Journal of
Sensor and Actuator Networks

MDPI

Article

A Statecharts-Based Approach for WSN Application Development

Ismo Hakala *,† **and Xinyu Tan** †

Kokkola University Consortium Chydenius, University of Jyväskylä, P.O. Box 567, FI-67701 Kokkola, Finland; tan.xinyu@chydenius.fi

* Correspondence: ismo.hakala@chydenius.fi

† These authors contributed equally to this work.

Received: 1 July 2020; Accepted: 21 September 2020; Published: 25 September 2020

Abstract: Wireless Sensor Network (WSN) software development challenges developers in two main ways: through system programming, which requires expertise in hardware and network management; and application programming, which requires domain-specific knowledge. However, domain programmers often lack WSN programming expertise. Likewise, system-specific programmers may find it difficult to understand domain-specific requirements. As a result, domain programmers often refrain from using WSN technology in domain-specific applications. Therefore, we propose a Finite State Machine (FSM)-based approach with an affiliated framework to decouple application functionality from WSN details. Instead of the traditional flat FSM, we use statecharts formalism because of its relaxed definition of system states. In this paper, we compare the statecharts paradigm against two basic WSN sensor node programming frameworks. The result exhibits that statecharts are an advanced paradigm in WSN application development. It motivated us to develop a statecharts framework. In our framework, we choose not to use the typical solution which converts statecharts to programming code. Instead of that, we implement a statecharts middleware associated with action libraries to interpret and actuate raw statecharts on an operating system. This approach allows domain programmers to concentrate on WSN application behavior, and system-specific programmers to focus on developing WSN services. We also introduce our statecharts middleware and present a living example with performance evaluation.

Keywords: statecharts; WSN; programming framework; wireless sensor; WSN application

1. Introduction

The Internet of Things (IoT) represents the tendency toward a future of ubiquitous intelligence. A European Commission study [1] estimated that the market value of IoT in the European Union would exceed one trillion euros in the next decade. Wireless Sensor Network (WSN) is a widely used communication solution that composes IoT units from multiple sub-units. This technology allows smart devices to collaborate and appear as one IoT object over the Internet.

In WSN application development, decoupling the system complexity and the domain-specific application is an increasing topic of study. This method leaves low-level implementation and network management to WSN system programmers and concentrates domain experts on developing application behavior according to domain specifications. Early studies attempted to establish a WSN-oriented Operating System (OS), such as TinyOS [2] and Contiki [3], that integrated hardware drivers, kernel functions, and network management services. These operating systems often provided system-specific interfaces. Application programmers must program using system-specific language or variants (e.g., C and nesC [2]). Nevertheless, WSN OS application development still requires decent programming skills and knowledge of the host OS.

To further decouple applications from system complexity, some research uses middleware to link applications and operating systems by introducing highly abstract interfaces for application programming. This alternative solution bypasses the traditional way of coding with system-specific languages. Examples include Structured Query Language (SQL) and SQL-alike languages solutions (e.g., SINA [4], TinyDB [5], and Cougar [6]). Some other solutions transplant high-level programming languages to nodes via middleware, such as Python of PyFUNS [7] and Java of Magnet OS [8]. Moreover, Graphical User Interface (GUI) programming experience is a recent middleware tendency to practice visualized programming, such as [9,10].

A typical WSN application responds to internal and external events and is thus event-driven. Therefore, for also being event-driven, a Finite State Machine (FSM) is an ideal model to present WSN applications. A state machine describes how the functionality of a system relates to the state in which it exists at the time and how the system states change in response to events. This scheme of reacting to event e on state a if condition c is met, then transitioning to state b, well-fits the human thinking pattern. For example, SenOS [11] innovatively applies state-transition-table-represented applications, and Oasis [9] installs FSM-represented physical phenomena contexts as sensor node applications. Nevertheless, an FSM is composed of all possible combinations of states regarding the valid event input sequences. Hence, when it represents a "beyond simple" application, the state machine tends to end up with many states and transitions. Practically, the applications running in sensor nodes are usually simple, which justifies the use of FSM-based approaches. However, system complications may arise if data processing, accumulating, or filtering of multiple sensors are involved (e.g., applications in the cluster head or the gateway). Programmers may find such FSMs being difficult to maintain. Thus, to alleviate this problem and to effectively represent a more complicated WSN application context, we propose a variant of Harel's statecharts formalism [12]. Harel's statecharts formalism extends FSM model by introducing a relaxed state definition which allows hierarchical and orthogonal composition of states, history states, and entry and exit actions of states. In addition, inspired by Object State Model (OSM) [13], our statecharts variant accepts both global and local variables. These variables are useful for storing the system historical state and bridging data exchanges between operations. The hierarchical state structure allows for a super-state encapsulating the cross-functional sub-state machines. This preserves the rest of state machine from the interference of state transitions occurring inside a super-state. The orthogonal region regulates the independency and concurrency between sub-state machines. Furthermore, the graphical logical relationships between states, events, and actions confirm the adage that "a picture is worth a thousand words".

Statecharts are often used as a tool to guide the coding progress. Alternatively, an advanced usage is to input statecharts to compatible code generators, such as implemented in [14]. However, the code produced using the code generator is highly platform-dependent, difficult to maintain, and potentially deviates from the original intentions. Inspired by SenOS [11,15], we propose using statecharts as a context that describes the scenarios of invoking actions over associated event occurrences. Therefore, a statecharts application effectively serves as a manual that guides an operating system's response to events. To support this mechanism, we implemented a statecharts middleware that includes a statecharts interpreter and an action library implementation. As a result, the raw statecharts context running on board, is reflected by the scenario of invoking specific actions over event occurrence schema.

When using this approach, a developer begins by plotting the application on a Graphic User Interface (GUI) design software. This software provides WSN-oriented action libraries, including commonly used actions and events. The design is textually represented in JavaScript Object Notation (JSON) [16] format because it is object-based. Before distributing a statecharts application, its JSON scripts are further compressed to a binary expression to reduce the size. The application can be distributed to the wireless sensors via cable or radio channels. The compressed binary scripts are actuated in the middleware upon an operating system. Thus, it is decoupled from the hosting platform.

Therefore, it is easier to perform on-the-fly application management operations via middleware remote protocols.

Statecharts formalism is a familiar model to experts from different domains, such as industrial automation engineers. A novice can use the statecharts approach to program a WSN application without much difficulty. This approach also allows college students to experience IoT technology without advanced embedded programming courses. Moreover, statecharts' actions also instruct system programmers in which platform-dependent functionalities need to be implemented.

This paper contributes to the literature by demonstrating that statecharts are an advanced paradigm in WSN domain-specific programming and by introducing a statecharts-supported framework. We offer a real-life case study using the statecharts approach and performance evaluation.

The sections of this paper continue as follows. In Section 2, we review related studies and their approaches to programming WSN applications. In Section 3, we discuss basic WSN programming frameworks and compare them to the statecharts paradigm. We introduce the statecharts approach and affiliated framework in Section 4. The statecharts approach is evaluated with a living example in Section 5. It is followed by a discussion and concluding remarks in Section 6.

2. Related Works

IoT technology has potential in many fields, including environmental monitoring [17], medical surveillance [18,19], and military surveillance [20]. Developing domain-specific IoT end-devices requires the participation of domain experts other than WSN programmers. As a result, it attracts many research groups proposing programming approaches for WSNs which, in turn, empowers domain engineers to customize their applications easily.

Bensaleh et al. [21] suggested categorizing WSN development approaches into low-level and high-level. Low-level approaches program application behavior using a step by step instruction sheet (i.e., imperative code). High-level approaches model development steps and frameworks associated with descriptive languages and aim to ease the development process.

Many early works proposed low-level approaches that provided WSN-oriented operating systems with node-level abstract interfaces. The applications of these OSes are often programmed with system-specific languages and their dialects. Although most WSN operating systems (e.g., Contiki, Mantis OS [22], openWSN [23], and SOS [24]) use C language, some apply C dialects. They are, for example, nesC of TinyOS, LiteC++ of LiteOS [25], C++ of RIOT [26], and Ansi C of uC/OS-III [27]. Other OSes apply alternative programming languages through middleware, such as Java of Megnet OS, instruction stack alike language of Maté [28] and Agilla [29], and python of PyFUNS.

Macro-programming is another popular low-level approach: it organizes and operates network devices as one or more groups. For example, TinyDB, SINA, and Cougar treat the WSN network as a database. They apply Structured Query Language (SQL) alike languages to access wireless sensor information. Ambient awareness [30] and service discovery [9] techniques allow efficient information exchanges in neighbor nodes. Moreover, a group of nodes can be addressed through cluster-based abstract interfaces, such as Hood [31], abstract region [32], and EnviroTrack [33]. Furthermore, SAN-logic [34], WSN Virtualization [35], and Khalid et al. [36] access WSN sensing units via sensor virtualization.

Application development with low-level approaches typically provide interfaces that use imperative programming languages designed for WSN specialists use. As a result, domain programmers with insufficient expertise will have to invest much time in learning when applying low-level approaches.

High-level approaches include design process modularization, model-based user interfaces with highly abstraction, and facilitated frameworks. Model-Driven Development (MDD) [37,38] and Model-Driven Engineering (MDE) [39] approaches are proposed to involve domain-specific experts in WSN application development. Aided by automatic code generation technology, system-specific programmers provide code generation models used in the development, including platforms, network

management, Quality of Service (QoS), and sensor services. Thus, domain programmers can use descriptive modeling languages (e.g., Domain-Specific Language (DSL) [40] and MARTE (Modeling and Analysis of Real-Time and Embedded systems) profile [41]) to instruct the code generator to produce platform-specific programs. Depending on the code generator's platform configuration, programs can be coded by C, nesC [40], or Python [10]. However, the generated programs tend to lack platform heterogeneity support. Furthermore, the automatic code generator appears to be a black box to domain programmers. Therefore, the yielded code could potentially deviate from the original design.

Other high-level approaches actuate descriptive code in nodes directly by implanting interpretive middleware in the node system. Their applications instruct the system to execute function models in response to event occurrences. For example, Kerasiotis et al. [42] packs system services into function blocks. These functions blocks are linked by assembly-alike instruction stack and distribute to the nodes as mobile agents.

The FSM computational model is also a strong candidate for WSN application development approaches. It not only has visual programming schemes and shows clean logical relationships between functional modules, but it also has an event-driven paradigm intuitively well-fitting in WSN application cases. Applying FSM as a programming model, SenOS represents an application's behavior in a state transition table, which executes system library functions as reactions to associated events. As a result, SenOS applications can be reconfigured on-the-fly. Nevertheless, this approach seems lacking support of variables and action parameters. Moreover, state transition tables are not flexible or efficient because they are typically sparse and waste the sensor node's limited memory resource.

OSM integrates descriptive language to achieve a statecharts-alike paradigm and uses the relaxed state definition alongside the in-state variables. OSM's approach effectively expresses a practical problem in a compact state machine. OSM equips two-level code generators to produce a platform-dependent code in C. As a result, OSM applications are compromised in portability.

Inspired by SensOS and OSM, our proposed middleware accepts a strategy that actuates the raw statecharts on board. We also provide a GUI design software with an action library collection for visual programming experience. We believe that this will allow novice programmers to use WSN applications easily.

3. Wireless Sensor Programming Frameworks

Depending on the hosting operating system, a WSN application programming framework falls into thread-based and event-driven models. For instance, TinyOS and Contiki native support event-driven modeling and extensively support thread-based modeling through middleware; SOS [24] supports event-driven modeling while LiteOS and Mantis OS support thread-based modeling.

This section demonstrates a simple case, which is extracted from our real-life application of the SmartHome application. We first present the programs using thread-based modeling and event-driven modeling in C language (some pseudo-system functions used for easier understanding). Then, we demonstrate our statecharts design with this case. The case scenario consists of sampling a motion-sensor detection result in 5 s period for 10 min and performs associated data-handling procedures (e.g., send to gateway).

3.1. Thread-Based

Thread-based is a familiar programming framework for programmers, as it is supported by many computational operating systems, such as Linux and its large number of variants. The program is processed in a logic flow sequence. A thread-based modeled program has advantages in understanding and maintenance. However, a single-threaded program is impractical due to difficulties in handling concurrent event occurrences. Fortunately, many operating systems provide multi-threaded solutions. In a multi-threaded supported operating system, multiple threads can co-exist concurrently. Because of this, multi-thread programming often requires pre-allocating memory stack space for context saving.

The amount of preserved memory is empirical; i.e., it needs to be sufficient to avoid a stack overflow problem yet not too much to leave unnecessary redundant memory. Furthermore, these operating systems also provide threading management interfaces, typically *create*, *terminate*, *suspend*, and *resume*.

Listing 1: Thread-based solution

```
1  #define N  120
2  bool flag_motion_detected = false;
3
4  void motion_thread(void)
5  {
6    motion_activate();
7    while(!motion_is_detected());
8
9    flag_motion_detected = true;
10   motion_deactivate();
11 }
12
13 void main_thread(void)
14 {
15   short buf[N];
16   int i = 0;
17   Thread_t th;
18   Timer_t tmr;
19
20   do {
21     thread_create(&th, motion_thread);
22     set_timer(&tmr, 5 * CLOCK_SECOND);
23     buf[i] = 0;
24
25     while(!timer_is_expired(&tmr));
26
27     if (flag_motion_detected) {
28       buf[i] = 1;
29       flag_motion_detected = false;
30     }
31     else {
32       motion_deactivate();
33       thread_terminate(&th);
34     }
35
36     if (++i == N) {
37       /* handle buffer */
38       /* ... */
39       i = 0;
40     }
41   } while (1);
42 }
43
44 void main(void)
45 {
46   Thread_t th;
47   thread_create(&th, main_thread);
48   /* ... */
49 }
```

The programming strategy under thread-based modeling in the given case is to create two threads. In the Listing 1 snippet, the *main_thread* is dispatched (in line 47) to the scheduler. It provides an endless loop in 5 s. At the beginning of the loop, a *motion_thread* is created (in line 21) to monitor the motion sensor. The buffer records a 0 as no motion is detected by default. If a motion signal is detected, the global flag is set to the situation (in line 9). When the timer expires (in line 25), the flag determines the motion detection result (in line 27). If a motion is detected, it records a 1 to the buffer. Otherwise,

it means that the motion thread is still alive and needs to be killed. Eventually, the buffer will be handled when it is full (in line 36–40).

There are sensor hardware-specific functions used in the code snippet, including *motion_activate* (in line 6), *motion_deactivate* (in lines 10 and 32), and *motion_is_detected* (in line 7). The programmer needs to implement them according to the thread-based code modeling.

Listing 1 shows clear logic relationships between the functional sections. However, the programming procedure also reveals that practicing a thread-based framework requires significant knowledge of the operating system. For example, the programmer needs to understand the multiple thread concurrency mechanism as well as thread management. A flow control technique is performed to control program behavior based on history. Herein, the flag (in line 27) is used to determine the motion detection result, therefore branching the program's next moves.

3.2. Event-Driven

An event-driven framework is a natural pattern fitting in WSN application scenes, as they usually wait for the internal or external stimuli to react correspondingly. The schema typically invokes associated actions in responding to the event occurrence. The actions are de facto callback functions assigned to related events. Furthermore, they are event blocking and run-to-complete. Therefore, a programmer should avoid implementing long-lived actions as such actions potentially prevent incoming events and lead to program malfunction. Practicing event-driven modeling requires a programmer to deconstruct the program into actions then assign them to associated events. The sequential order of executing assigned actions depends on the timing of event occurrence. It is unambiguous to observe from the code and causes potential logic conflict.

Listing 2 snippet applies event-driven modeling to implement the given case. It begins with starting the first cycle by setting a 5-second timer and assigning a timeout event to *timeout_callback* (in line 33). The motion sensor is activated, and *motion_callback* is assigned to a motion-detected event (in line 34). When a motion-detected event is triggered, the flag is set to record this occurrence (in line 9). On the other hand, invoked by a timer-expiring event, a *timeout_callback* handles the buffer, reactivates the motion sensor depending on the flag status (in line 22), and starts a new cycle.

Alike Section 3.1, event-driven example code also requires the implementation of sensor hardware functions, including *motion_activate* (in lines 23 and 34) and *motion_deactivate* (in line 10. Contrarily, the implementation needs to adapt to event-driven modeling.

Listing 2 shows that stack management needs to be considered to reserve data. Because all actions share one stack in an event-driven framework, the context is not preserved after the action's exit. One must pre-allocate global memory space for sharing data between actions (in lines 3–5). Moreover, flow control is once again involved, similar to what was discussed in Section 3.1. The program determines a motion-detected event occurrence by checking *flag_motion_detected* (in line 17).

3.3. Statecharts

Statecharts formalism is an FSM variant, designed to present a state machine in a compact representation. Statecharts are well suited to solve WSN-related problems due to their event-based nature. The visualized presentation of statecharts supports the design of the application well.

When plotting statecharts, we use rounded rectangles boxes denoting *state*, while expressing the hierarchy relations by encapsulation. An arrow indicates a *transition* between states with the same depth. The arrow originates from the source state and terminates in the destination state. An arrow is labeled with an event and optionally a square bracketed condition. The Mealy machine-alike actions are listed after the event and conditions separated by a slash. On event occurrence, the system state transits when the condition is met. The state machines of statecharts are contained in a rectangle frame boxing a statecharts *region*. Only one first-level state machine is allowed

in a region. It ensures the independence of the statecharts. A dotted end arrow denotes a default state. A state machine is initialized to be in the default state if not otherwise specified.

Listing 2: Event-based solution

```
#define N  120

bool flag_motion_detected = false; {
short buf[N];
int i; {

void motion_callback(void)
{
  flag_motion_detected = true;
  motion_deactivate();
  buf[i] = 1;
}

void timeout_callback(void)
{
  if (++i == N) {
    /* handle buffer */
    /* ... */
    i = 0;
  }

  if (flag_motion_detected) {
    motion_activate(motion_callback);
    flag_motion_detected = false;
  }

  set_timer(5 * CLOCK_SECOND, timeout_callback);
  buf[i] = 0;
}

void main(void)
{
  set_timer(5 * CLOCK_SECOND, timeout_callback);
  motion_activate(motion_callback);
  buf[i] = 0;
  /* ... */
}
```

Both regions and states allow the declaration of variables and activities. The scope of variables is bound by the domain of the state or region. An activity is functionality with live time, such as "beep" or "take average temperature over time". An activity is activated when entering a state or region and is deactivated when leaving one. Compared to an activity, an action is an operation that ideally takes zero executive time. It applies to states and transitions. A state allows actions both in entering a state and exiting a state, and a transition executes actions after the event is validated.

Abstract actions and activities used in statecharts are independent of the platform and implemented by system-specific programmers. In using a statecharts model, one is capable of programming wireless sensors with little knowledge of them.

Designing statecharts begins with identifying states of the system. In the given case, shown in Figure 1, the system is in a *Motion sensing* (state 1) state of sampling motion data after the initialization state (not shown). This state repeats periodically, driven by a timer-expired event *evtTimer*. On event occurrences, the event guard validates the transition. Therefore, *evtTimer* triggered transition on the right is taken when the buffer is not full, judging by condition $[i < N - 1]$; otherwise, the bottom transition is taken. Hosting by *Motion-sensing* state, a sub-state machine reflects motion detection results and acts accordingly. In each sampling cycle, it is evident that no motion is detected as at the beginning, as in *No motion* (state 2) state. Detecting a motion event moves a sub-state machine to *Motion* (state 3) state and records a 1 in the buffer.

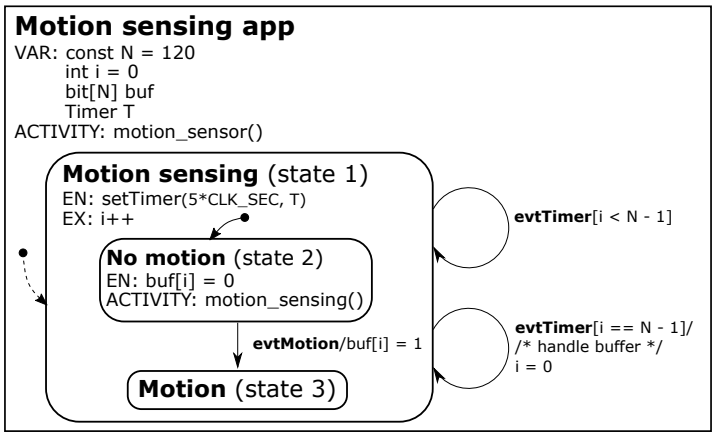

Figure 1. Statecharts demonstrating motion sensing.

In this example, there are two sensor hardware-specific actions/activities from the action library. The *motion_sensor()* activity initializes motion-sensor functionality when entering *Motion-sensor app* region, then destroys the motion-sensor process when leaving the region. Likewise, *motion_sensing()* activity enables motion detection progress when entering *No motion* (state 2) state, and disables when leaving it. The actions and activities from the action library are implemented in advance by system-specific programmers. Moreover, to implement domain-specific actions/activities, the system-specific can swiftly develop them without domain-specific knowledge.

The statecharts paradigm reacts to events by transiting the system from one state to another state. A valid event invokes corresponding actions during state transition. This mechanism of statecharts supports event-driven paradigm. In contrast to thread-based frameworks, event-driven frameworks align with WSN application scenarios; however, traditional event-driven frameworks do not clearly communicate how systems respond to event sequences (i.e., multiple events may occur in different orders or simultaneously). Moreover, the executive order of all functional modules in statecharts is presented clearly in a "thread-based way."

Statecharts exhibit a coherent relationship between states and events. They do not require a thorough understanding of the platforms or operating systems. Furthermore, the graphical syntax of statecharts visualizes the problem, aiding in afterward validation and troubleshooting.

In the example, the reader may observe that the statecharts influence the hosting platform and surroundings only through action libraries' actions. This mechanism effectively divides domain specifications and system specifications. The system experts would extend existing libraries on domain programmers' requirements. Contrarily, other system-specific language approaches, like the afore-mentioned thread-based and event-driven examples, require deeper cooperation between domain-specific and system-specific programmers to accomplish an application program.

4. Statecharts Approach

The statecharts approach aims to improve the WSN application development experience by distinguishing domain-specific programmers and system-specific programmers. Figure 2 demonstrates collaboration between a domain-specific programmer and a system-specific programmer. A domain programmer designs a statecharts application using the statecharts editor. The application uses abstract actions from the action abstract interface library to actuate WSN sensor nodes. Meanwhile, a system programmer implements action library collections in the statecharts middleware. The implemented actions will be automatically synchronized to the statecharts framework database. However, there are always application-specific actions unaccounted for by common library collections. The domain programmer and the system programmer would collaborate to design the application-specific abstract interfaces. Thus, the system programmer would implement these functions in the middleware, while the domain programmer would be back to the statecharts design.

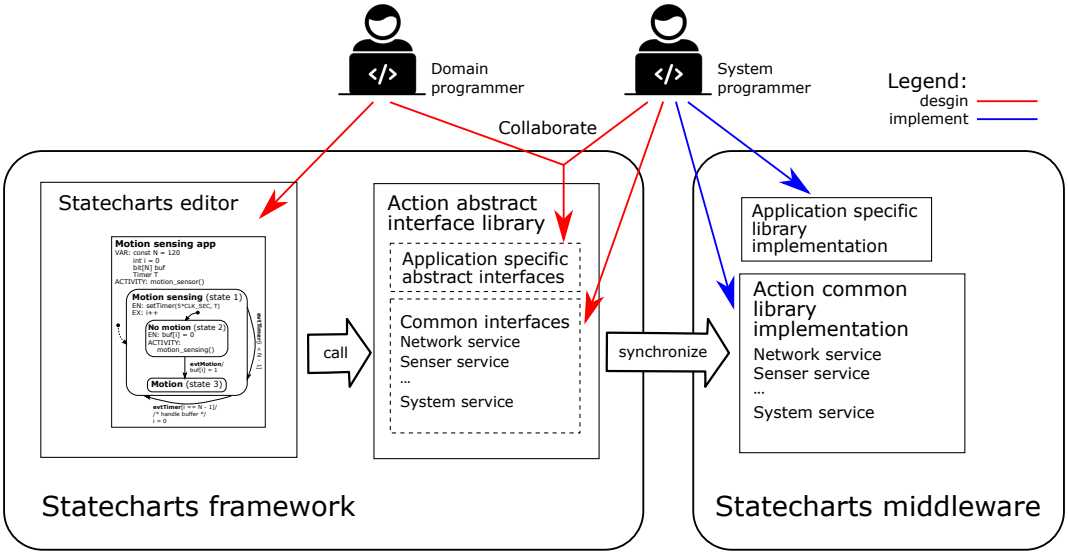

Figure 2. The statecharts approach.

Statecharts showcase logical relationships between events and actions in a visual format. It helps domain experts without a strong programming background to understand applications quickly. So they can participate in the application design progress. After all, applications serve domain-specific requirements, and the domain experts know the best of them. In contrast, traditional application development approaches often employ system-specific programming languages, which only some group members may understand. This problem may yield potential risks when validating collected data because the application may not be understandable to the domain experts.

Because Statecharts applications are interpreted and actuated in middleware as descriptive scripts, they are independent of the platform. Therefore, an application can be distributed across different platforms, while platform heterogeneity is handled by middleware. Moreover, deployed applications are re-configurable by application context modification.

We implemented an affiliated framework for the statecharts approach. The framework includes a statecharts editor, an abstract action interface library, and a statecharts compressor. To actuate statecharts applications in nodes, we implemented a statecharts middleware upon platforms as a run-time environment.

4.1. Statecharts Editor

The statecharts editor is a web-based application (Figure 3), which is implemented to aid in the statecharts design process. A programmer can plot the statecharts in the canvas while the editor automatically suggests events and actions that can be used. The statecharts editor holds a list of actions and events that are implemented in the action library. The list is automatically updated once new actions being implemented by the system programmers.

Listing 3: Statecharts file capture

```
{
  "/REGION": {
    "property": {
      "alias": "Door legacy",
      "id": 2 },
    "/INITSTATE": 0,
    "/VAR": [
      { "display": "const SenID = 58",
        "id": 0,
        "type": "const",
```

```
        "initial": 58 }
    ],
    "/ACTIVITY": [
      { "id": 4866,
        "display": "sensor_acitivity(SenID)",
        "parameter": [{"id": 0}] }
    ],
    "/STATE": [
      { "property": {
          "alias": "State0",
          "id": 0 },
        "/TR": [
          { "/TARGET": 0,
            "/ACTION": [
                { "id": 4870,
                  "display": "sensor_submit_gateway(SenID)",
                  "parameter": [{"id": 0}] }
            ],
            "/EVENT": [
             {  "alias": "evt_sensor_activity[id == SenID]",
                "id": 33795,
                "condition": [
                  { "display": "evtVal(0) == SenID",
                    "comparison": "==",
                    "lexp": [{ "evtVal": 0 }],
                    "rexp": [{ "id": 0 }] }
            ...
```

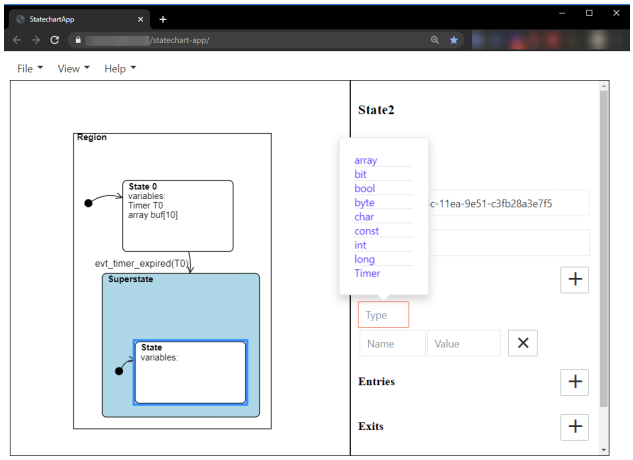

Figure 3. Statecharts editor snippet.

Once a statecharts application design is completed, it is saved in an object-based JSON format. In this format, statecharts objects maintain the same hierarchical order as its graphical presentation. Listing 3 depicts a snippet of a door sensor application from one of our real-life cases. In this format, the statecharts components are labeled by a leading slash with a capitalized label name. The components may contain several attributes and objects. Statecharts editor can also load a statecharts JSON file and present it visually in the canvas.

Being string-based, JSON is considerably heavy for sensor nodes, since they are limited in the resource. To this end, a statecharts compressor converts a statecharts context from JSON to a binary-based representative. It significantly trims the size of context (e.g., the original Listing 3 text is

1627 bytes, compressed to a 39 bytes binary-based format). Eventually, the compacted statecharts are distributed by using Over-The-Air (OTA) protocol to the sensor nodes.

4.2. Statecharts Middleware

Statecharts application context uses descriptive language, but operating systems usually accept system-specific programming language. Therefore, it requires a specialized middleware between them as a liaison. Figure 4 shows the middleware implemented to interpret statecharts application context to a series of scenarios of actions invoking over events occurrences. The middleware reacts to the events and invokes corresponding actions from the action library regarding associated statecharts application context.

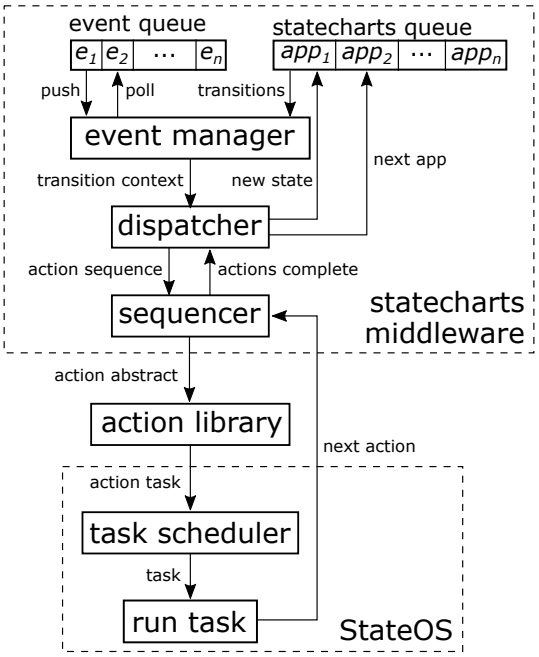

Figure 4. StateOS middleware.

Statecharts middleware uses a *statecharts queue* to manage installed applications. The multiple applications' concurrency is attained by actively switching between the statecharts. Each statecharts application has a data structure that stores relative context, such as current state and transition progress.

An *event queue* pushes events to the *event manager* assigning them to transition candidates in the *statecharts queue*. Validated transitions are forwarded to the *dispatcher*, which initiates transition progress. During this progress, incoming events are placed in the *event queue*. The *dispatcher* interprets the transition context and extracts the abstract statements and actions from the context. These statements and actions are given to the *sequencer* in an executive order.

The *sequencer* invokes the *action library's* associated actions and ensures that they are run-to-the-end without being interrupted by other actions. When an action is complete, it invokes the next action in the sequence. Upon completing all actions, the *sequencer* indicates the *dispatcher* completes the transition process and refreshes the state configuration.

This process is repeated until all statecharts in the queue are processed. When complete, the *event manager* checks the *event queue* for the events that occurred during the procedure.

4.2.1. Action Library

The actions of statecharts describe the computational behavior of an application. The comprehensiveness of actions in the action library defines statecharts applications' limitations. Actions in the action library are categorized as network actions, data processing actions, sensor service actions, and

system actions. Using these actions, a domain programmer can develop statecharts applications for most WSN cases.

Application-specific actions can be added to the action library. Domain-specific programmers define the interfaces and behaviors of these actions and outsource their implementation to system programmers. Their collaboration would be smooth and sufficient.

4.2.2. File System

To managing statecharts application, statecharts middleware implements a file system that saves statecharts to different media by choice. For example, the file system can work with hardware self-programming functions to save larger statecharts applications in ROM/flash memory to save spaces. On the other hand, response-time-sensitive applications are run directly from the ROM memory to achieve better performance.

4.2.3. OTA

The statecharts Over-The-Air (OTA) distribution module allows automatic software distribution and updating. The OTA module can efficiently transmit statechart applications to the remote nodes with a low energy cost because of statecharts application's feasible size.

In our solution, a new sensor node queries an application list from its predecessor. The list is composed based on the sensors installed and their job description, including the application's name, version, and digital signature. If a node lacks the listed applications, it acquires them from its predecessor. Also, if the predecessor lacks the required application, it contacts higher level predecessors until it finds the application.

4.2.4. Supported Platform

Statecharts middleware was designed to support multiple platforms; however, it is currently only implemented for StateOS, a WSN OS based on Hakala and Tikkakoski's former work [43]. StateOS applies a cross-layer design to reduce messaging overhead effectively. It applies micro-kernel architecture, in which a hybrid task scheduler (inspired by [44]) is implemented. There are two task queues in the scheduler, as preemptive queue and cooperative queue. They both apply cooperative ordering manner in their respective queues. The preemptive queue tasks are privileged to preempt cooperative queue tasks. This scheduling strategy preserves sufficient real-time capability for WSN application while occupies only one extra memory stack for storing task context. In contrast, traditional preemptive scheduling requires reserving a memory stack for each task or thread. Moreover, StateOS also provides macro-based abstract flow control interfaces, which support both event-based and multi-thread models.

5. SmartHome Application and Evaluation of the Statecharts Approach

The SmartHome application is an example of how the statecharts approach can be used in a daily life application. This application aims to provide in-house information regarding senior people's well-being to home-care nurses [45]. The application's design adopts an IoT architecture, including a wireless sensor network, a gateway, and cloud services. The application was developed step by step over three years. The statecharts approach was introduced to our development process for group communication purposes. By using statecharts, our domain experts understand the WSN application easily. Furthermore, they can make suggestions and validate whether the application meets the domain-specific requirements.

One objective of the SmartHome application [46] was to recognize the participant's daily movement patterns. To obtain the participant's movement data, we implemented wireless sensor nodes in the participant's apartment. Each node was equipped with a passive infrared (PIR) sensor. We deployed these nodes in each room of the apartment, monitoring the activity of the participant.

The design of the application is similar to the example provided in Section 3. Figure 5 shows our sensor application statecharts. In the statecharts region, we declare some global variables and activate a motion-sensor service. Thus, the statecharts are in *wait sensor ready* state (i.e., it is a default state to the region), which waits for the *evtMotionReady* event. The motion-sensor service triggers the *evtMotionReady* event, which transits the statecharts to the *Motion-sensing* super-state.

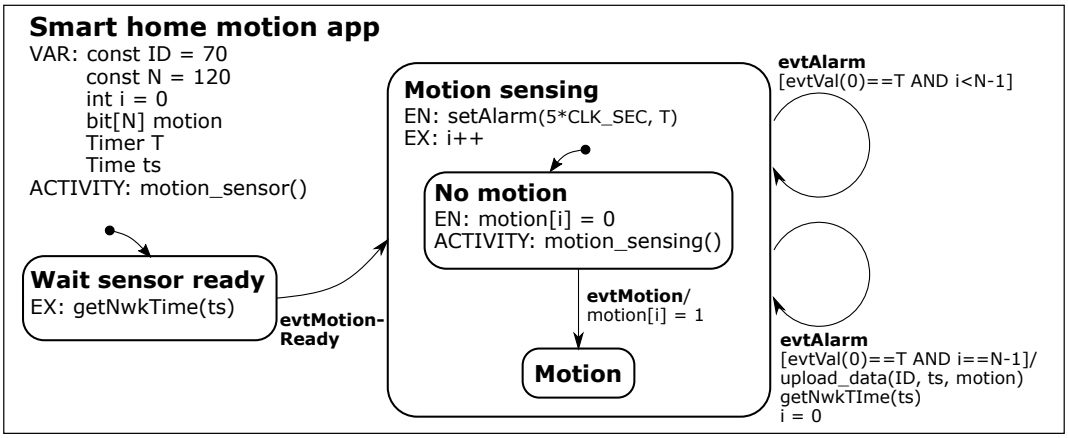

Figure 5. Motion-sensing statecharts.

The *Motion-sensing* super-state is recursive and periodically records motion-sensing data and uploads them to a gateway. The *no motion* state is a default sub-state of *Motion-sensing* super-state, which initializes the motion-data buffer and activates a motion-sensing activity. When a motion activity is sensed, we record the new motion status in the buffer and deactivate the motion-sensing activity by transiting the statecharts to the *motion* state.

This motion statecharts application is compressed (to 154 bytes) and downloaded to a WSN sink (as an access point to WSN). The sink initiates OTA protocol distributing the application to all motion-sensor nodes, as shown in Figure 6. The collected motion data from different sensor nodes is accumulated, encoded, and compressed in the sink and upload to the Internet using a LoRa [47] gateway.

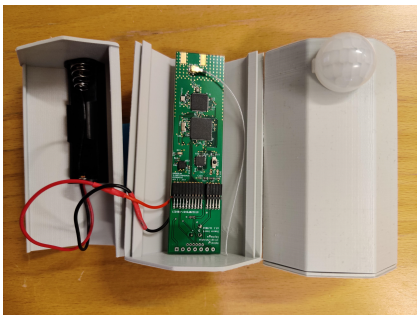

Figure 6. Motion-sensing hardware.

Evaluation of the Statecharts Approach

To evaluate the statecharts application's performance, we compare the SmartHome motion activity application to an equivalent thread-based C program. Both programs are examined on the same platform, including the same operating system, StateOS. A difference is that statecharts application is actuated upon middleware, while the flat-coded program runs on StateOS directly. Therefore, the executive time and the use of data memory of both approaches can be evaluated related to each other.

Table 1 presents the executive time measurements of both programs. In general, the executive time of responding to an event in statecharts approaches is two to three times slower than flat-coded

approaches. Nevertheless, we argue that since both approaches have an executive time in milliseconds, the difference is negligible (especially considering that a typical wireless sensor active period less than one percent of its lifetime). Also, in some situations (e.g., the *wait sensor ready* stage shown in Table 1), because the relationship between the state and event is already established in the statecharts context, it negates the need for some of the statecharts' steps to register the event in the event management module. In contrast, the flat-code approach must always register the pending event to the event handler. Therefore, sometimes the statecharts approach is somewhat fast than the flat-code approach.

Table 1. Execution time comparison.

Stage	Statecharts (us)	Flat-Code (us)	Description
Initialization	1005	690	Initializes sensor
Wait sensor ready	19	358	Maintains suspension until *evtMotionReady* event occurs
evtMotionReady	413	233	Responds to *evtMotionReady* event
Motion sensing	987	243	Activates motion sensor and alarm timer
evtMotion	833	260	Responds to *evtMotion* event
evtAlarm (No motion)	1156	736	Responds to *evtAlarm* event when no motion detected
evtAlarm (Motion)	872	482	Responds to *evtAlarm* event when motion is detected

Both programs are stored in flash memory, the size of statecharts scripts is 154 bytes compared to 1268 bytes with the compiled flat-coded program. Statecharts scripts are descriptive and compressed in a compact format. It is generally smaller than the equivalent machine code. Statecharts scripts can be executed directly from flash memory but with an executive speed trade-off. Sometimes, in a performance preferred application, the statecharts scripts can be imaged and performed in data memory.

The data memory footprint of the statecharts is evaluated by recording the high water marks of memory usage in different stages. Table 2 shows that if the scripts are executed in flash memory, the statecharts consume less memory than the flat-coded program. Nevertheless, in the SmartHome application, we run statecharts in data memory to achieve better event response time. In this case, the statecharts actually use more memory compared to the flat-coded program.

Statecharts memory footprint mainly consists of three parts: variables, a control block, and statecharts scripts. A control block is a 16 bytes only data structure saving the statecharts' status and context. On the other hand, flat-coded programs need to allocate more memory to manage the contexts of related tasks and events by using the platform provided interfaces.

Table 2. Memory footprint evaluation.

Stage	Flat-Code (Bytes)	Statecharts (Bytes)				
		Scripts excl. [1]			Scripts incl. [2]	
	Total	Variable	Control	Total	Scripts	Total
Wait sensor ready	88	40	16	56	154	210
Motion sensing	88	56	16	72	154	226
Motion detected	136	104	16	120	154	274

[1] The statecharts are actuated in flash memory, so the scripts are not included in data memory. [2] The statecharts are actuated in data memory, so the scripts' in-memory image is taken to account in data memory.

The power consumption of a WSN application is related to the CPU active duty circle. Because the statecharts approach is about three times slower than the flat-code approach, practicing the statecharts approach will consume more energy to actuate an application. However, a typical WSN application is designed to sleep over 99% of the time. The extra consumed energy by longer executive time is affordable. The radio activity consumes a bigger portion of energy compared to CPU. The size of a statecharts application is about nine times smaller than the flat-code one. So it consumes less energy to transmit a statecharts application to another device over radio frequency.

The evaluation results reveal that the statecharts approach has a latency trade-off. Considering a typical WSN low-power radio takes tens of milliseconds to transmit a message [48]. The statecharts approach's millisecond-level executive time is sufficient in most of WSN applications. The statecharts control block can be considered a memory stack assigned to thread in a multi-threaded system. In multi-threaded OSes, it is empirical to assign at least 128 bytes memory stack for each thread to prevent stack overflow (e.g., freeRTOS [49] and Mantis OS). Therefore, statecharts' 16 bytes control block is favorable to WSN restricted memory resources.

Furthermore, to evaluate the usefulness of the statecharts approach, we compared the statecharts middleware against some of the well-known and state-of-the-art approaches in Table 3. In general, a high-level approach is user experience-oriented and provides a descriptive programming model with affiliated frameworks. Therefore, compared to low-level approaches, the statecharts approach, as a high-level approach, is more friendly to novice programmers.

Typically, many high-level approaches use descriptive language to program applications and use code generators to produce the system-specific programs. As we argued in Section 2, the generated programs are platform-depended, challenging to understand, and potentially deviating against the original design. Compared to the code-generating approaches, the statecharts application actuates the platform in a "what you see is what you get" paradigm by running the raw statecharts context directly on the device. Statecharts are an event-driven paradigm which better suits for WSN cases. Furthermore, the statecharts visual formalism would enhance user experience even better.

Table 3. Comparing the statecharts approach to other approaches

Approach	Level	Scripts Language	Paradigm	Code Generator	Visual Programming
Statecharts middleware	high	bit-wise JSON	statecharts	no	yes
SenOS	high	STT [1]	FSM	no	no
OSM	high	OSM lang.	OSM model	yes	no
PyFUNS	low	python	instruction stack	yes	yes
SenNet [40]	high	DSL	UML [2]	yes	no
Kerasiotis' approach [42]	high	instruction stack	function blocks	no	no
Modesene [37]	high	DSML [3]	MDD	yes	no
WSN Virtualization [35]	low	java	UML model	no	no
TinyDB	high	SQL-alike	database	no	no
OASiS [9]	high	TinyGALS [50]	FSM	no	no

[1] STT is short for State transition table. [2] UML is short for Unified Modeling Language. [3] DSML is short for Domain-Specific Modeling language.

6. Discussion and Conclusions

In this paper, we have argued that our statecharts approach effectively eases WSN application development. The development procedure is decoupled into domain-specific tasks and system-specific tasks. The system programmers were dedicated to providing quality action libraries

to be used in statecharts. Aided by the statecharts framework, the domain programmers developed a statecharts application to fulfill the field requirements.

Statecharts have a graphical syntax. The logical relationships between function modules are presented visually. Therefore, the studying curve of statecharts is smoother than in transitional programming approaches. Furthermore, the state-machine variants, such as FSMs and statecharts, are widely used instruments in scientific and industrial fields. The domain programmers from those fields could more easily implement a statecharts approach.

Team collaboration is utterly essential presently. A statecharts application is intelligible to all team members and requires little explanation. Thus, a team-wide discussion about a statecharts application can be conducted swiftly. It is beneficial if the field requirements are proposed by those without a programming background. The statecharts approach has the potential for early-stage IoT programming education. It does not require comprehensive knowledge of hardware, programming languages, or operating systems to practice IoT applications.

The evaluation results in Section 5 revealed that using the statecharts approach has an executive time trade-off about three times greater than in "flat coding". It is easy to understand that actuating a statecharts application requires extra steps compared to a system-specific application. Nonetheless, the difference in event response time using the statecharts approach is on the millisecond scale. It is sufficiently real-time for most WSN cases. Moreover, managing statecharts applications often involve extra context memory space compared to a system-specific program. The engaged context memory (16 bytes per application) is, in fact, smaller than a thread memory stack size in multi-thread OSes (e.g., typically 128 bytes in freeRTOS and Mantis OS). The statecharts application context is usually in stasis; therefore, the data addressing and reading speed is vital to overall performance. Depending on platforms, a statecharts application can be stored in the data memory (best performance but a limited resource), the program memory (mediocre performance but a sufficient resource), or other media. The choice relies on the application's preference to achieve better performance or compromise to budget data memory.

So far, our statecharts framework is still preliminary and under development. The editor and action libraries are preliminary. As shown in Section 5, motion-sensor actions are developed as application-specific functions rather than common sensor services. As we further develop the statecharts framework, the action library is expected to become more comprehensive for better WSN application usage.

This paper focuses on proposing a WSN programming approach. We only briefly introduced the statecharts notations and supported framework. We will present the details of our statecharts semantics and supported middleware in future publications. At the moment, our statecharts approach accepts only the statecharts middleware combined with StateOS. It is one of our future objectives to port statecharts middleware to other platforms (e.g., Arduino [51]).

Author Contributions: Conceptualization, I.H.; supervision, I.H.; software, X.T.; writing—original draft, X.T.; writing—review & editing, I.H.; All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: We declare that there is no conflict of interest.

References

1. Stefania, A.; David, B.; Martin, C.; Mike, C.; Philip, C.; Gabriella, C.; Sergio, G.; Giorgio, M.; Domenico, R.; Richard, S. Definition of a Research and Innovation Policy Leveraging Clou Computing and IoT Combination. *Study Prep. Eur. Comm. Commun. Netw. Content Technol.* **2013**, *95*. [CrossRef]
2. Levis, P.; Madden, S.; Polastre, J.; Szewczyk, R.; Whitehouse, K.; Woo, A.; Gay, D.; Hill, J.; Welsh, M.; Brewer, E.; et al. Tinyos: An operating system for sensor networks. In *Ambient Intelligence*; Springer: Berlin/Heidelberg, Germany, 2005; pp. 115–148.

3. Dunkels, A.; Gronvall, B.; Voigt, T. Contiki-a lightweight and flexible operating system for tiny networked sensors. In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, Tampa, FL, USA, 16–18 November 2004; pp. 455–462.
4. Srisathapornphat, C.; Jaikaeo, C.; Shen, C.C. Sensor information networking architecture. In Proceedings of the 2000 International Workshop on Parallel Processing, Toronto, ON, Canada, 21–24 August 2000; pp. 23–30.
5. Madden, S.R.; Franklin, M.J.; Hellerstein, J.M.; Hong, W. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. (TODS)* **2005**, *30*, 122–173. [CrossRef]
6. Demers, A.; Gehrke, J.; Rajaraman, R.; Trigoni, N.; Yao, Y. The cougar project: A work-in-progress report. *ACM Sigmod Rec.* **2003**, *32*, 53–59. [CrossRef]
7. Bocchino, S.; Fedor, S.; Petracca, M. Pyfuns: A python framework for ubiquitous networked sensors. In *European Conference on Wireless Sensor Networks*; Springer: Berlin/Heidelberg, Germany, 2015; pp. 1–18.
8. Barr, R.; Bicket, J.C.; Dantas, D.S.; Du, B.; Kim, T.D.; Zhou, B.; Sirer, E.G. On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Oper. Syst. Rev.* **2002**, *36*, 1–5. [CrossRef]
9. Kushwaha, M.; Amundson, I.; Koutsoukos, X.; Neema, S.; Sztipanovits, J. Oasis: A programming framework for service-oriented sensor networks. In Proceedings of the 2007 2nd International Conference on Communication Systems Software and Middleware, Bangalore, India, 7–12 January 2007; pp. 1–8.
10. Serna, M.A.; Sreenan, C.J.; Fedor, S. A visual programming framework for wireless sensor networks in smart home applications. In Proceedings of the 2015 IEEE Tenth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, 7–9 April 2015; pp. 1–6.
11. Hong, S.; Kim, T.H. Senos: State-driven operating system architecture for dynamic sensor node reconfigurability. In *International Conference on Ubiquitous Computing*; ACM: New York, NY, USA, 2003; pp. 201–203.
12. Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.* **1987**, *8*, 231–274. [CrossRef]
13. Kasten, O.; Römer, K. Beyond event handlers: Programming wireless sensors with attributed state machines. In Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, Boise, ID, USA, 15 April 2005; p. 7.
14. YAKINDU STATECHART TOOLS. Available online: https://www.itemis.com/en/yakindu/state-machine/ (accessed on 17 August 2020).
15. Kim, T.H. Design and Implementation of a State-Driven Operating System for Highly Reconfigurable Sensor Networks. *Int. J. Distrib. Sens. Netw.* **2013**, *2013*, 7.
16. JSON. Available online: https://www.json.org (accessed on 17 August 2020).
17. Suryadevara, N.K.; Mukhopadhyay, S.C.; Kelly, S.D.T.; Gill, S.P.S. WSN-based smart sensors and actuator for power management in intelligent buildings. *IEEE/ASME Trans. Mechatron.* **2014**, *20*, 564–571. [CrossRef]
18. Liu, L.; Stroulia, E.; Nikolaidis, I.; Miguel-Cruz, A.; Rincon, A.R. Smart homes and home health monitoring technologies for older adults: A systematic review. *Int. J. Med. Inform.* **2016**, *91*, 44–59. [CrossRef]
19. Rashidi, P.; Mihailidis, A. A survey on ambient-assisted living tools for older adults. *IEEE J. Biomed. Health Inform.* **2012**, *17*, 579–590.
20. Đurišić, M.P.; Tafa, Z.; Dimić, G.; Milutinović, V. A survey of military applications of wireless sensor networks. In Proceedings of the 2012 Mediterranean conference on embedded computing (MECO), Bar, Montenegro, 19–21 June 2012; pp. 196–199.
21. BenSaleh, M.S.; Saida, R.; Kacem, Y.H.; Abid, M. Wireless Sensor Network Design Methodologies: A Survey. *J. Sens.* **2020**, *2020*. [CrossRef]
22. Abrach, H.; Bhatti, S.; Carlson, J.; Dai, H.; Rose, J.; Sheth, A.; Shucker, B.; Deng, J.; Han, R. MANTIS: System support for multimodal networks of in-situ sensors. In Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications, San Diego, CA, USA, 19 September 2003; ACM: New York, NY, USA, 2003; pp. 50–59.
23. Watteyne, T.; Vilajosana, X.; Kerkez, B.; Chraim, F.; Weekly, K.; Wang, Q.; Glaser, S.; Pister, K. OpenWSN: A standards-based low-power wireless development environment. *Trans. Emerg. Telecommun. Technol.* **2012**, *23*, 480–493. [CrossRef]
24. Han, C.C.; Kumar, R.; Shea, R.; Kohler, E.; Srivastava, M. A dynamic operating system for sensor nodes. In Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services, Seattle, WA, USA, 6–8 June 2005; pp. 163–176.

25. Cao, Q.; Abdelzaher, T.; Stankovic, J.; He, T. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In Proceedings of the 2008 International Conference on Information Processing in Sensor Networks (ipsn 2008), St. Louis, MO, USA, 22–24 April 2008; pp. 233–244.
26. Baccelli, E.; Hahm, O.; Günes, M.; Wählisch, M.; Schmidt, T.C. RIOT OS: Towards an OS for the Internet of Things. In Proceedings of the 2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Turin, Italy, 14–19 April 2013; pp. 79–80.
27. Labrosse, J.J.; Torres, F. *uC/OS-III: The Real-Time Kernel and the NXP LPC1700*; Micrium Press: Weston, FL, USA, 2010.
28. Levis, P.; Culler, D. Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Not.* **2002**, *37*, 85–95. [CrossRef]
29. Fok, C.L.; Roman, G.C.; Lu, C. Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks. *ACM Trans. Auton. Adapt. Syst.* **2009**, *4*. [CrossRef]
30. Dedecker, J.; Van Cutsem, T.; Mostinckx, S.; D'Hondt, T.; De Meuter, W. Ambient-oriented programming. In Proceedings of the Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, San Diego, CA, USA, 16–20 October 2005; ACM: New York, NY, USA, 2005; pp. 31–40.
31. Whitehouse, K.; Sharp, C.; Brewer, E.; Culler, D. Hood: A neighborhood abstraction for sensor networks. In Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services, Boston, MA, USA, 6–9 June 2004; pp. 99–110.
32. Welsh, M.; Mainland, G. *Programming Sensor Networks Using Abstract Regions*; NSDI: Bostion, MA, USA, 2004; Volume 4, p. 3.
33. Abdelzaher, T.; Blum, B.; Cao, Q.; Chen, Y.; Evans, D.; George, J.; George, S.; Gu, L.; He, T.; Krishnamurthy, S.; et al. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In Proceedings of the 24th International Conference on Distributed Computing Systems, Tokyo, Japan, 26 March 2004; pp. 582–589.
34. Wu, Y.; Rowe, A. Logic-based programming for wireless sensor-activator networks. In Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems, Chicago, IL, USA, 12–14 April 2011; pp. 163–173.
35. Patkar, K.; Prasad, V. Software Framework for Wireless Sensor Network Virtualization. In Proceedings of the 2019 International Conference on Smart Systems and Inventive Technology (ICSSIT), Tirunelveli, India, 27–29 November 2019; pp. 1136–1143.
36. Khalid, Z.; Khalid, U.; Sarijari, M.A.; Safdar, H.; Ullah, R.; Qureshi, M.; Rehman, S.U. Sensor virtualization Middleware design for Ambient Assisted Living based on the Priority packet processing. *Procedia Comput. Sci.* **2019**, *151*, 345–352. [CrossRef]
37. Kifouche, A.; Hamouche, R.; Kocik, R.; Rachedi, A.; Baudoin, G. Model driven framework to enhance sensor network design cycle. *Trans. Emerg. Telecommun. Technol.* **2019**, *30*, e3560.
38. Tei, K.; Shimizu, R.; Fukazawa, Y.; Honiden, S. Model-driven-development-based stepwise software development process for wireless sensor networks. *IEEE Trans. Syst. Man Cybern. Syst.* **2014**, *45*, 675–687. [CrossRef]
39. Doddapaneni, K.; Ever, E.; Gemikonakli, O.; Malavolta, I.; Mostarda, L.; Muccini, H. A model-driven engineering framework for architecting and analysing wireless sensor networks. In Proceedings of the 2012 Third International Workshop on Software Engineering for Sensor Network Applications (SESENA), Zurich, Switzerland, 2 June 2012; pp. 1–7.
40. Salman, A.J.; Al-Yasiri, A. SenNet: A programming toolkit to develop wireless sensor network applications. In Proceedings of the 2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Larnaca, Cyprus, 21–23 November 2016; pp. 1–7.
41. Becker, L.B.; Basso, F.P.; Fröhlich, A.A.; Paulon, A. Model-driven development of WSN applications. In Proceedings of the 2013 III Brazilian Symposium on Computing Systems Engineering, Niteroi, Brazil, 4–8 December 2013; pp. 161–166.
42. Kerasiotis, F.; Koulamas, C.; Papadopoulos, G. Developing wireless sensor network applications based on a function block programming abstraction. In Proceedings of the 2012 IEEE International Conference on Industrial Technology, Athens, Greece, 19–21 March 2012; pp. 372–377.

43. Hakala, I.; Tikkakoski, M. From vertical to horizontal architecture: A cross-layer implementation in a sensor network node. In Proceedings of the First International Conference on Integrated Internet ad Hoc and Sensor Networks, Nice, France, 30–31 May 2006; ACM: New York, NY, USA, 2006; p. 6.
44. Laukkarinen, T.; Kaseva, V.A.; Suhonen, J.; Hamalainen, T.D.; Hannikainen, M. HybridKernel: Preemptive kernel with event-driven extension for resource constrained wireless sensor networks. In Proceedings of the 2009 IEEE Workshop on Signal Processing Systems, Tampere, Finland, 7–9 October 2009; pp. 161–166.
45. Klemets, J.; Määttälä, J.; Hakala, I. Integration of an in-home monitoring system into home care nurses' workflow: A case study. *Int. J. Med Inform.* **2019**, *123*, 29–36. [CrossRef] [PubMed]
46. Jansson, J.; Hakala, I. Managing sensor data streams in a smart home application. *Int. J. Sens. Netw.* **2020**, *32*, 247–258. [CrossRef]
47. LoRa. Available online: https://www.lora-alliance.org (accessed on 17 August 2020).
48. IEEE Standard for Low-Rate Wireless Networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*; IEEE: Toulouse, France, 2016; pp. 1–709.
49. freeRTOS. Available online: https://www.freertos.org (accessed on 24 September 2020).
50. Cheong, E.; Liebman, J.; Liu, J.; Zhao, F. TinyGALS: A programming model for event-driven embedded systems. In Proceedings of the 2003 ACM symposium on Applied computing, Melbourne, FL, USA, 9–12 March 2003; pp. 698–704.
51. Arduino. Available online: https://www.arduino.cc/ (accessed on 17 August 2020).