Moving)* to the transition from the context *Healthy* to the context *Diseased* in figure 3.2. After applying the mapping rules to the *Health Conditions* and the *Activity* groups, in figure 3.11 we notice, that the state *DiseasedRunning* is reachable in the state-space. This means that the system can be in a state, when both the *Diseased* and the *Running* contexts can be active at the same time, and this is a flaw. Later in this work, we provide the tools that can automatically find this kind of flaws.

In Section 3.3, we also suggested how a programmer can use the notion of *trigger* to disable GPS module and to save the energy by triggering the *Resting* context in figure 3.2 whenever the *Reachable* context is activated. As a result of such activation, we observe a transition *UnreachableRunning* → *ReachableResting*, as shown by a dotted line in Figure 3.12. This transition implicitly implies two concurrent transitions in the initial model: *Unreachable* → *Reachable* and *Running* → *Resting*. The latter, however, does not exist in the initial model displayed in figure 3.2. Thus, the resulted state-machine has revealed an illegal transition *Running* → *Resting*. This kind of transitions are automatically revealed, as we show in the next section.

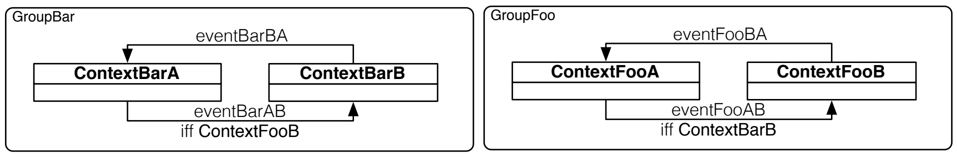


Figure 3.13: *Example of a part of a large and complex context-oriented model.*

Building an equivalent state-machine from the initial context-oriented model can also make another flaws more evident. In large models it is usually difficult to track all the dependencies of the transitions. Moreover, it is not trivial to find mutually contradictory dependencies. For example, let us assume a context group *GroupFoo* where we have a transition *ContextFooA* $\rightarrow$ *ContextFooB* with a dependency *iff ContextBarB*, as in figure 3.13. We also consider another context group *GroupBar* where we have a transition *ContextBarA* $\rightarrow$ *ContextBarB* with a dependency *iff ContextFooB*. It results in a non-evident deadlock: the context *ContextFooB* is activated only if the *ContextBarB* is active and vice versa, so the system is stuck in the *ContextFooA* and the *ContextBarA*. If there are only two contexts in the *GroupFoo*, as in our example, the current situation leads to another non-evident problem: the context *ContextFooB* and *ContextBarB* become unreachable. As we show in the next section, these issues can be revealed automatically even before implementing and deploying the actual software to the real device.

These issues are hard to find in a context-oriented model; an equivalent state-machine, however, can make them more evident. Even though, we provided an algorithm to build this state-machine, it is difficult to follow this algorithm manually due to an exponential state explosion. Addressing this challenge, we create a tool support described next.

3.6 Tool Support

To better support developers in designing, implementing and deploying the context-oriented software, we provide set of tools. The whole work-flow of the development process with our tools is shown in Figure 3.14. The **Designer** uses our tool GREVECOM, which is described in Section 3.6.1, to build the model of the software, to verify it (as described in Section 3.6.2), and to generate the source-code templates in CONESC. Then, the **Programmer** modifies the templates by adding the actual application specific functionality. The resulting source code is handed then over to our **Translator**

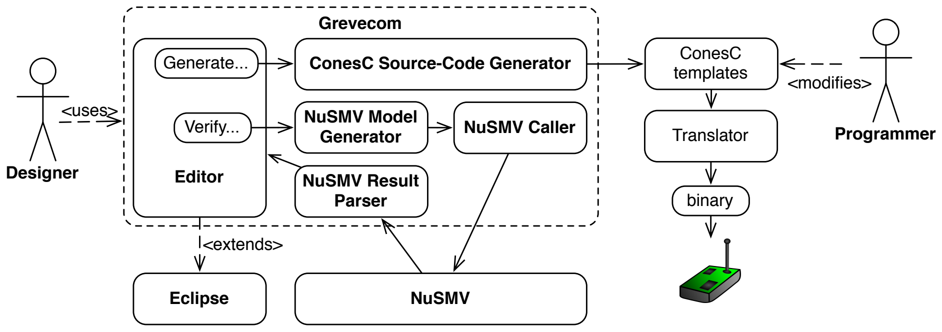


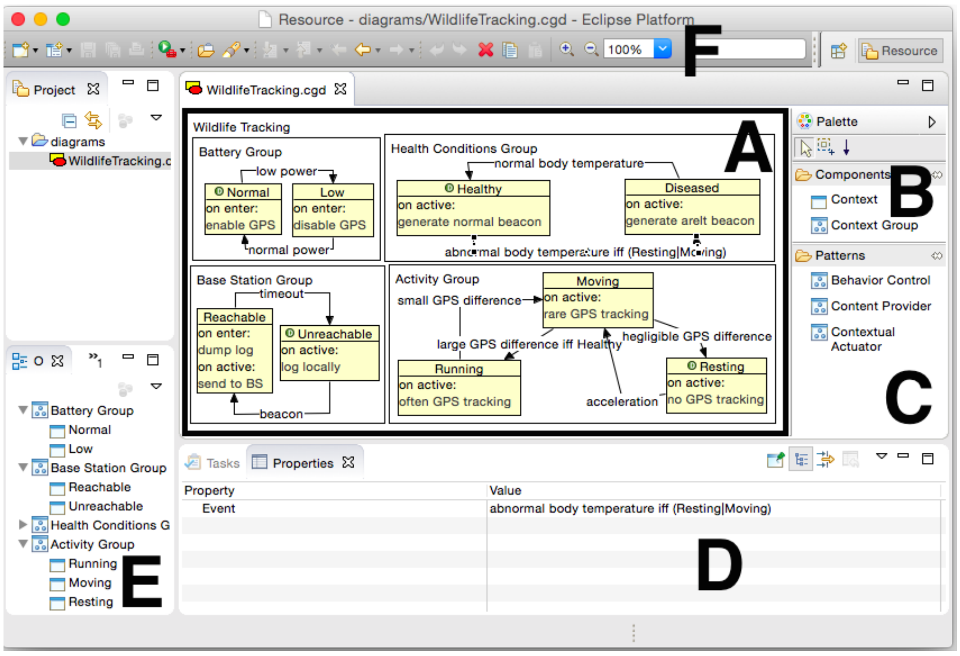
Figure 3.14: GREVECOM *architecture and work-flow.*

tool. The latter translates the CONESC sources to the plan nesC, as described in Section 3.6.3, and uses the nesC tool-chain to produce the binary, which is ready for the deployment.

3.6.1 Grevecom

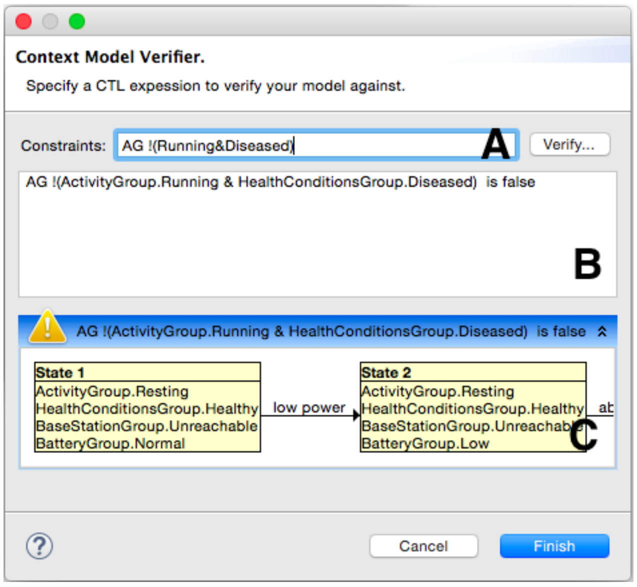
The Graphical Editor and Verifier for Context-Oriented Models (GREVECOM) allows a designer to build a context-oriented model, to verify it against environmental evolutions and user-defined constraints, and to generate the templates for a programmer. The usage diagram of the GREVECOM is shown in Figure 3.14. The **Designer** uses the **Editor** – i.e., an Eclipse plug-in, which allows the designer to build the model, to verify it and to generate CONESC templates. The **NuSMV Model Generator** module is called whenever the **Designer** wants to *Verify* the model, as shown in the diagram in Figure 3.14. This module builds a state-machine for the NuSMV model-checker from the context-oriented model built in the **Editor**, and via the **NuSMV Caller** calls the external **NuSMV** model-checker. The results are captured by the **NuSMV Result Parser** module, which interprets them in terms of the original context-oriented model and forwards them to the **Editor** to display. Whenever the **Designer** chooses the *Generate...* option, the **CONESC Source-Code Generator** module takes the context-oriented model as an input and creates CONESC templates.

The main element in the GREVECOM is a canvas, as shown in Figure 3.15 **A**, where the designer can put *context groups* and *contexts* from the components palette **B**. The palette **C** also contains the most recurring patters that are described in the details later. Contexts can be bound by transitions with labels. Each label contains a representation of the environmental event and an optional dependency. Events are important in the



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

Figure 3.15: GREVECOM *main window.*


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

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

Figure 3.16: GREVECOM *verification wizard.*

model, but their implementation is application specific. Hence they are ignored when the templates are generated. Dependencies, however, appear in the templates after the source-code generation. Properties tab **D** allows a designer to change the properties of any object on the canvas. For example, to add an event or a dependency to a transition. The hierarchy tree **E** helps a designer to track the structure of large models, when it is difficult to display the whole model in the canvas **A**. The customized tool-bar **F** has a similar purpose: the designer can zoom in/out the model if it is too large. Apart from the properties **D**, a context can be modified in the dedicated context edit wizard, where a designer can change the *name*; actions on *enter*, *active*, or *exit*; make the context *default* or *error*; and add *triggers*.

A designer can verify the model by choosing the command *Verify...* from the menu. The verification occurs in a new window, as in Figure 3.16, where the designer can type a list of CTL constraints divided by a semicolon, as in Figure 3.16 **A**. For example, a constraint *AG !(Running&Diseased)* verifies that the model in Figure 3.2 can not be in the *Running* and *Diseased* contexts simultaneously. Independently of the user-defined constraint, as we show below, a model will be also verified against automatically generated specifications such as: violation of transitions between contexts; reachability of contexts; and deadlocks. The results of the verification are printed in the frame **B**. Any found counterexample is graphically represented as a sequence of activated contexts in the frame **C**.

3.6.2 NuSMV Translation

During the verification, GREVECOM builds a state-machine from a context-oriented model using the rules described in Section 3.5. The algorithm described there gives an exponential time of model generation, but in NuSMV we can encode a state as a vector, where each variable represents a state of a single context group. Similarly to the algorithm 1, GREVECOM scans each *Group* in the *Model* and generates a symbolic variable, which represents a state of a single context group. The possible assignments of this symbolic variable are the names of the contexts within the corresponding context group. Then, GREVECOM builds a set rules for the symbolic variable to change its value. To do that, GREVECOM scans triggers and outgoing transitions of each context.

This approach to the generation of the state-machine is computationally simpler than the one described in Section 3.5. Indeed, let us take N_{CG} as a number of context groups and M_C as a number of contexts per group. The maximum number of outgoing transitions per context is $M_c - 1$, and

```
1  MODULE main
2   VAR
❸    bsg_state : {Reachable, Unreachable};
❹    ag_state : {Running, Moving, Resting};
5  //...
6    event : {timeout, beacon, //...
7    };
8   ASSIGN
9    next(bsg_state) :=
10    case
11     bsg_state = Reachable & event = timeout : Unreachable;
⓬     bsg_state = Unreachable & event = beacon : Reachable;
13 //...
14    esac;
15   next(ag_state) :=
16    case
⓱     bsg_state = Reachable : Resting;
18 //...
19    esac;
20 //...
21 SPEC AG(EF bsg_state=Reachable)
22 SPEC AG(bsg_state=Reachable->EF bsg_state=Unreachable)
23 SPEC AG(ag_state=Running->AX(ag_state=Running|ag_state=Moving))
24 //...
```

Figure 3.17: *Generated context-oriented NuSMV model.*

the maximum number of triggers per contexts is $N_{CG} - 1$. As GREVECOM scans outgoing transitions and triggers, the complexity is the following: $T = N_{CG} * M_C * (N_{CG} - 1 + M_C - 1) = O(N_{CG}^2 * M_C^2)$. We evaluate the time needed for the generation of the state-machine form a context-oriented model in Chapter 4.

A snippet of the generated model is displayed in Figure 3.17. For the *Base Station* group we create a symbolic variable **bsg_state**, as on line ❸. There are two contexts in the group: *Reachable* and *Unreachable*. Hence, the possible assignments of the symbolic variable **bsg_state** are the values **Reachable** and **Unreachable**. Similarly, the symbolic variable **ag_state**, which is declared on line ❹, represents the state of the *Activity* group. Based on the transition *Unreachable→Reachable* in Figure 3.2, we build a transition rule on line ⓬ in Figure 3.17: the variable **bsg_state** changes its value from **Unreachable** to **Reachable**, whenever the **event** equals to **beacon**. In the context *Reachable* in Figure 3.2 there is a trigger *Resting*. Based on this information we build another rule on line ⓱ in Figure 3.17: whenever the **bsg_state** is assigned to **Reachable**, the value of the **ag_state** is changed to **Resting**.

Along with building a state-machine for NuSMV, we also build a set of constraints to check typical, but non-evident flaws in the model. We add to the model the following requirements:

- *Reachability*: each context must be activated at least once. For example, to make sure that the context *Reachable* in Figure 3.2 must be activated at least once, we build a constraint on line ㉑ in Figure 3.17: the value **`Reachable`** must be assigned to the **`bsg_state`** eventually in the future.

- Absence of *deadlocks*, meaning that at some point of the execution, there are no sequences of events that lead to the activation of any other context in a group. To avoid this in the *Base Station* group, for example, we have to make sure that after the activation of the context *Reachable* in Figure 3.2, the context *Unreachable* can be eventually activated. The constraint on line ㉒ in Figure 3.17 represents this requirement: if the **`bsg_state`** has a value **`Reachable`**, it must be changed – e.g., assigned with the value **`Unreachable`** – eventually in the future.

- All the transitions must be *legal*. The transitions defined in the context-oriented model are legal. However, triggers implicitly create transitions that contradict to the explicitly defined transitions. We call them *illegal transitions*, and they may become evident after generating the state-machine from the context-oriented model. For example, from the *Running* context the only transition to the *Moving* context is possible, as illustrated by Figure 3.2. This requirement is presented on line ㉓ in Figure 3.17: if the **`ag_state`** has a value **`Running`**, the next value must be either **`Moving`** or the same as before (**`Running`**).

For all of the contexts from all of the context groups, GREVECOM generates the rules and the constraints as described above. Should any of the constraints be violated, NuSMV provides a counterexample, and GREVECOM displays it as a sequence of the events and states in a diagram. The latter allows a designer to understand where the flaws appear in the model and why. After fixing the flaws, the designer uses GREVECOM to automatically generate the CONESC templates and hand it over to the programmer, as in Figure 3.14, for implementing the actual software.

3.6.3 Translator

After the software being implemented, the programmer has to compile and to deploy the software. To this end, we develop a translator, which converts CONESC sources to plain nesC, and uses the nesC tool-chain to compile

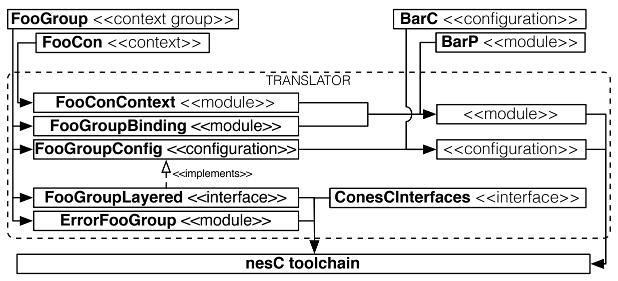


Figure 3.18: CONESC *translation to nesC code for a generic* **`FooGroup`** *context group and an individual context* **`FooCon`**.

a final binary. Our translator performs two passes through the input code. First, it reads the main **`Makefile`** to recursively scan the component tree. Based on the information gained during the first pass, including the list of every context and context groups defined in the code, the translator parses every input file to convert the CONESC code to plain nesC and to generate a set of support functionality. The resulting sources are then compiled using the standard nesC toolchain.

Figure 3.18 illustrates the details of the operations during the second pass. Generally, the input to the translator includes four types of components: context groups, contexts, nesC configurations, and nesC modules where CONESC constructs appear. In Figure 3.18, context groups and contexts are represented by a sample **`FooGroup`** context group and an individual **`FooCon`** context, whereas nesC configurations (modules) with CONESC constructs are represented as **`BarC`** (**`BarP`**).

Based on every context group, we generate a custom nesC module, such as **`FooGroupBinding`** in Figure 3.18, that implements the dynamic binding of layered functions to the active context. This module is a part of a configuration, such as **`FooGroupConfig`**, also automatically generated. This configuration implements a nesC interface our translator produces, such as **`FooGroupLayered`**, that exports the layered functions defined in the group. Optionally, an error context is also generated in plain nesC, as indicated by **`ErrorFooGroup`** in this case, if the programmer does not provide one. Each individual context is translated to a corresponding nesC module with the proper interfaces to be wired within the aforementioned configuration, as in the case of **`FooConContext`** for the **`FooGroupConfig`**.

At this stage, context and context groups disappeared, yet CONESC constructs, such as **`activate`**, may still appear within the source code. Our

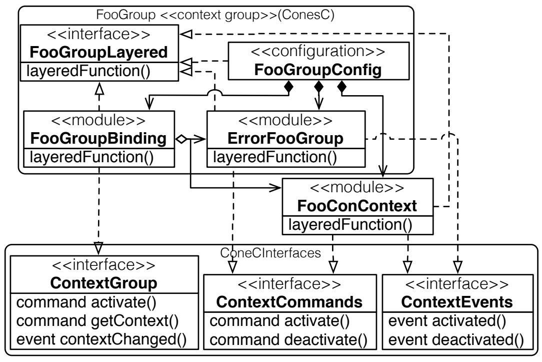


Figure 3.19: *Hierarchy of generated components.*

translator converts these constructs to functionally-equivalent nesC code both in the nesC files generated out of context groups and individual contexts, and in the plain nesC files that possibly includes them, such as **`BarC`** and **`BarP`** in Figure 3.18. The resulting sources are then wired to generic interfaces that define the standard commands and events in CONESC, such as **`contextChanged`** for context groups, as in Figure 3.7, and **`activated/deactivated`** for individual contexts, as in Figure 3.5 and 3.3. The result is plain nesC code that can be given as input to the nesC toolchain.

The hierarchy of the components generated based on the **`FooGroup`** is displayed in Figure 3.19. The **`FooGroupLayered`** is a basic interface for all the generated components, as it declares the *layered functions*. The main generated component is a configuration, such as the **`FooGroupConf`**, which instantiates and links together the contexts and the control modules. The control module **`FooGroupBinding`** implements the **`FooGroupLayered`** interface to enable a behavioral variations, and the **`ContextGroup`** interface to provide the operations over the *contexts*, e.g., the **`FooConContext`**. The **`ErrorFooGroup`** module is generated if it was not declared in the **`FooGroup`**. Each *Context* is translated into a module, as, for example, the **`FooConContext`**, which implements the **`ContextCommands`**, the **`ContextEvents`**, and the **`FooGroup`** interfaces mentioned above.

Our translator is implemented using JavaCC [38]. Three aspects are worth noticing. First, the generated code is still human-readable, and a programmer can modify it to implement fine-grained optimizations. Second, the code is completely hardware-independent. Therefore, hardware compatibility is the same as the original nesC toolchain, allowing us to support a wide range of WSN platforms and not to modify our translator due to hardware idiosyncrasies. Second, the whole translation process is only seemingly straightforward. Rendering the logic embedded within the CONESC abstractions does require a fairly sophisticated processing. To give an intuition, we measured the size of the CONESC implementations of the application we use for evaluation, described next, against the size of the nesC implementations output by our translator. On average, we observe three times as much lines of code in the automatically-generated nesC code.

3.7 Summary

In this chapter we provided the full support for the developing of the adaptive software for WSNs. Our concepts described in Section 3.3 allow a designer to create an elegant context-oriented model of the adaptive WSN software. Being implemented in a particular language, e.g., in our own context extension to nesC called CONESC, these concepts give a programmer a powerful tool to express the adaptive behavior of the software, and to gain control over the adaptation process, as we have shown in Section 3.4. Moreover, in Section 3.5 it is shown that with our concepts, the designer is able to verify the context-oriented model against the environmental evolutions and the user-defined constraints even before implementing and deploying the actual software. Finally, in Section 3.6 we provided the programming tools, that the designer and the programmer can use to design, to implement, and to compile the adaptive WSN software.

CHAPTER 4

Early Experience and Evaluation

In this chapter we describe our early experience of using our design concepts. To do this, we developed a number of WSNs applications and discussed them it in Section 4.1. In the same section, we also outlined the typical patterns that recurrently appear independently of the application. Based on these application, in Section 4.2, we evaluated our approach by performing a number of experiments.

4.1 Early Experience

In order to demonstrate the generality of our approach, we applied our concepts to the developing of three representative applications. In addition to the wildlife tracking application, we implemented the adaptive protocol stack and the smart-home controller, which are described in Section 2.1.1. We describe the implementation of each application in details in Section 4.1.1, and then, in Section 4.1.2, show typical patterns that are naturally exhibited when our concepts are applied.

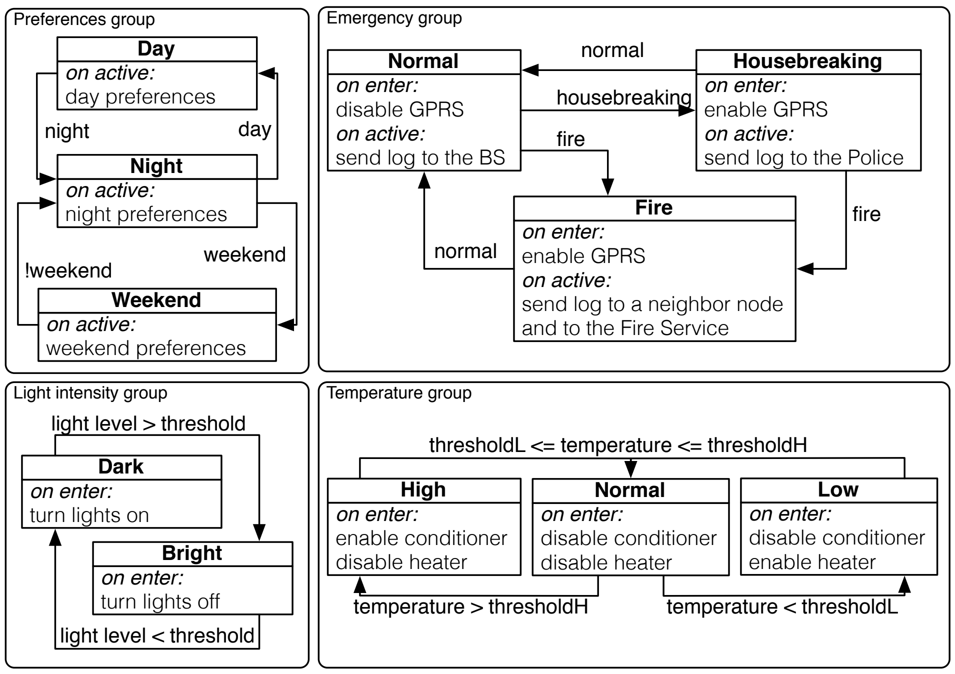


Figure 4.1: *Smart-home controller design.*

4.1.1 Applications

In Section 2.1.1 we discussed example adaptive applications for WSNs. In this section we provide a context-oriented design of these applications.

Smart home. As we discussed in Section 2.1.1, the smart home controller regulates temperature and lighting conditions in a house relying on both the environmental and the user-defined information. The context-oriented design of the smart-home controller is shown in Figure 4.1. The functionality is driven by user-defined preferences dependent on the current context. The *Preferences* group contains contexts, which provide different operating parameters depending on day/night and weekends vs. working days conditions. Such parameters, compared against current light and temperature readings, drive the context transitions within the *Temperature* and *Light* context groups. Any time a transition in these groups occurs, the node operates actuators to control HVAC and lighting system. The controller also exploits smoke sensors, fire sensors, and cameras to detect fire and housebreaking situations. It notifies the user about the emergency and sends data to the police or firemen, depending on the situation. The latter functionality

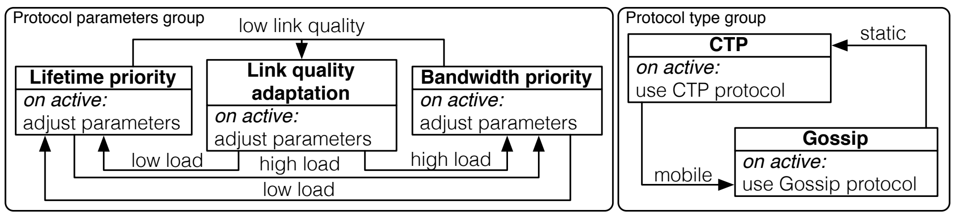


Figure 4.2: *Adaptive protocol stack design.*

is encapsulated in the *Emergency* context group.

Adaptive protocol stack. In Section 2.1.1, we argued about the need of adaptivity in this application. Indeed, the software must maintain two orthogonal functionalities: *i)* switching between the protocols, and *ii)* tuning these protocols depending on the situation. Figure 4.2 illustrates the context-oriented design of this application. The *Protocol Type* group contains two different implementations of the routing layer: *CTP* and *Gossip*. The latter is preferred in mobile networks, while the former is activated whenever the node is static. Orthogonal to the protocol switching, the *Protocol Parameters* group enables the fine-grained tuning of the parameters of the protocols depending on the priorities in the network.

As the provided example applications illustrate, even a fairly complicated and entangled functionality can be designed and structured in an effective and elegant way with our concepts. Moreover, as we figured out, our concepts exhibit particular design patterns, which we discuss next.

4.1.2 Emerging Patterns

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

Behavior control. Programmers may employ a single context group to specify different *behaviors* for the same high-level functionality. One such example is the *Base Station* group in Figure 3.2, which includes two different behaviors for the functionality to report contact logs to the users. The functionality itself is exported by one or more layered functions defined on the group. The chosen behavior is then determined by activating a single context within the group.

We found similar designs in other applications as well. In the adaptive

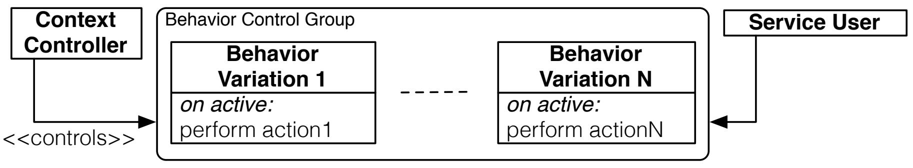


Figure 4.3: *Behavioral control pattern.*

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

Figure 4.3 shows an abstract view of such commonly recurring pattern. In addition to the context group exporting the adaptive functionality and the single contexts therein, programmers also define an additional *context controller* component, which activates the single contexts within the group depending on the situation. Figure 3.6 shows a CONESC example for the wildlife monitoring application. Similar designs apply to the smart-home controller and the adaptive protocol stack as well.

Content provider. Different from the behavior control pattern, which provides non-trivial context-dependent processing, we observe cases where context-dependent *data* is offered to other functionality with little to no processing involved. In the wildlife monitoring application, for example, the *Health Conditions* group in Figure 3.2 provides differently formatted beacons to the radio driver for broadcast transmissions. Layered functions are, in this case, defined for the group merely to retrieve the context-dependent data.

In this case as well, we notice the same pattern in other applications. In the smart-home controller, for example, a context group is defined to manage the user preferences depending on day vs. night. These data are simply retrieved differently from a data storage by two different contexts modeling day or night situations. Whatever user preference is to be considered at a given point is then handed over to the control loop in charge of setting the functioning of the climate systems.

As shown in Figure 4.4, this pattern's structure differs from that of behavior control in that the role of the *controller* component is often fairly trivial. In the smart-home controller, for example, the controller compo-

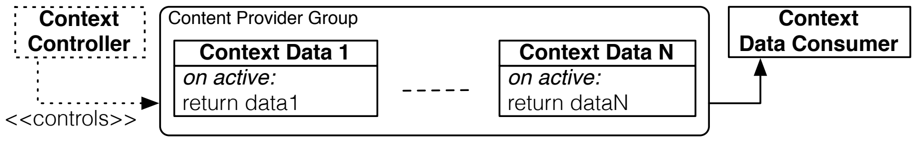


Figure 4.4: *Content provider pattern.*

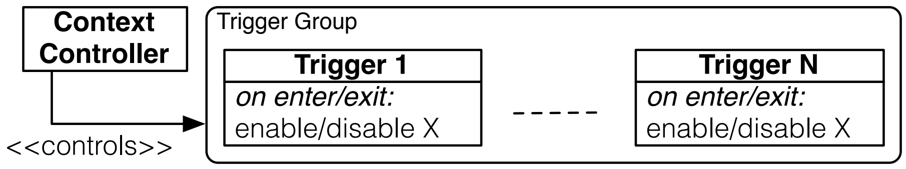


Figure 4.5: *Trigger pattern.*

nent is simply based on the time of the day. On the other hand, the component consuming the context-dependent data plays a key role. Indeed, while functionality structured according to behavior control can be considered stand-alone, the context provider needs to be tailored to the data consumer.

Trigger. We also recognize designs where single contexts are used only to trigger specific operations when entering/exiting, but no significant context-dependent functionality or data is offered as the context remains active. One example in the wildlife monitoring application is the *Battery* group in Figure 3.2. The included contexts are used to enable/disable the GPS sensor depending on battery levels, but no other functionality is provided to other components. In this case, layered functions are often not defined, in that the predefined `activated` and `deactivated` events within the single contexts suffice.

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

In more general terms, a *context controller* component is present in this case as well to drive the context transitions in the group, as shown in Figure 4.5. However, unlike the other patterns, there is no other significant component that either uses context-dependent functionality or consumes context-dependent data. The functionality is mostly self-contained.

4.2 Evaluation

To evaluate our approach, we implement each application described in Section 4.1.1 using either CONESC or nesC. The resulting implementations are functionally equivalent. Based on these applications we evaluate our approach along seven dimensions. In Section 4.2.1, we analyze how tight the components are *coupled* in our implementations. Coupling generally determines the ease of maintenance and evolution of software [43]. We also use code metrics to asses the *complexity* of the software in Section 4.2.2, that often affects a system's reliability and ease of debugging [43]. Based on illustrative case studies, in Section 4.2.3 we measure efforts required for *evolving* the software. In Section 4.2.4 we measure the *performance overhead* when using CONESC in terms of MCU and memory penalty. Section 4.2.5 quantifies the time needed to *generate* a model for NuSMV from a context-oriented model. In Section 4.2.6 we calculate the time required to *verify* the context-oriented model of the applications. Finally, Section 4.2.7 shows how the verification can be *scaled* on very large models.

4.2.1 Coupling

There are seven types of coupling between software modules, according to Stevens et al. [75]. In Table 4.1 we summarize these types. It is generally known that the tightest is coupling, the more difficult is extending, maintaining, and debugging the software. We manually inspect the source code, to investigate the types of coupling in CONESC against nesC implementations.

Results. Our investigations are shown in Table 4.2. As we observe, CONESC implementations are generally less coupled as compared to their nesC counterparts. In CONESC different behavioral implementations are encapsulated in different contexts allowing programmers to avoid *Content* coupling. Contrary, nesC programmers are forced to expose internal module information to bind command calls or events to different modules, making modules operations dependent on each other. For the same reason, nesC programmers are using global variables to switch between several behavioral variations depending on the situation. This creates *Common* coupling, which is avoided in CONESC implementations, since the necessary functionality is automatically generated by our translator. Finally, *Control* coupling is avoided in CONESC as well. This is a result of dynamic module

Table 4.1: *Coupling types.*

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

Table 4.2: *Coupling comparison:* CONESC implementations save most types of coupling that are unavoidable in nesC.

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

binding driven by the context transitions, which has to be manually coded in nesC.

While both CONESC and nesC allow a programmer to avoid *Stamp* and *Message* coupling, *Data* and *External* couplings are not avoided neither in CONESC nor in nesC. They both rely on typed interfaces, thus, in both

Table 4.3: *Complexity comparison:* CONESC yields simpler implementations that are easier to debug and to reason about.

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

implementations different modules use a common data format, and these types of coupling are unavoidable.

4.2.2 Complexity

We estimate the complexity of the implementations by measuring the number of variable and function declarations in each module. These are generally considered as intuitive indicators of code complexity [43]. Complexity is also a function of a number of states in which a program can find itself [43]. A state here is any possible assignment of values to the program variables. Thus, the number of states must be computed by looking at the different combinations of values of variables for every possible execution.

To perform the latter analysis, we use SATABS [14] – a model-checking tool for C programs. It performs an off-line verification against user-defined assertions. To do so, it searches all relevant program executions, where the assertions hold. The result of such verification process are counterexamples, which show the execution where one or more assertions do not hold. As a complementary information, SATABS also returns the number of states it explored in the program. With a specific configuration, we forced SATABS to explore *all* possible program executions and to return the total number of distinct states of the program. We use SATABS on a per-function basis, implementing empty stubs to replace the code that we cannot process with SATABS, e.g. hardware drivers.

Results. Table 4.3 illustrates our results. CONESC shows a significant

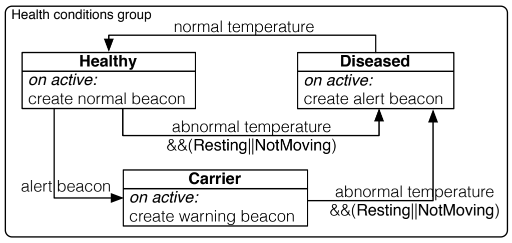


Figure 4.6: *The extended* Health Conditions *group.* We added the Carrier context.

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. On the other hand, in nesC this requires to define a set of global variables to check what variation needs to be triggered depending on the situation. As a result, the number of per-function states programmers have to deal with also drastically decreases in CONESC, making the implementations easier to understand.

The number of lines of code are roughly comparable in both implementations: nesC and CONESC. It should be also noted that, however, the size of generated source code is trice as larger, which indicates the "expressive power" of the abstraction, besides the intuition of the complexity involved in the translation. It also demonstrates that our few simple concepts do capture a significant portion of processing.

4.2.3 Software Evolution

Due to changes in requirements, WSN software should constantly change. The better is an implementation modularized, the easier are the modifications, since the changes affect only a small part of the code [43]. For each application we estimate the effort to modify the CONESC implementation compared to the nesC counterpart. We study three types of modifications: adding a new context group, adding a new context, and removing a context.

To estimate the effort to remove the context, we modify the scenario of the adaptive protocol stack. Say, the programmer wants to remove one of the CTP parameter sets after testing, since this set has shown a bad performance. To study an addition of the new context, we extend the wildlife tracking application to the case, where it is necessary to track a spread of the disease. To do this, the programmer adds a new context *Carrier* to the *Health conditions* group, as shown in Figure 4.6. This context generates a beacon for an animal who was in contact with a diseased one, but shows no

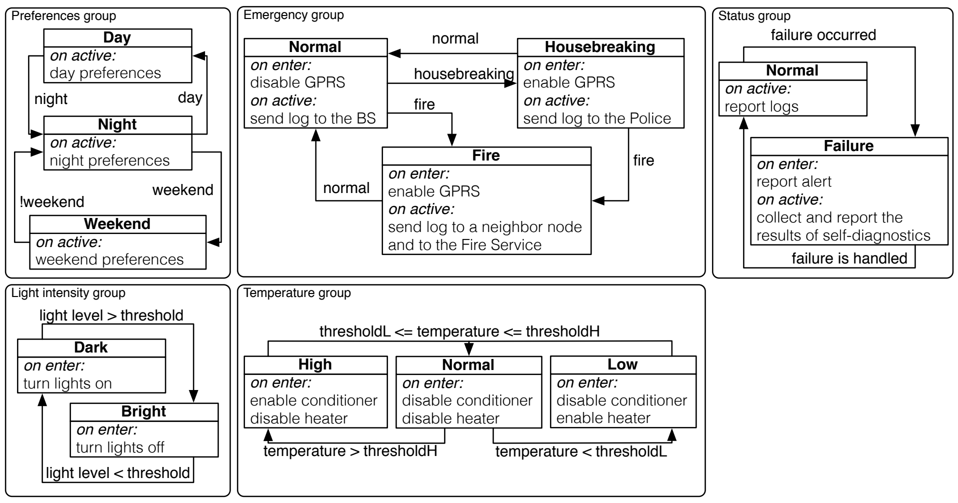
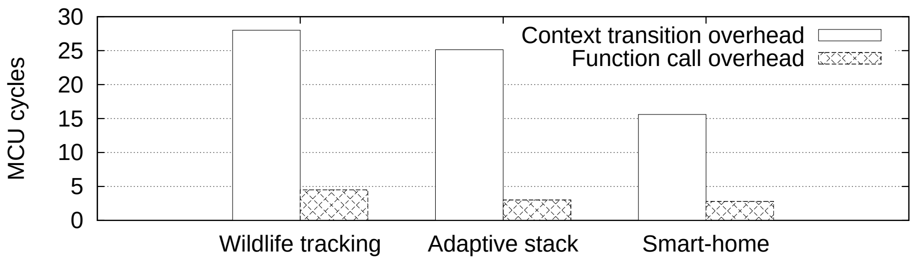


Figure 4.7: *The extended model for the* Smart Home *application.* We added the Status context group.

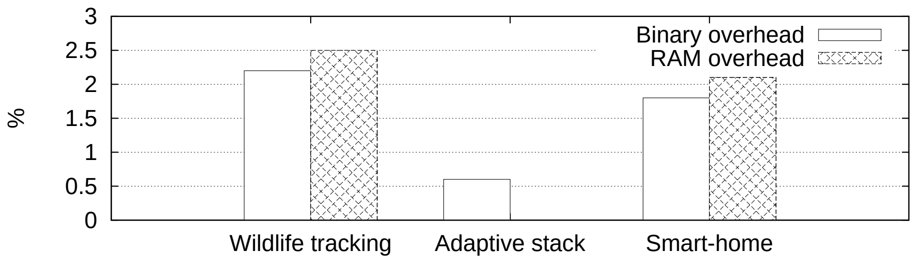
symptoms yet. We study the addition of a context group by extending the smart-home controller scenario. We consider a case, when the programmer wants to add a periodic run-time check to the controller execution loop. Should a potential failure be discovered, the controller changes its behavior. To this end, the programmer adds an entirely new context group *Status* with contexts *Normal* and *Failure*, as illustrated in Figure 4.7.

Results. Removing a context in CONESC implementation of the adaptive protocol stack only requires to modify three sequential lines of code, while in nesC the developer must modify several lines of code scattered throughout the main module. To add a context in the wildlife tracking application, it is necessary to modify only 5 lines of code in CONESC implementation. To implement the same extension in nesC, besides the code modifications, the programmer must add new global states, further complicating the control flow. Adding a new context group in the smart-home controller requires to modify about 40 lines of code in CONESC, besides the implementations of the new contexts. However, in nesC, in addition to the same modifications in the source code, we need to add more global states, again making the implementation more entangled.

It is worth noticing, that the efforts to apply context-unrelated changes are the same in both CONESC and nesC implementations.



(a) *MCU overhead.*



(b) *Memory overhead.*

Figure 4.8: *MCU and memory overhead:* the resource usage penalty for using CONESC is almost negligible.

4.2.4 MCU and Memory Overhead

The advantages brought by our approach come at the cost of system overhead. To assess it, we measure 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 estimate the former using nesC and GNU-C tool-chains, while we estimate the latter by using the MSPSim MSP430 emulator [22]. As the executions are deterministic, the experiments yield the same results independently of the number of executions.

Results. The results are shown in Figure 4.8. The MCU overhead for layered functions varies from 2 to 5 MCU cycles depending on the application. This is negligible overhead in terms of energy consumption, since the simplest operations in TinyOS – turning on/off LED – consumes 8 MCU cycles. The context transition overhead is slightly larger, but in the same order of magnitude. This is a result of the activation rules described in Section 3.4.3: additional MCU cycles are needed to check if the transition is possible, then to check dependencies, and finally to execute the body of the **`check()`**.

Most importantly, the memory overhead is 2.5% in the worst case. The complexity of the application, however, largely dictates the corresponding memory penalty. For example, the wildlife tracking application, being the most complicated in terms of contexts, context changes, and data processing, shows the highest memory overhead. The overhead for the adaptive protocol stack is negligible, since a nesC programmer would use the same set of variables compared to the one generated by our CONESC translator automatically.

4.2.5 Model Generation Time

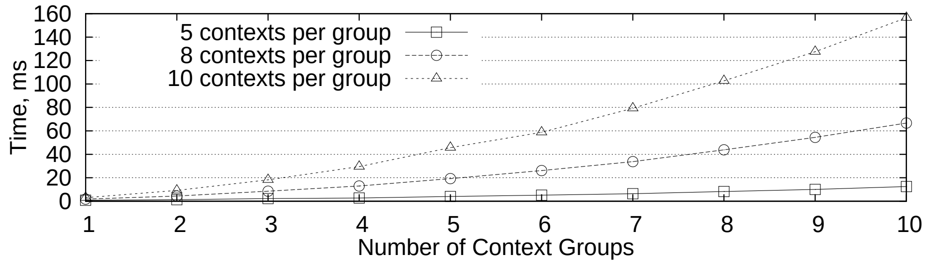
We measured the time needed for generating a model for NuSMV symbolic checker, with respect to the number of context groups and contexts per group. We use a machine with Intel Core2Duo and 4Gb RAM. During the experiment we launch the algorithm with different configurations – e.g. a set of constant N_{CG} (number of context groups) and M_C (number of contexts per group). The generating algorithm, described in Section 3.6.1, is implemented in Java. For each fixed configuration we execute the algorithm measure the CPU time of the current thread in java virtual machine using the dedicated library *ThreadMXBean*. We perform a set of calls, until we reach a standard deviation $\leq 5\%$.

Our example applications contain two to four context groups with two to three contexts included in a single group. For evaluation purposes, however, we generate artificially synthesized models. The configuration of the models is the following:

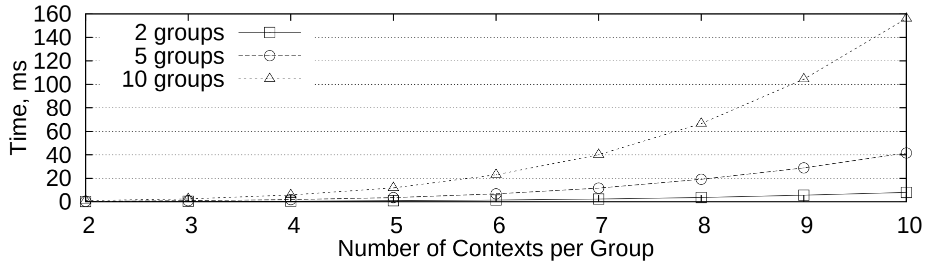
- The maximum number of the transitions within the context group is achieved when each context is connected with every other context in the same group.
- The maximum number of triggers in the context is achieved when the contexts triggers a single context in every other context group.
- These numbers for the whole model are maximized with an equal number of contexts per group.

Thus, in the worst case, each group has an equal number of contexts; and each context is bounded by a transition with every other context in the group.

The typical device a CONESC program is written for has 10kB of RAM, and does not involve any significant processing. Thus, we do not expect designers to deploy a lot of contexts and context group, and, hence, we limit our measurements with boundaries: $M_C \in [2, ..., 10]$ and $N_{CG} \in [1, ..., 10]$.



(a)



(b)

Figure 4.9: *Generation Time for NuSMV models.* The generation time remains lower than 160ms even on large numbers.

With $M_C = 10$ and $N_{CG} = 10$ the model is fairly unrealistic and we consider it only for comparison.

Results. As we observe in Figure 4.9, both graphs have a perfect shape of quadratic function, just as we previously calculated: $T = O(N_{CG}^2 * M_C^2)$. The generating time remains lower than 160 ms even for fairly large models, hence we pay a little price of generation time.

4.2.6 Verification Time

In this section we show how much time is required to verify context-oriented models of the real applications described in Section 4.1.1. The experiment is executed on the machine described in Section 4.2.5. For each application we automatically built a state-machine that is handed over to the NuSMV model-checker. To measure the CPU time required for the verification, we launched NuSMV with UNIX command *time*, which returns the CPU-time upon the execution. We launch NuSMV ten times and calculate the standard deviation. If it is $> 5\%$, then we relaunch NuSMV for another ten times, until the obtained standard deviation is $\leq 5\%$.

For each application we measure the verification time of the correct

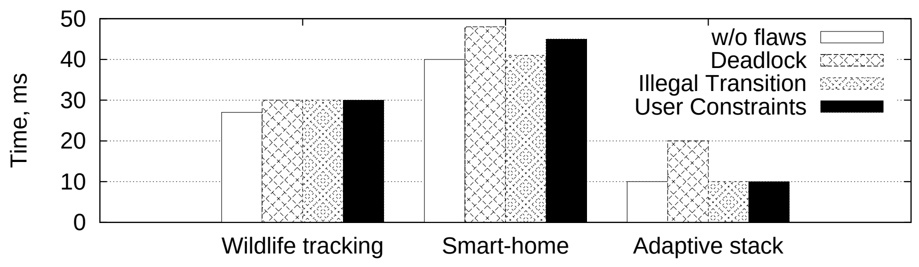


Figure 4.10: *Verification Time.* Generation of counterexamples leads to the increase in verification time.

model; of the model with different types of flaws such as deadlocks and illegal transitions; and of the model with user-defined constraints. As we mentioned in Section 3.6.1, for every a set of constraints is automatically built to check different flaws in the model. Along with the verification time, in this experiment we also measure, how the additional user-defined constraints affect the verification time. In our experiments, we did not observe a strong connection between a constraint type and the verification time. Hence we focus only the type of flaws.

To measure the relationship between the type of flaws and the verification time, we slightly modified each context-oriented model, so each of them does not satisfy only one of the constraints generated by GREVECOM. To add a deadlock, in each model we modified two dependencies and made them mutually contradictory. By modifying a single trigger in each model, we created an *illegal transition* – i.e., a transition, which is not allowed by the initial design.

Results. Firstly, as we observe in Figure 4.10, the verification time of the correct model is always lower than the verification time of other models. Secondly, the verification time increases with the number of constraints. This is the results of the specific NuSMV execution: constraint verification procedures are isolated from each other and executed sequentially. Thus, NuSMV requires additional CPU-time whenever a new constraint is added. Thirdly, we observe an increased time of the verification of the models with flaws, without a significant connection to a type of a flaw, though. As we show in the following Section 4.2.7, this happens because in NuSMV, counterexamples are generated in a dedicated procedure that is executed after the actual verification. Generally, the verification time is negligible: 50ms at most. Even combined with the generation time measured in the Section 4.2.5, the total overhead is very low.

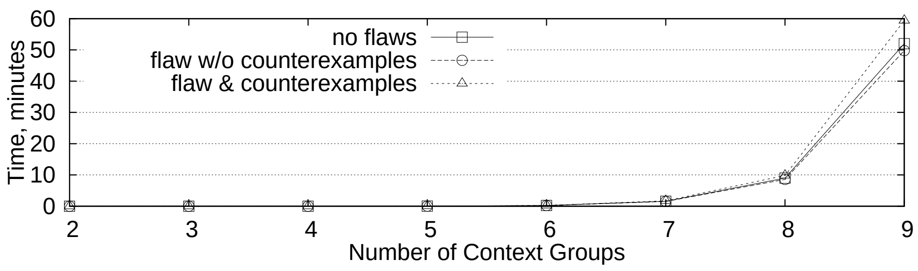


Figure 4.11: *Verification Time.* Verification of the correct model requires more time than verification of the model with a flaw and without generating counterexamples.

4.2.7 Scaling of Verification

Differently from the Section 4.2.6, we evaluate the scaling ability of our approach. It is difficult to find a real application with a context-oriented model that is large enough. That is why we measure the verification time of artificially synthesized context-oriented models. The setting of this experiment is identical to the one described in Section 4.2.6.

In our previous experiments we observed that verification time loosely depends on the type of flaws. That is why we examine two types of model: a fully correct model and a model with an arbitrarily chosen flaw – a deadlock. For both models in this experiment, we sequentially increase the number of context groups, while keeping the number of contexts per group constant – 5 contexts per group. We also keep the number of transitions, triggers, and dependencies constant in both models.

Results. Our results are displayed in Figure 4.11: *no flaws* is the verification time of the fully correct model; *flows w/o counterexamples* is the verification time of the model with a deadlock, but without generating counterexamples; and *flaws & counterexamples* is the verification time of the model with a deadlock, plus generating counterexamples.

Generally, we observe that the verification time grows exponentially with the linear increase of the number of context groups. It can be explained by the exponential explosion of states: whenever we add a new context group, the number of states is multiplied by the number of contexts in the new group. Because of that, our approach can not be scaled on large models very well, but the verification time still remains reasonable with more realistic configurations: models with $N_C = 5$ and $M_{CG} \leq 7$ are verified in less than a minute.

As we can see, the verification time of the model with flaw and without generating a counterexample is slightly lower than the time needed to verify a correct model. The reason is that NuSMV verifies only a part of the initial model, incrementally extending this part to the whole model. The verification process is complete when either the specification is false or the verified part covered the whole model. Thus, NuSMV can report false specification very early on without verifying the whole model.

NuSMV also separates the process of verification and the process of generating a counterexample. If generation of counterexample is enabled and specification is false, NuSMV scans a state-space again building a counterexample from the visited states. It results in an increased verification time, as we observe in Figure 4.11.

4.3 Summary

In this chapter we show how our concepts can be used in the designing of the adaptive WSN software. In Section 4.1 we provided a detailed description of the context-oriented models of the applications, which are described in the previous chapter. The results of our evaluation in Section 4.2 have shown that with our concepts the yielded software is more modularized and decoupled. It is easier to understand and to debug this software. Moreover, the efforts required to modify the software is reduced when our concepts are used. These benefits come, however, with a negligible price: typical WSN applications show up to 2.5% memory overhead and up to 200ms of the verification time.

Part II

Dealing with Time Requirements in Adaptive Software

CHAPTER 5

Concepts and Early Prototype

The problem of the adaptation in time-critical systems is similar to the one we observed in WSNs. The difference here is that the time aspect is crucial in such systems. Examples of such systems are automotive embedded systems, miniaturized robots, aerial drones, etc.

In this chapter we focus on aerial drones as a representative example of time-critical systems, as described in Section 5.1. Based on the latter, we outline the problem in Section 5.2, and propose our solution in Section 5.3. In our prototype that is described in Section 5.4, we show how our concepts can be implemented and used. The preliminary evaluation is provided in Section 5.5.

5.1 Background

Nowadays, the necessity of small aerial drones becomes more and more evident. Indeed, their size and agility allows them to perform tasks, which are impossible for other types of robots. For example, indoor rescue operations [79] or indoor gas leak localization [10]. The small size allows miniaturized drones to successfully execute tasks in such environments, where it is crucial to have a high agility due to complicated topology of space.

Algorithm 2: A typical controller for obstacle avoidance.

```
1  $Direction$ ← STRAIGHT_FORWARD
2  while True do
3      read hardware buttons $stop$ and $reset$
❹      if $stop$ then
5          a user-defined stop routine
6          break
7      end
❽      if $reset$ then
9          a user-defined reset routine
10     end
⓫      step()
12     wait for 10ms
13 end
14 Procedure step()
⓯      reading a $proximity$ from sensors
16     for each $reading$ in $proximity$ do
⓱          adjusting the $Direction$ according to the $reading$
18     end
⓳      modulating motors according to the $Direction$
20 end
```

However, the small size is a limitation as well: execution time is limited due to small batteries; crashes and faults are more critical; small control boards have memory and performance limitations.

Aerial drones are representative of CPSs. Their control boards modulate the motors depending on readings from a set of sensors that are embedded into the board. The drones are representative of time-critical systems as well. Indeed, to keep the flight, the control board has to continuously re-modulate motors depending on the input from the sensors. Any delay in modulation leads to the unexpected or inadequate behavior, or even to a crash. Crucially, every task in this platform has time boundaries. Should the boundaries be violated, and the system will not be able to react to the environmental changes in time.

A natural way to program an aerial drone is to create a dedicated controller, which is specific for the task at hand. The controller runs either on the base-station or on the board of the robot and contains three main steps: *i)* sensing; *ii)* analyzing; and *iv)* actuating. Every 10-100 millisecond, or even faster, the controller calls a function, where a programmer has implemented a sensing routine, a data analysis, and an action execution. A typical controller [72] is displayed in algorithm 2.

The main loop in Algorithm 2 maintains the control input, such as *stop* and *reset* buttons on lines ❹ and ❽ . The **`step()`** procedure is called from the loop every 10 ms, as on line ⓫ . Firstly, the controller reads the data from the proximity sensors on line ⓯ . Based on these readings, the controller adjusts the movement *Direction* to avoid possible collisions, as on line ⓱ . Finally, the calculated *Direction* is used to modulate the motors on line ⓳ . This controller is fairly simple, because the only environmental dimension we referring to is the existence (or absence) of obstacles. More realistic scenarios would require to consider more dimensions, which will eventually lead to the problem that is described in the following section.

5.2 Problem

To illustrate the problem, let us refer to the gas leak localization task described in Chapter 2. The latter can be split into two sub-tasks: *i)* hover and sense; and *ii)* localize the leak. A natural way to implement this kind of functionality is to use flight modes that are encapsulated in a dedicated software controller: a single controller for a single sub-task. The solutions for these sub-tasks already exist [19, 23, 62], but combining them into a single controller is a challenge.

Each controller operates in its own software environment – a set of variables and asynchronous operations – and they are not always compatible. The problem arises when a programmer needs to switch between the controllers: whenever a controller switching is required, the system has to re-initialize the variables and periodic tasks within the time boundaries between two sequential calls of the step-function. Should it not be the case, a robot could not adapt to the new situation.

For example, the drone hovers over some location, but should its neighbor detect an area with a high gas concentration, the drone should move towards this area and broadcast its readings. This requires an initialization of a periodic task that broadcasts the sensed values. Moreover, the initialization has to be executed within the deadline and without a proper adaptation technique, the implementation of this kind of software becomes complicated.

Orthogonal to the main task, each drone should adapt to environmental changes, such as: *i)* battery level – every time battery is low, the drone returns to a charging site; and *ii)* the swarm may be partially within the range of Base Station, thus each drone might need to switch between different communication protocols depending on availability of the connection with the Base Station. These adaptations should occur independently of each other, which makes the controller even more complicated.

Current work towards adaptivity in resource-constrained and time-critical systems lacks the proper language support to implement this kind of software. Even though, there are plenty of algorithms [61] and patterns [47] for adaptive software under resource- and time-constraints, without a proper language support the implementation of the adaptation mechanism is totally on programmer's shoulders. Thus, the adaptive software may become entangled and hard to debug, to understand, and to maintain.

For example, our investigations of the real code-base of ArduPilot [4] revealed that the adaptation mechanism is implemented by programmers manually. Programmers had to manually deal with the adaptation under time constraints, which makes the software hard to maintain and to understand in general, and even more, when programmers tackle aspects that are orthogonal to the main functionality. For example, in conditions of low battery drone has to suspend a task execution and return to the base to recharge. Gas concentration impacts on the sampling mode: if the gas concentration is high, drone tries to get closer to the location of the concentration peak, instead of centrally-defined sampling position. Finally, Base station availability dictates a relay protocol. Without a proper adaption technique, these aspects make the source code look like "spaghetti code".

The step function displayed in Figure 5.1 is a simplified version of the ArduPilot's step function. The control functionality is implemented in the function **`step()`**, which is called every 10ms, as shown on line ❸. This function calls different controller-based step-functions depending on the global variable **`current_controller`**: either **`hover_step()`** or **`leak_loc_step()`**, as on lines ❼ and ❿. The controller is changed by the function **`set_controller()`**, which is defined on line ㉔ and called with different parameters depending on the readings. This function initializes a single controller, as on lines ㉙ and ㉜, executes a cleanup routine of the previous controller on line ㊶, and on line ㊷ updates a global variable that holds a current controller.

There are three major problems: *i)* the possible conflicts during the controller switching are not handled; *ii)* the absence of the dedicated language abstractions makes it impossible to guarantee that the step function execution will satisfy the time boundaries; and *iii)* the absence of activation policy may freeze the whole system. We illustrate these problem below, and address them in the following section.

To illustrate the problems, let us refer to our design concepts that are that are introduced in Chapter 3. Figure 5.2 displays a context-oriented model of the adaptive controller. According to the gas leak localization application described above, a single drone is *Hovering* and sensing the gas concentration. As soon as the gas concentration is higher then the predefined threshold, the drone activates the *Leak Loc.* context, as displayed in Figure 5.2. In this context, the drone broadcasts the sensed values, and, hence, receives the values from the neighbors. To localize the leak, the drone decreases the distance to the neighbor with the highest value of the

```
 1 // should be called at 100hz or more
 2 // calls the appropriate controller
 3 static void step() {
 4   sense();
 5   switch (current_controller) {
 6     case HOVER:
 7       hover_step();
 8       break;
 9     case LEAK_LOC:
10       leak_loc_step();
11       break;
12   }
13 }
14 // reads data from sensors and updates global variables
15 void sense() {
16   int16_t concentration = read_gas_concentration();
17   if (concentration > THRESHOLD) {
18     set_controller(LEAK_LOC);
19   } else {
20     set_controller(HOVER);
21   }
22 }
23 static int8_t current_controller = NONE;
24 static bool set_controller(uint8_t controller) {
25   // boolean to record if controller could be set
26   bool success = false;
27   switch(controller) {
28     case HOVER:
29       success = hover_init();
30       break;
31     case LEAK_LOC:
32       success = leak_loc_init();
33       break;
34     default:
35       success = false;
36       break;
37   }
38   // update controller
39   if (success) {
40     // perform any cleanup required by previous controller
41     exit_mode(current_controller, controller);
42     current_controller = controller;
43   } else {
44     // Log error that we failed to enter desired controller
45     Log_Write_Error(controller);
46   }
47   return success;
48 }
```

Figure 5.1: *Adaptive controller.*

gas concentration. In case of emergency – e.g., unexpected obstacle – the drone activates the *Landing* context. Using this model we illustrate the major problems.

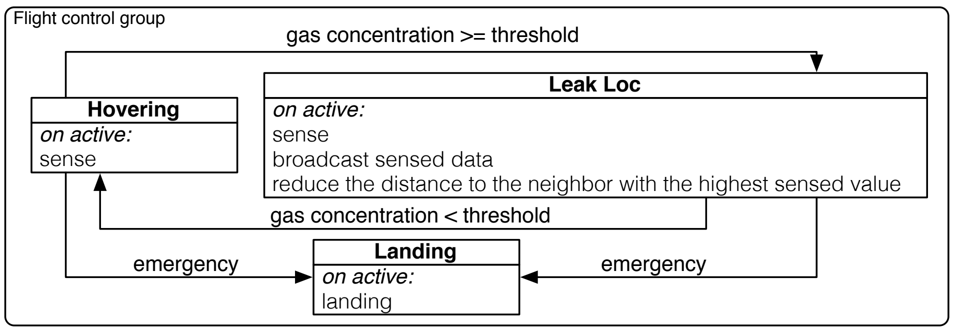


Figure 5.2: *A Flight Control context group model of the gas leak localization application.*

Activation conflicts. In this model, after the initialization routine and before the cleanup routine, the two contexts may interfere with the work of each other. Indeed, the *Leak Loc.* context contains a periodic task *broadcast sensed data*, as displayed in Figure 5.2. Since the cleanup routine of the *Leak Loc.* context is executed only after the *Hovering* context is initialized, as on lines ㉜ and ㊶ in Figure 5.1, this task is still active even after the initialization is complete. The *Hovering* context, however, is not supposed to broadcast the data.

Periodic tasks are not the only problem: any asynchronous operations – e.g., interrupts, callbacks, or asynchronous tasks – will lead to the interference between the contexts' functionality. This interference should be carried out by the programmer by providing a safety mechanism manually.

Undefined activation time. In ArduPilot it is required to call the step function every 10ms. Assume that the controller-based step function require 5ms to execute. It means that the activation time can not exceed 5ms. In ArduPilot, however, a programmer can not guarantee that these time boundaries are met. Moreover, the programmer can not leave the drone without any control for more than 10ms. Should the activation process not finish within the 5 ms, the drone will be out of the control.

No activation policy. Assume the value of the gas concentration in our example is close to the threshold. A stochastic nature of the sampling data will lead to the rapidly growing number of sequential commands to activate the *Leak Loc.* and the *Hovering* contexts. Without a proper activation policy, the system may stuck trying to execute all of them. The problem becomes even bigger without the proper deadline mechanism, as the control over the drone will be lost while the system is stuck.

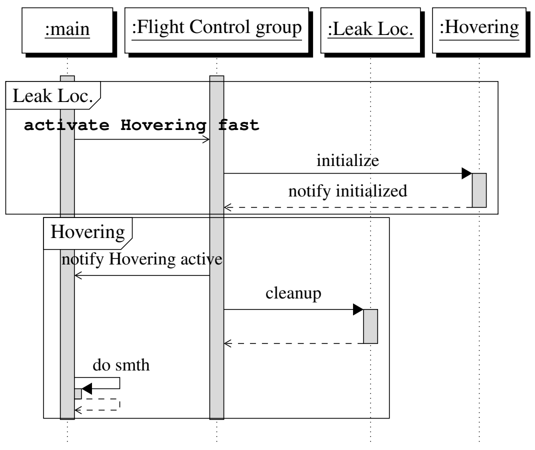


Figure 5.3: *A fast activation of the context Hovering.*

5.3 Solution

In this section we address these problems by providing simple concepts. We tackle the problem of conflicts by introducing *activation types* in Section 5.3.1. To allow a programmer to specify activation time boundaries, in Section 5.3.2, we introduce the notion of *deadline*. Should the deadline be violated, the programmer will shortly be warned. Finally, our *activation queue* described in Section 5.3.3 provides an activation policy for multiple activation commands.

5.3.1 Activation Type

In order to provide a control over the activation time, we introduce two types of the activation: *i) lazy* and *ii) fast*. The former allows a programmer to activate a context and to avoid any conflicts without much efforts. The latter type takes less time, but the programmer should take care of the possible conflicts during the activation. Below, we discuss the both types and the conflicts that might occur during the activation.

Fast activation. This type of activation provides the fastest possible activation of the context. The sequence diagram of this activation type is displayed on Figure 5.3. Let us consider the drone is in the *Leak Loc.* context. To activate *Hovering* as fast as possible, a programmer uses the command **`activate Hovering fast`**. In this case, the context *Hovering* is initialized, while the context *Leak Loc.* is still active. As soon as the context *Hovering* is initialized, the main process receives the notification, and only after that the context *Leak Loc.* is cleaned up.

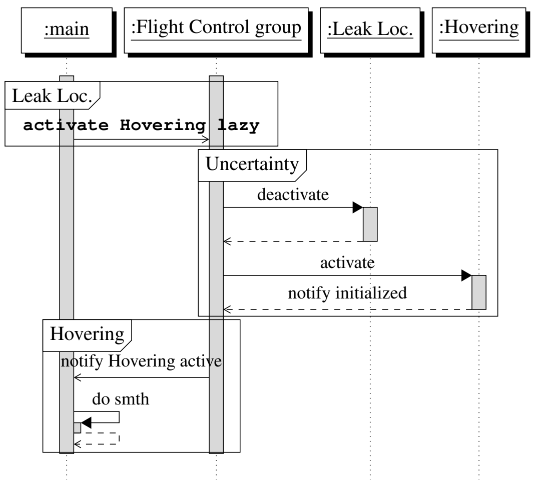


Figure 5.4: *A lazy activation of the context Hovering.*

In this activation scenario, the context *Leack Loc.* is active during the initialization of the *Hovering* context. Then, the latter is active during the *Leack Loc.* cleanup routine. It leads to the possible interference during the activation procedure, as we discussed previously. This interference should be carried out by the programmer either by providing the safety mechanism manually, or by using our notion of *Wrappers*. The latter are the software components that encapsulate any asynchronous operations. Any *wrapper* instance is dedicated to a single context, and whenever the asynchronous task of the context is about to be executed, the *wrapper* intercepts the command and executes it only if the context is active. These additional efforts are required form the programmer as a price for the fast context activation.

Lazy activation. Another type of the context activation does not require any additional efforts from the programmer, since it is intended to perform a safe activation. The diagram 5.4 depicts the process of a *lazy activation* of the context *Hovering*. To avoid the interference between two contexts, system deactivates the context *Leak Loc.*, and then activates the context *Hovering*. During this procedure, the system is in *Uncertainty* state, because neither of contexts is active. This activation type avoids the interference between the contexts by default and the programmer may not care about them. These benefits come with a price of the activation time: it is a sum of the time needed for the clean up of the previous context and the time needed for the initialization of the next context.

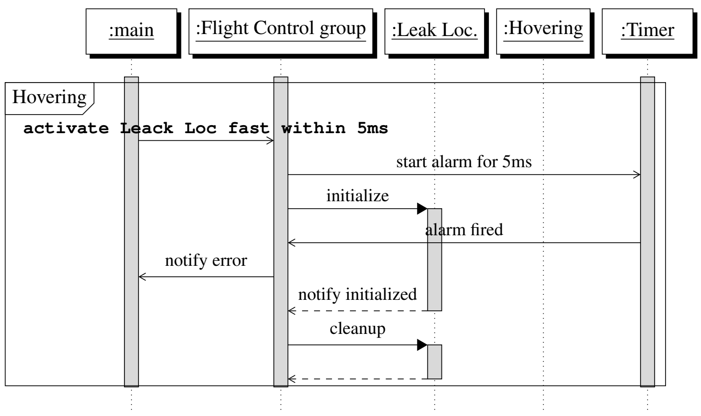


Figure 5.5: *A fast activation with the deadline.*

5.3.2 Deadline

To allow a programmer to strictly control the activation time, we introduce a modifier **`within`**. Being used after the activation command, this modifier determines the time boundaries for the activation that has no deadlines otherwise. For example, in our scenario, the drone can not be out of the control for more that 10ms, and the activation routine should finish within 5ms, as we discussed before. To this end, the programmer can use the command **`activate Leak Loc within 5ms`**. Should the activation process not finish within the 5 ms, the activation will stop, and the previous context will be activated again.

The diagram in Figure 5.5 illustrates the execution flow of the command **`activate Leak Loc fast within 5ms`**. The dedicated *Timer* is started by the *Flight Control group*. If the initialization of the context *Leak Loc.* takes more than 5ms, the alarm is fired, the activation is canceled, and the context is cleaned up.

The similar procedure also applied to a deadline violation in the *lazy activation*. The diagram in Figure 5.6 illustrates the deadline violation at the clean-up of the context *Leak Loc*. In this case, the context *Hovering* is reinitialized. If the alarm is fired during the initialization of the context *Leak Loc.*, system has to cleanup the context, and to re-initialize the context *Hovering*, as displayed in diagram 5.7.

5.3.3 Activation Queue

In order to handle multiple activation commands, we introduce a notion of *activation queue*. Whenever several activation commands appear in the

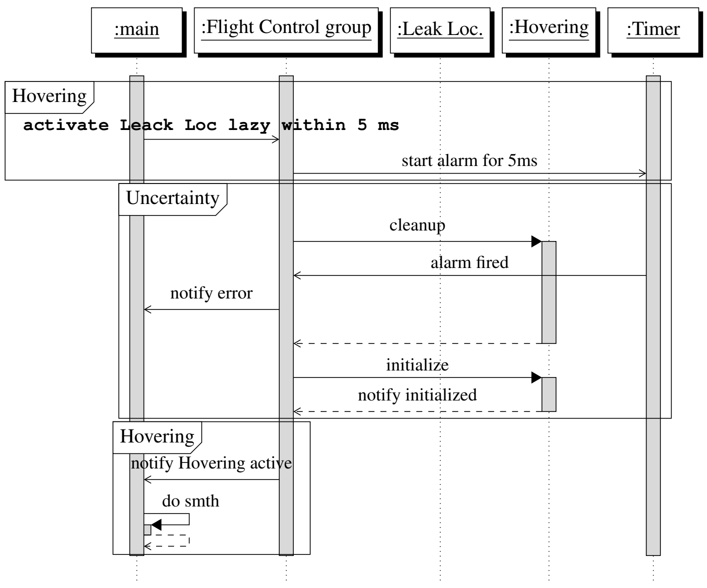


Figure 5.6: *A lazy activation with the violated deadline during a cleanup.*

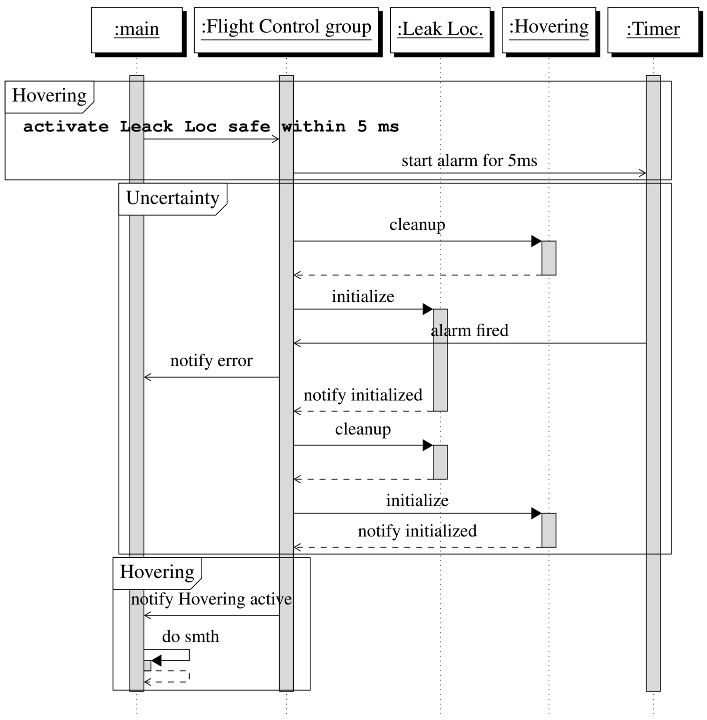


Figure 5.7: *A lazy activation with the violated deadline during an initialization.*

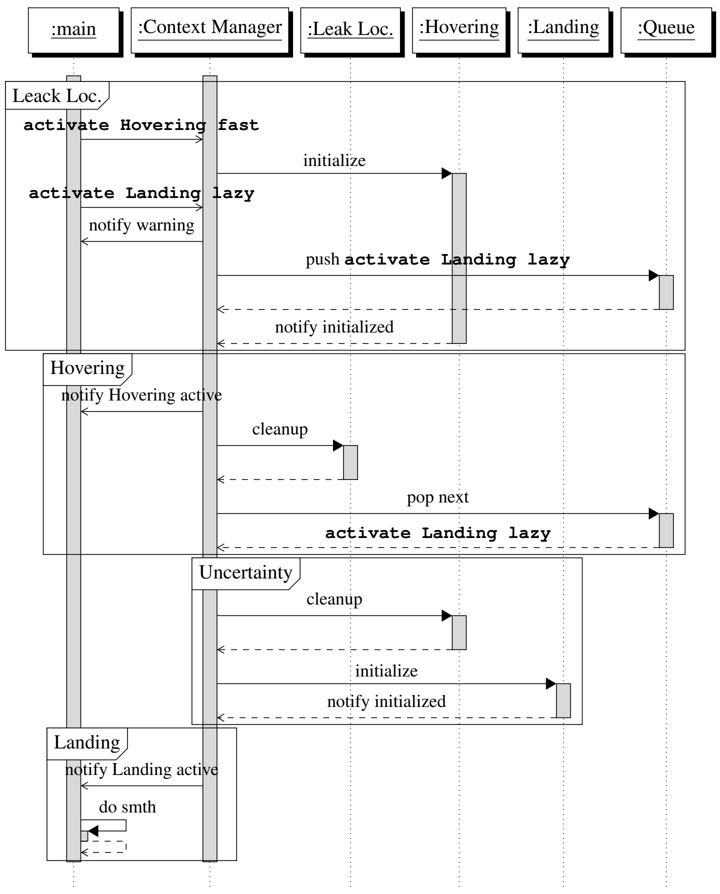


Figure 5.8: *A multiple context activation.*

execution flow, they are stored in the *Queue*, as displayed in diagram in Figure 5.8. In the latter, the activation of the context *Landing* is following just after the activation of the context *Hovering*. Since the activation of the latter is not complete, the activation command of the context *Landing* is moved to the *Queue*, and the *warning* notification is fired, indicating that the activation is suspended. As soon as the activation of the context *Hovering* is complete, the next command – **`activate Landing lazy`** – is popped and executed.

The appearance of the same activation command more then once in the queue will lead to the repeatable activation. For example, if the activation sequence $Hovering \rightarrow LeakLocalization \rightarrow Hovering$ will be executed, it will not change the active context, but will consume the precious time. That is why, being stored in the queue, the whole sequence will be

```
template<typename FunctionalityClass>
class Context : public FunctionalityClass{
 public:
  virtual void initialize() = 0;
  virtual void cleanup() = 0;
  //...
};
```

Figure 5.9: *Context template.*

ignored except the last command.

The additional modifier **`immediately`** allows a programmer to start the activation procedure immediately and to clear the queue. For example, in case of the emergency – e.g., an unexpected obstacle – the drone must land no matter what, and all the queued activation commands are not relevant anymore. As a result of the command **`activate Landing fast immediately`** the system activates the context *Landing* immediately.

5.4 Prototype

To show our concepts in action, we implemented a prototype. Since the majority of the aerial drones – our target platforms – are based on the Cortex-M3 micro-controller, we used a Nucleo STM32L152 prototyping board. The latter provides all the features of the Cortex-M3 micro-controller via the dedicated C++ libraries. This platform is not yet supported by our Translator, and there is no integration with GREVECOM, hence, for this board we manually implemented our design concepts described in Chapter 3.

In our prototype, every user-defined context inherits the *Context* class, that is is depicted in figure 5.9. This class is a template, an argument of which is an abstract class with the layered functions. Every user-defined context provides its own implementation of the layered functions.

Similarly, every user-defined context group inherits the *Context Group* template shown in Figure 5.10, and provides a layered function, variations of which are implemented in the contexts. The *Context Group* template provides different activation policies, as we discussed previously: **`FAST`** and **`LAZY`**, as on line ❶. The **`activate()`** function provided by the template accepts two arguments: a context and a type of the activation, as it is shown on line ❺. An overloaded function on line ❻ accepts an additional *deadline* parameter. If the activation is successful, the template *calls* a *callback* that can be set on line ❾. Otherwise, the template calls the

```
❶ enum activation_t {NONE, SAFE, LAZY};
2 template<typename FunctionalityClass>
3 class ContextGroup : public FunctionalityClass {
4  public:
❺   void activate(Context<FunctionalityClass>* ctx, activation_t type);
❻   void activate(Context<FunctionalityClass>* ctx, activation_t type,
       float deadline);
7   void activate_immediately(Context<FunctionalityClass>* ctx,
        activation_t type);
8   void activate_immediately(Context<FunctionalityClass>* ctx,
        activation_t type, float deadline);
❾   void set_callback(void (*callback)(Context<FunctionalityClass>*));
❿   void set_errback(void (*errback)(Context<FunctionalityClass>*));
11   //...
12 };
```

Figure 5.10: *Context group template.*

```
class FooGroupFunctionality {
 public:
  virtual void layeredFunction() = 0;
  //...
};
```

Figure 5.11: *Layered function declaration.*

```
class FooContextA : public Context<FooGroupFunctionality> {
 public:
  void layeredFunction(){//...}
  void initialize(){//...}
  void cleanup(){//...}
  //...
};
```

Figure 5.12: *Context component.*

```
class FooGroup : public ContextGroup<FooGroupFunctionality> {
 public:
  void layeredFunction(){//...}
  //...
};
```

Figure 5.13: *Context group component.*

`errback` that is set on line ❿.

The usage of these templates is very easy. Firstly, a programmer has to define the layered functions as an abstract class, as shown in Figure 5.11. This class serves as a base for both a context and a context group compo-

```
❶ void contextChanged(Context<FooGroupFunctionality>* ctx) {//...}
❷ void deadlineError(Context<FooGroupFunctionality>* ctx) {//...}
3 int main() {
4   FooGroup group;
❺   group.set_callback(&contextChanged);
❻   group.set_errback(&deadlineError);
7   FooContextA ctxA;
8   FooContextB ctxB;
9   while(1){
❿     group.activate(&ctxA, SAFE);
⓫     group.layeredFunction();
⓬     group.activate_immideately(&ctxB, FAST, 0.5);
13   }
14 }
```

Figure 5.14: *Usage.*

```
1 void FooContextA::initialize(){
❷   _ticker = new Ticker();
❸   _ticker->attach(&task, 0.2);
4   //...
5 }
```

Figure 5.15: *Context FooContextA initialization.*

nents. For example, Figure 5.12 depicts a context component, which implements its own **`initialize()`**, **`cleanup()`**, and **`layeredFunction()`**. The context group component should inherit the context group template, as in Figure 5.13.

The example usage of this implementation of our concepts is shown in Figure 5.14. To call the layered function, a programmer uses the context group component, as it is shown on line ⓫. Whenever the programmer wants to activate the context, he or she uses a function **`activate()`** as on line ❿. If the activation was successful, a programmer-defined callback is called. The latter is defined on line ❶ and set on line ❺. Similarly, the **`deadlineError()`**, which is defined on line ❷ and set on line ❻, will be called if the activation is failed.

The context group component also handles the *activation queue*. As we previously described, the queue activates the context in FIFO order. The queue is hidden form the programmer, however, in order to activate an emergency context, a programmer can use the function **`activate_immediately()`**, as on line ⓬. The latter cleans the queue and activates the defined context as soon as possible.

During the *fast* context activation, the functionality of two context can interfere with each other. To help a developer to take care of this, we also

```
_ticker = new SafeTicker(this);
```

Figure 5.16: *SafeTicker example.*

implemented a set of *wrappers*. Consider the context **FooContextA** has a periodic task that is launched during the initialization, as on line ❸ in Figure 5.15. However, as we argued in the previous section, if the context **FooContextB** is activated *fast*, the periodic task will be executed even when this context is already active. A programmer can use our wrappers by simply replacing the line ❷ with the line that is displayed in Figure 5.16, no further refactoring is needed. Similarly, the programmer can use the **SafeInterruptIn** and **SafeTimeout** classes that wrap the standard **InterruptIn** and **Timeout** classes provided by the board manufacturer. Our *Safe* classes have to be instantiated with the pointer to the context object in order to track the active context and to prevent any asynchronous operations – e.g., tasks, interrupts, timers – when the context is not active. In the following section we use this prototype to perform a set of experiments.

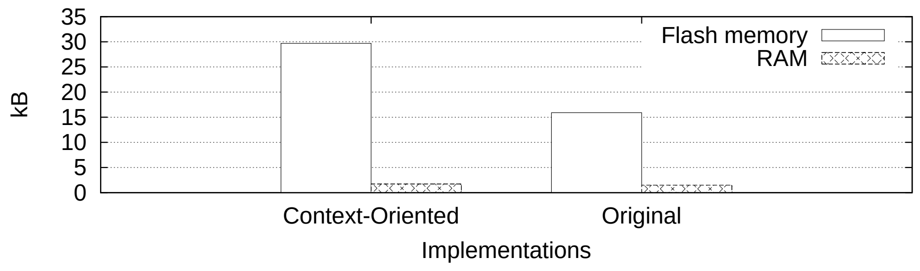
5.5 Preliminary Evaluation

Based on our prototype described in previous section, we measure the performance and memory overhead that comes with the benefits of our solution. In our experiments we use the on-line IDE [50] that is provided by the manufacturer of our prototyping board. All the experiments were developed under the on-line IDE and then run on the Nucleo board described in the previous section.

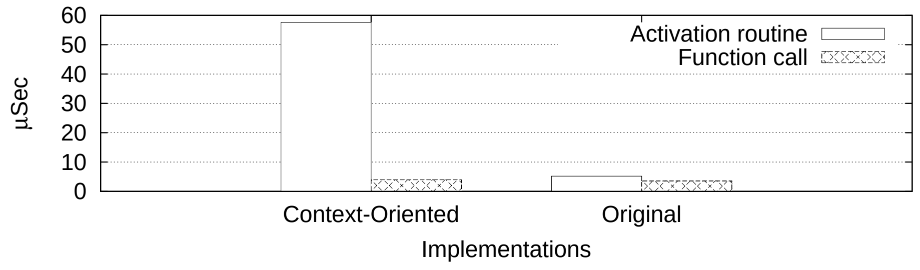
Firstly, in Section 5.5.1 we measure Flash memory and RAM overhead that come with our approach. Then, we measure the MCU time required by the additional processing within our wrappers in Section 5.5.2. Finally, in Section 5.5.3 we analyze the trade-offs developers might expect from our concepts.

5.5.1 Activation Time and Memory Overhead

Our concepts take care of a significant part of processing in the time-critical context-oriented software. These advantages come with the cost of memory and MCU performance overhead, which we asses in this section. In doing so, we implemented two functionally equivalent adaptive controllers for the Nucleo board: *i)* the original controller that is described in Section 5.2, and



(a) *Memory overhead.*



(b) *MCU usage.*

Figure 5.17: *Memory overhead and MCU usage.*

ii) a controller that uses our concepts.

We estimate the memory overhead by using the compilation information exposed by the on-line IDE. To measure the MCU time overhead we launch the context activation routine of the context-oriented controller for 1000 times and measure the routine execution time with the standard **`Timer`** component. For the original controller, we measured the time required for the execution of the *switch* statement. Similarly, we assess the time required for the controller-specific step-function.

Results. Figure 5.17a depicts the comparison of the flash memory and RAM usage between the different approaches. The RAM usage in both controllers is fairly negligible and the overhead is only 13%. The flash memory usage of the *Context-Oriented* controller is up to 87% higher. This is due to the fairly simple implementation prototype. Indeed, in our prototype there is only one context group with two contexts, which make the size of the application logic very small: only 15kB. More realistic applications, however, are much bigger – e. g. ArduPilot [4] requires more than 262kB of the flash memory – and the relative size of the adaptation mechanism is much smaller in these cases.

The MCU overhead is displayed in Figure 5.17b. The controller-based

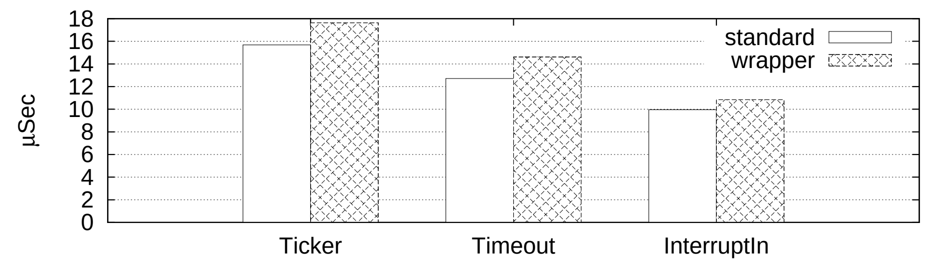


Figure 5.18: *The overhead of the wrappers.*

step-function call overhead is negligible: only 7%. The significant value of the activation routine in *Context-Oriented* controller is due to the additional processing that is covered by our concepts. This processing adds a functionality and guarantees that do not exist in the original implementation. For example, unlike the original implementation, in our prototype developers can choose between different activation types. The activation queue handles the multiple activation commands transparently to the developers. Moreover, through the deadline handling we guarantee that the activation will be finished or interrupted within the specific amount of time.

5.5.2 Wrapper Classes Overhead

The *wrappers* are designed to avoid the interference between the contexts transparently to the developer. For example, when using the *fast* activation type, the developer should either manually avoid possible conflicts during the activation, or use our wrappers. The latter encapsulate the classes that include any asynchronous operations. In this section we measure the additional MCU time that is required to invoke the asynchronous operation by using our wrappers – such as **`SafeTicker`**, **`SafeInterruptIn`**, and **`SafeTimeout`** – as compared to their counterparts, such as **`Ticker`**, **`InterruptIn`**, and **`Timeout`** that are provided by the board manufacturer. In doing so, for the **`Ticker`** and **`Timeout`** classes and their wrappers we launch the asynchronous task so that it is executed immediately, then we measure the time between the launch and the actual task execution. For the **`InterruptIn`** and its **`Safe`**-implementation, we programmatically invoke the interrupt and measure the time between the interrupt and the task execution.

Results. Our measurements are displayed in Figure 5.18. The overhead is due to the additional processing that is performed by the wrapper. In

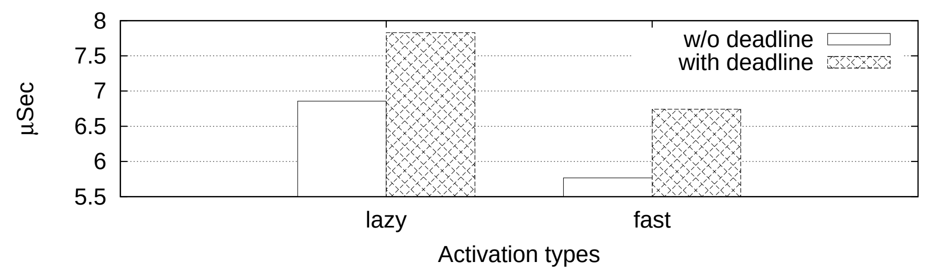


Figure 5.19: *Activation types and options overhead.*

this processing the software checks if the context associated with the asynchronous task is active and decides whether to execute the task or not. As expected, the additional processing require more MCU time. Our example, however, is artificial and greatly simplified, but it shows that with a cost of a small performance overhead our wrappers allow the developers to transparently avoid the conflicts during the context activation.

5.5.3 Activation Types and Options

As we have shown above, our concepts bring a performance overhead. In this section, we show the trade-offs a developer might expect when using our concepts with different options. The developer can choose between two orthogonal activation parameters: the activation type, and the optional deadline. The activation type is a mandatory parameter and can be either *fast* or *lazy*. The deadline, however, is optional. The processing of the deadline parameter is designed so that the value of the deadline does not affect the performance overhead, and we measure only two cases: *i)* activation with a deadline, and *ii)* activation without a deadline. Hence, there are four combinations of options: *fast with deadline*, *fast without deadline*, *lazy with deadline*, and *lazy without deadline*. A single combination of the parameters delivers specific trade-offs the developer would expect from the design of the different activation types. For each combination we asses the time required for the activation as described above in Section 5.5.1.

Result. As shown in Figure 5.19, the *lazy* activation type requires more MCU time to activate a context, as expected. This is due to the additional processing that prevents the possible conflicts during the activation. It is justified in the applications where developers heavily rely on the asynchronous operations and do not want to trade the performance of the standard classes with the safety delivered by our wrappers. On the other hand, one can get the faster activation by using the *fast* activation type, which is suitable for the applications where developers need the faster adaptivity. However, as the cost of the latter the developers must use our wrappers whenever any asynchronous operations are needed. In any case, the deadline option leads to the additional MCU performance overhead.

5.6 Summary

In this section we focused on the timing aspect of the adaptation decisions in time-sensitive CPSs. In Section 5.1, we aimed at understanding how the typical time-sensitive CPSs are programmed. This served to outline the problem and to describe it in Section 5.2. We addressed this problem in the Section 5.3 by providing our language independent design concepts. Then, in Section 5.4 we have shown how the developers can use our concepts in a real language, and what benefits they might expect using this concepts. Finally, Section 5.5 contains the performance evaluation of our concepts.

CHAPTER 6

Conclusion and Future Work

Cyberphysical systems play a great role in the interactions between the physical world and its virtual representation. They do not only gather data from the environment, but also take actions on the physical world. This way, CPSs provide the functionality that bridges the physical and virtual worlds. The key property of such functionality is that it is emerging from the tight interaction between the CPS device and the real world. The latter, however, exhibits multiple dimensions that are changing continuously and independently, but affect the functionality simultaneously. This requires the CPSs software to be *adaptive*.

A large part of CPSs is also time-sensitive. In such systems, the control logic is often encapsulated in a dedicated control-loop that is run at a certain frequency. Crucially, the control logic must examine the sensors input and generate the control decisions in real-time. Should any adaptation process be needed, the execution of the adaptation must be finished within certain *time bounds*.

Nonetheless, the above aspects pose difficult challenges to developers, as they are still left without a dedicated programming support to define the adaptive functionality with respect to the complected environmental dynamics. The solutions presented in this work aimed at simplifying the programmer's life in developing adaptive time-sensitive CPSs software, while also improving the system's dependability. Unlike the previous work, we focused on resource constrained systems, and delivered programming abstractions that allow the developers to design, to implement, and to verify the adaptive CPSs software against environmental evolutions.

To reach our ultimate goal, we firstly identified the challenges that developers must address while developing adaptive time-sensitive CPSs software for resource-constrained systems. Then, we examined the state of the art in the areas of adaptive and time-critical software for both resource-constrained and mainstream systems. We compared different approaches and identified the missing programming support, which is often resulted in a "spaghetti code". This served to establish the conceptual framework, which is then implemented in our contribution.

In the first part of the thesis, we solely addressed the *adaptivity* problem in resource-constrained CPSs software. In doing so, we adapted Context-Oriented Programming paradigm to Wireless Sensor Networks—a paradigmatic example of the resource-constrained CPSs—by providing a handful of design concepts. The latter were implemented in CONESC—our own extension to the real language for WSN called nesC. Moreover, we provided a modeling semantics built on top of our concepts, which allow to model the adaptive software even before the actual deployment. Our own tool called GREVECOM supports this semantics and allow to verify the model against the environmental dynamics, and to generate CONESC templates based on the model. Thus, we covered a significant part of the developing of the adaptive WSN software.

Our early experience in using our concepts revealed certain patterns that naturally exhibited whenever these concepts are applied. These pattern may significantly improve the developing process, as they already implement a certain interaction between the software components. The latter are also less coupled, which makes them less dependent on each other, and the software may be modified more easily. The complexity of the components also becomes lesser when our concepts are used making the software easier to debug. These also simplify the software evolutions as it requires the software to be more understandable and less complicated. All these benefits come with a little performance overhead. Thus, the off-line verification requires less then one minute and the system requires up to 3% of additional memory and up to 30 MCU cycles whenever our programming abstractions are involved into the execution. The MCU overhead is negligible, though, as even the simplest operation—turning LED on/off—requires 8 MCU cycles.

In the second part, we focused on the timing aspect in enforced adaptation decisions in CPSs. In doing so we provide language independent design concepts, that rely on well-specified semantics when triggering adaptations. Our *deadline* concept aims at defining the time boundaries in adaptation decisions. Thus, whenever the deadline is violated, the error event is fired up allowing the developer to apply proper countermeasures. We also introduced two different adaptation mechanisms that deliver adaptation latency vs. programming efforts trade-offs, and allow the developers to choose the most appropriate adaptation type. The benefits of our approach is shown in a simple prototype built for Cortex-M3 platforms. Our evaluation has revealed that the benefits came with a modest performance overhead.

As a logical continuation of our work we work on the real-world applications of our concepts. In doing so, we aim at providing a tool-chain built atop of these concepts for the developers of the sensing vehicles software. The latter represent typical time-sensitive CPSs, as on these platforms the control logic is often changed at run-time due to the environmental dynamics. We are also intended to extend our formalization semantics to allow the developers to model the timing aspect of the adaptation. With this semantics we upgrade the verification algorithm to be able verify the model against the timing limitations. With this regard, there are several open challenges. How to formalize the adaptation process under time limitations? How a designer is supposed to specify the fact that the adaptation is executed is within the time boundaries? These questions are currently awaiting an answer we aimed to provide.

Bibliography

[1] L. Abeni and L. Palopoli. Adaptive real-time scheduling for legacy applications. In *Emerging Technologies and Factory Automation, 2008. ETFA 2008. IEEE International Conference on*, pages 583–590, Sept 2008.

[2] P. Alfke and R. Padovani. Radiation tolerance of high-density fpgas, 1998.

[3] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, pages 6:1–6:6, New York, NY, USA, 2009. ACM.

[4] ArduPilot. www.ardupilot.com.

[5] Neil C Audsley. *Deadline monotonic scheduling*. Citeseer, 1990.

[6] Pawel Bachara, Konrad Blachnicki, and Krzysztof Zielinski. Framework for application management with dynamic aspects J-EARS case study. *Information & Software Technology*, 52(1):67–78, 2010.

[7] Gordon S. Blair, Geoff Coulson, Michael Clarke, and Nikos Parlavantzas. Performance and integrity in the openorb reflective middleware. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 268–269, London, UK, UK, 2001. Springer-Verlag.

[8] J. Boner. Aspectwerkz - dynamic aop for java, 2004.

[9] Themistoklis Bourdenas and Morris Sloman. Starfish: Policy driven self-management in wireless sensor networks. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '10, pages 75–83, New York, NY, USA, 2010. ACM.

[10] Timo Rolf Bretschneider and Karan Shetti. Uav-based gas pipeline leak detection. In *Proceedings of the Asian Conference on Remote Sensing*, 2015.

[11] Michael Breza, Richard Anthony, and Julie A. McCann. Scalable and efficient sensor network self-configuration in bioans. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, Boston, MA, USA, July 9-11, 2007*, pages 351–354, 2007.

[12] L. Capra, W. Emmerich, and C. Mascolo. Carisma: context-aware reflective middleware system for mobile applications. *Software Engineering, IEEE Transactions on*, 29(10):929–945, Oct 2003.

[13] Alberto Cerpa and Deborah Estrin. Ascent: Adaptive self-configuring sensor networks topologies. *IEEE Transactions on Mobile Computing*, 3(3):272–285, July 2004.

[14] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In Nicolas Halbwachs and LenoreD. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Berlin Heidelberg, 2005.

[15] R. F. Conde, A. G. Darrin, F. C. Dumont, P. Luers, S. Jurczy, N. Bergmann, and A. Dawood. Adaptive instrument module - a reconfigurable processor for spacecraft applications, 1999.

[16] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G.P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *Pervasive Computing and Communications, 2007. PerCom '07. Fifth Annual IEEE International Conference on*, pages 69–78, March 2007.

[17] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1:1–1:42, March 2008.

[18] Jim Dowling, Tilman Schafer, Vinny Cahill, Peter Haraszti, and Barry Redmond. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In Walter Cazzola, RobertJ. Stroud, and Francesco Tisato, editors, *Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg, 2000.

[19] Frederick Ducatelle, GianniA. Di Caro, Alexander FÃ¶rster, Michael Bonani, Marco Dorigo, StÃ©phane Magnenat, Francesco Mondada, Rehan O'Grady, Carlo Pinciroli, Philippe RÃ©tornaz, Vito Trianni, and LucaM. Gambardella. Cooperative navigation in robotic swarms. *Swarm Intelligence*, 8(1):1–33, 2014.

[20] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, Nov 2004.

[21] Michael Engel and Bernd Freisleben. Toskana: A toolkit for operating system kernel aspects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on Aspect-Oriented Software Development II*, volume 4242 of *Lecture Notes in Computer Science*, pages 182–226. Springer Berlin Heidelberg, 2006.

[22] Joakim Eriksson, Fredrik Österlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. Cooja/mspsim: Interoperability testing for wireless sensor networks. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 27:1–27:7, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[23] Eliseo Ferrante, AliEmre Turgut, Alessandro Stranieri, Carlo Pinciroli, Mauro Birattari, and Marco Dorigo. A self-adaptive communication strategy for flocking in stationary and non-stationary environments. *Natural Computing*, 13(2):225–245, 2014.

[24] Niclas Finne, Joakim Eriksson, Nicolas Tsiftes, Adam Dunkels, and Thiemo Voigt. Improving sensornet performance by separating system configuration from system logic. In JorgeSÃ¡ Silva, Bhaskar Krishnamachari, and Fernando Boavida, editors, *Wireless Sensor Networks*,

volume 5970 of *Lecture Notes in Computer Science*, pages 194–209. Springer Berlin Heidelberg, 2010.

[25] Scott D. Fleming, Betty H. C. Cheng, R. E. Kurt Stirewalt, and Philip K. McKinley. An approach to implementing dynamic adaptation in c++. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[26] Franck Fleurey, Brice Morin, and Arnor Solberg. A model-driven approach to develop adaptive firmwares. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, pages 168–177, New York, NY, USA, 2011. ACM.

[27] Hossein Fotouhi, Marco Zuniga, M$#225;rio Alves, Anis Koubaa, and Pedro Marrón. Smart-hop: A reliable handoff mechanism for mobile wireless sensor networks. In *Proceedings of the 9th European Conference on Wireless Sensor Networks*, EWSN'12, pages 131–146, Berlin, Heidelberg, 2012. Springer-Verlag.

[28] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[29] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *SIGPLAN Not.*, 38(5):1–11, May 2003.

[30] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 1–14, New York, NY, USA, 2009. ACM.

[31] Sebastián González, Kim Mens, and Patrick Heymans. Highly dynamic behaviour adaptability through prototypes with subjective multimethods. In *Proceedings of the 2007 Symposium on Dynamic Languages*, DLS '07, pages 77–88, New York, NY, USA, 2007. ACM.

[32] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Value-based scheduling in real-time database systems. *The VLDB Journal*, 2(2):117–152, apr 1993.

[33] S. Hauck. The roles of fpgas in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615–638, Apr 1998.

[34] Wei-Je Huang and E.J. McCluskey. Column-based precompiled configuration techniques for fpga. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 137–146, March 2001.

[35] C. Huebscher and A. McCann. An adaptive middleware framework for context-aware applications. *Personal Ubiquitous Comput.*, 10(1):12–20, December 2005.

[36] M. Mezini I. Aracic, V. Gasiunas and K.Ostermann. Overview of caesarj. 3880:135 – 173, Feb 2006.

[37] Michael Jackson. The world and the machine. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 283–292, New York, NY, USA, 1995. ACM.

[38] JavaCC - The Java Compiler Compiler. `javacc.java.net`.

[39] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Eventcj: A context-oriented programming language with declarative event-based context transition. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 253–264, New York, NY, USA, 2011. ACM.

[40] Roger Keays and Andry Rakotonirainy. Context-oriented programming. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, pages 9–16, New York, NY, USA, 2003. ACM.

[41] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[42] JeongGil Ko, Chenyang Lu, M.B. Srivastava, J.A. Stankovic, A. Terzis, and M. Welsh. Wireless sensor networks for healthcare. *Proceedings of the IEEE*, 98(11):1947–1960, Nov 2010.

[43] P. Koopman. *Better Embedded System Software*. Carnagie Mellon Press, 2010.

[44] Tei-Wei Kuo and A.K. Mok. Load adjustment in adaptive real-time systems. In *Real-Time Systems Symposium, 1991. Proceedings., Twelfth*, pages 160–170, Dec 1991.

[45] J. Lach, W.H. Mangione-Smith, and M. Potkonjak. Algorithms for efficient runtime fault recovery on diverse fpga architectures. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT '99. International Symposium on*, pages 386–394, Nov 1999.

[46] Philip Levis and David Culler. Mate: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[47] Joseph P. Loyall, Paul Rubel, Richard Schantz, Michael Atighetchi, and John Zinky. Emerging patterns in adaptive, distributed real-time, embedded middleware, 2002.

[48] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. *SIGPLAN Not.*, 43(9):335–346, September 2008.

[49] PedroM. Martins, JulieA. McCann, and Susan Eisenbach. The environment as an argument. In Claudio Russo and Neng-Fa Zhou, editors, *Practical Aspects of Declarative Languages*, volume 7149 of *Lecture Notes in Computer Science*, pages 48–62. Springer Berlin Heidelberg, 2012.

[50] mBed. `www.mbed.org`.

[51] Marius Mikalsen, Nearchos Paspallis, Jacqueline Floch, Erlend Stav, George A. Papadopoulos, and Akis Chimaris. Distributed context management in a mobility and adaptation enabling middleware (madam). In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 733–734, New York, NY, USA, 2006. ACM.

[52] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.

[53] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. Figaro: Fine-grained software reconfiguration for wireless sensor networks. In *Proceedings of the 5th European Conference on Wireless Sensor Networks*, EWSN'08, pages 286–304, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 41(3):145–158, March 2007.

[55] W. Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. In *Proceedings of IEEE International Conference on Communications, ICC 2010, Cape Town, South Africa, 23-27 May 2010*, pages 1–6, 2010.

[56] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 489–498, New York, NY, USA, 2007. ACM.

[57] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[58] Bence Pásztor, Luca Mottola, Cecilia Mascolo, Gian Pietro Picco, Stephen Ellwood, and David Macdonald. Selective reprogramming of mobile sensor networks through social community detection. In *Proceedings of the 7th European Conference on Wireless Sensor Networks*, EWSN'10, pages 178–193, Berlin, Heidelberg, 2010. Springer-Verlag.

[59] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin. Jac: A flexible solution for aspect-oriented programming in java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 1–24, London, UK, UK, 2001. Springer-Verlag.

[60] Andrei Popovici. *PROSE: a study on dynamic AOP*. PhD thesis, ETH Zurich, 2003.

[61] C. Prehofer and M. Zeller. A hierarchical transaction concept for runtime adaptation in real-time, networked embedded systems. In *Emerging Technologies Factory Automation (ETFA), 2012 IEEE 17th Conference on*, pages 1–8, Sept 2012.

[62] Andreagiovanni Reina, Marco Dorigo, and Vito Trianni. Collective decision making in distributed systems inspired by honeybees behaviour. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '14, pages 1421–1422, Richland, SC, 2014. International Foundation for Autonomous Agents and Multiagent Systems.

[63] P. Richardson, L. Sieh, and A.M. Elkateeb. Fault-tolerant adaptive scheduling for embedded real-time systems. *Micro, IEEE*, 21(5):41–51, Sep 2001.

[64] Oscar Nierstrasz Robert Hirschfeld, Pascal Costanza. Context-oriented programming, March-April 2008.

[65] S. M. Sadjadi and P. K. McKinley. Act: An adaptive corba template to support unanticipated adaptation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, pages 74–83, Washington, DC, USA, 2004. IEEE Computer Society.

[66] S.Masoud Sadjadi, PhilipK. McKinley, BettyH.C. Cheng, and R.E.Kurt Stirewalt. Trap/j: Transparent generation of adaptable java programs. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1243–1261. Springer Berlin Heidelberg, 2004.

[67] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR*, abs/1105.0069, 2011.

[68] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *J. Syst. Softw.*, 85(8):1801–1817, August 2012.

[69] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Contexterlang: Introducing context-oriented programming in the actor model. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 191–202, New York, NY, USA, 2012. ACM.

[70] Sanjin Sehic, Fei Li, and Schahram Dustdar. Copal-ml: A macro language for rapid development of context-aware applications in wireless sensor networks. In *Proceedings of the 2Nd Workshop on Software Engineering for Sensor Network Applications*, SESENA '11, pages 1–6, New York, NY, USA, 2011. ACM.

[71] P.P. Shirvani. Fault-tolerant computing for radiation environments, 2001.

[72] Roland Siegwart and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Bradford Company, Scituate, MA, USA, 2004.

[73] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. Java&#8482; on the bare metal of wireless sensor devices: The squawk java virtual machine. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM.

[74] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM.

[75] W. Stevens et al. Classics in software engineering. chapter Structured Design. 1979.

[76] Lakshminarayanan Subramanian and Randy H. Katz. An architecture for building self-configurable systems. In *Proceedings of the 1st ACM International Symposium on Mobile Ad Hoc Networking & Computing*, MobiHoc '00, pages 63–73, Piscataway, NJ, USA, 2000. IEEE Press.

[77] Marcin Szczodrak, Omprakash Gnawali, and Luca P. Carloni. Dynamic reconfiguration of wireless sensor networks to support heterogeneous applications. *2013 IEEE International Conference on Distributed Computing in Sensor Systems*, 0:52–61, 2013.

[78] TinyOS 2.1.2. `www.tinyos.net`.

[79] S. Waharte and N. Trigoni. Supporting search and rescue operations with uavs. In *Emerging Security Technologies (EST), 2010 International Conference on*, pages 142–147, Sept 2010.

[80] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks*, IPSN '06, pages 416–423, New York, NY, USA, 2006. ACM.

[81] Virtex-II Pro and Virtex-II Pro X FPGA User Guide. Datasheet. UG012 (v4.2), 2007.

[82] Marc Zeller and Christian Prehofer. A multi-layered control approach for self-adaptation in automotive embedded systems. *Adv. Soft. Eng.*, 2012:10:10–10:10, January 2012.

[83] Ying Zhang and Krishnendu Chakrabarty. Dynamic adaptation for fault tolerance and power management in embedded real-time systems. *ACM Trans. Embed. Comput. Syst.*, 3(2):336–360, May 2004.

[84] Marco Zimmerling, Federico Ferrari, Luca Mottola, Thiemo Voigt, and Lothar Thiele. ptunes: Runtime parameter adaptation for low-power mac protocols. In *Proceedings of the 11th International Conference on Information Processing in Sensor Networks*, IPSN '12, pages 173–184, New York, NY, USA, 2012. ACM.