I Introduction

Wireless Sensor Networks (WSNs) are composed of a large number of tiny devices (known as “motes”) capable of sensing the environment and communicating. They are usually employed to continuously monitor physical phenomena in large or unreachable areas, such as wildfire in forests and air temperature in buildings. Each mote features limited processing capabilities, a short-range radio link, and one or more sensors (e.g. light and temperature) [2].

WSNs are usually designed with safety and (soft) real-time requirements under constrained hardware platforms. At the same time, developers demand effective programming abstractions, ideally with unrestricted access to low-level functionality. These particularities impose a challenge to WSN-language designers, who must provide a comprehensive set of features requiring correct and predicable behavior under platforms with limited memory and CPU. As a consequence, WSN languages either lack functionality or fail to offer a small and reliable programming environment.

System-level development for WSNs commonly follows one of three major programming models: *event-driven*, *multi-threaded*, or *synchronous*. In event-driven programming [23, 13], each external event can be associated with a short-lived function callback to handle a reaction to the environment. This model is efficient, but is known to be difficult to program [1, 14]. Multi-threaded systems emerged as an alternative in WSNs, providing traditional structured programming in multiple lines of execution [14, 9]. However, the development process still requires manual synchronization and bookkeeping of threads [30]. Synchronous languages [3] have also been adapted to WSNs and offer higher-level compositions of activities with a step-by-step execution, considerably reducing programming efforts [27, 28].

Despite the increase in development productivity, WSN system languages still fail to ensure static safety properties for concurrent programs. However, given the difficulty in debugging WSN applications, it is paramount to push as many safety guarantees to compile time as possible [32]. Shared-memory concurrency is an example of a widely adopted mechanism that typically relies on runtime safety primitives only. For instance, current WSN languages ensure atomic memory access either through runtime barriers, such as mutexes and locks [9, 33], or by adopting cooperative scheduling which also requires explicit yield points in the code [27, 14]. In either case, there is no additional static guarantees or warnings about unsafe memory accesses.

We believe that programming WSNs can benefit from a new language that takes concurrency safety as a primary goal, while preserving typical multi-threading features that programmers are familiarized with, such as shared memory concurrency. We present CÉU[1], a synchronous system-level programming language that provides a reliable yet powerful set of abstractions for the development of WSN applications. CÉU is based on a small set of control primitives similar to Esterel's [10], leading to implementations that more closely reflect program specifications. As a main contribution, we propose a static analysis that permeates all language mechanisms and detects safety threats, such as concurrent accesses to shared memory and concurrent termination of threads, at compile time. In addition, we introduce the following new safety mechanisms: *first-class timers* to ensure that timers in parallel remain synchronized (not depending on internal reaction timings); *finalization blocks* for local pointers going out of scope; and *stack-based communication* that avoids cyclic dependencies. Our work focuses on *concurrency safety*, rather than *type safety*.[2]

In order to enable the static analysis, programs in CÉU must suffer some limitations. Computations that run in unbounded time (e.g., compression, image processing) cannot be elegantly implemented [36], and dynamic loading is forbidden. However, we show that CÉU is sufficiently expressive for the context of WSN applications. We successfully implemented the *CC2420* radio driver [39], and the *DRIP*, *SRP*, and *CTP* network protocols [39][3] in CÉU. In comparison to *nesC* [19], the implementations reduced the number of source code tokens by 25%, with an increase in ROM and RAM below 10%.

The rest of the thesis is organized as follows: Chapter II introduces CÉU through comparisons with state-of-the-art languages representing the prevailing concurrency models used in WSNs. Chapter III details the design of CÉU, motivating and discussing the safety aspects of each relevant language feature. Chapter IV presents two demo applications, exploring the safe and

[1]Céu is the Portuguese word for *sky*.

[2] We consider both safety aspects to be complementary and orthogonal, i.e., type-safety techniques [11] could also be applied to CÉU.

[3] *DRIP* is a data dissemination protocol to reliably deliver data to every node in the network. *SRP* is a routing protocol to deliver packets from an origin node to a destination node. *CTP* is a collection protocol to deliver packets from any node to a collection of roots in a network.

high-level programming style promoted by the language. Chapter V evaluates the implementation of some network protocols in Céu and compares some of its aspects with *nesC* (e.g. memory usage and tokens count). We also evaluate the responsiveness of the radio driver written in Céu. Chapter VI presents a formal semantics of Céu restricted to its control primitives, which comprises most novelties and challenging parts of our design. Chapter VII discusses key aspects of the implementation of Céu, such as the static analysis algorithm and stackless lines of execution. Chapter VIII discusses related work to Céu. Chapter IX concludes the thesis and makes final remarks.