Behavioural Design of Sensor Network Applications Using Activity-Driven States

Amir Taherkordi, Frank Eliassen, and Einar Broch Johnsen
Department of Informatics
University of Oslo, Norway
{*amirhost,frank,einarj*}*@ifi.uio.no*

***Abstract*—The challenge of designing and programming Wireless Sensor Network (WSN) applications has gained increasing attention in recent years. While most existing programming models for WSNs share the same goal of improving software modularity, there exists a gap between the *structural* software design patterns offered by them and the high-level description of system components. The gap has appeared due to the lack of a software design solution that can model the unique *behavioural* and *dynamic* aspects of WSN software, *e.g.*, activities, states, timed operations, and event-driven control flow. In this paper, we present a behavioural design solution for sensor networks based on the principles of finite automata, abstracting the complicated dynamic aspects of WSN software systems through the concept of *activity-driven states*. This promises a design model which effectively fills the above gap and provides the programmer with concrete design elements that can be directly mapped to the constructs of target programming languages. Moreover, it allows more accurate verification and validation of software systems for WSNs by precisely formulating their behavioural elements.**

***Keywords*-Sensor Networks, Behavioural Design Model, Automata, Activity-driven States**

I. Introduction

WSN technology is being increasingly adopted in a wide variety of applications ranging from building and industrial automation to more safety critical applications such as healthcare and patient monitoring. Whereas the early WSN applications were primarily concerned with sensing primitive environmental data and sending those data to a central node, sensor nodes in today's applications are often multi-functional and tightly interact with other devices in the field in order to realize intricate use-case scenarios. This implies that advanced WSN applications need particular attention to the analysis and design of software prior to implementation. This can significantly improve the process of software development and allow the developer to validate and verify the correctness of software functionality based on proper analysis and design models.

From the software design perspective, a critical need for building WSN applications is the *dynamic design view* of the system which emphasizes the flow of control and sensed data, processes and their interactions, and changes to the state of sensor nodes. The recent programming models for WSNs have attempted to address this concern to some extent, both at the node level and the network level. Beside the careful consideration given to resource limitations in those models, one main goal has been to address the *structural design of sensor software* by introducing higher levels of abstraction, *e.g.*, event-based programming [6], componentization [8], [5], [17], and tuple space [4].

However, the software development process suffers from a gap between the description of dynamic aspects of the system and the aforementioned structural design models. In other words, we need design methodologies that allow the programmer to obtain a concrete *behavioural design view* of WSN applications. Although this concept is not new in the software engineering community, a careful consideration should be given when it comes to the engineering of WSN software. The reason is that, in this case, the behaviour of system is closely intertwined with events, timing, states, concurrency, and error-prone actions. A design method that considers all these concerns in a single modeling approach can provide an unprecedented dynamic view of the application logic which aids programmer understanding of code, error detection, and program verification.

Research on modeling approaches for WSNs has recently received increasing attention. It is mainly focused on abstracting complexities of the application logic using standard modeling techniques, such as Model-Driven Development (MDD) and behavioural UML diagrams. The main focus of research in this area has been on modeling coarse-grained and network-level behaviours (*e.g.*, sampling and aggregation) and optimization [7], [19], [18]. Although this is seen as an important step towards engineering the behavioural aspects of WSN software, further work is required to formulate the behaviour at a more concrete level. Additionally, the traditional state machine theories such as Timed Automata [1], Time Petri Net [20], and StateCharts [10] can serve as a good inspiration to this paper, but they address a limited number of issues and impose formalizations that are not needed for modeling WSNs.

In this paper, we aim to facilitate the design and development of WSN applications via a design method abstracting the *behaviour* of the software based on the concept of *activity-driven states* and the associated modeling framework, called SenseAct. This provides the developer with: *i)* a new design approach to model system processes as a set of timed activities along with conditions to transit between the activities, and *ii)* coarse-grained programming constructs that are obtained from the design model and abstract the

behavioural states in the system. SENSEACT is inspired by the principles of *finite automata* and formulates the new design model based on the concepts of event, timing, activity and state. These are essentially the common attributes of most WSN applications which are inherently event-driven, function in a timely manner, and transit between different states triggered by events.

The rest of paper is organized as follows. In Section II, we demonstrate a typical motivation application for the behavioural design of WSNs. The principles of SENSEACT are presented in Section III. In Section IV, we introduce SENSEACT-based design and development of WSN applications and report our preliminary evaluation result. Related work is presented in Section V and then we conclude this paper and identify some future work in Section VI.

II. MOTIVATION

In this section, we focus on a typical healthcare application and motivate the need for a concrete behavioural design method for WSNs. Healthcare applications introduce strict requirements on end-to-end system reliability and data delivery [11]. In applications that require reliable data delivery, communications are often bidirectional in order to ensure the receipt of data messages by the gateway. This process becomes even more complex when the sensor nodes are configured to listen to the acknowledgement (ACK) messages from the gateway unit in a time window, reducing the energy usage by a periodic listening scheme [2]. In this case, modeling the control flow, scheduling, waiting, and actions in each step is a non-trivial design problem.

We present a simplified scenario of a *patient tracking application* which requires constant and reliable location data propagation by sensor nodes. The more frequent is the sampling of location data, the more accurate is the localization in terms of time. In addition to motion detection, the sensor nodes, in such applications, may be equipped with other types of sensors, *e.g.*, temperature sensor to measure the body temperature of tracked patients. Below, we describe the sequence of actions that implements this simple scenario:

"When a node starts up, it checks the initial settings and default parameters like radio channel, then it sends the first positioning packet. Afterwards, it waits for t_0 milliseconds (ms) before starting the listening window of t_1 ms (for receiving the gateway's ACK). If no ACK data is received, it will create a second listening window of t_2 ms. If this remains in the same situation, the last listening window of t_3 ms is created. If the node eventually receives the ACK, the program proceeds with the next round of packet propagation after T_i ms interval. Otherwise, it immediately repeats the above sequence of actions. In parallel to the above process, the temperature sensor is polled every t_4 ms for detecting temperature changes."

Modeling the behavioural view of this simple scenario needs a design approach that can abstract scheduling, listening, order of activities, and events. It should also abstract program states and their transition plan so that the developer can verify the accuracy of the program control flow. From a bottom-up view, the design method should yield artefacts interpretable to programming constructs that implement behavioural activities, their interactions, and timing constraints. These arise the need for a single modeling approach that can mitigate the complexity of modeling and developing dynamic aspects of WSN applications. To the best of our knowledge, no existing work proposes such an approach.

III. SENSEACT: PRINCIPLES AND BASIC DEFINITIONS

A SENSEACT model is essentially an extension of finite automaton (*i.e.*, a graph containing a finite set of nodes or locations and a finite set of labelled edges) consisting of *activity-driven states* and transitions. An activity-driven state, called ActState, not only indicates the state that a program is running in, but also represents an activity being performed within a state. For example, the data listening ActState is a state in which the program constantly listens to incoming data traffic from a gateway unit. As shown in Figure 1, ActStates can also be labelled with a time value, indicating the duration of activity execution, *e.g.*, the duration of ActState S_2 is set to t_3 ms. Transitions between ActStates take place according to the output of activity execution using *transition functions*. Similar to ActStates, transitions in SENSEACT can be delayed, rather than being performed immediately. Delayed transitions are useful when an ActState is successfully completed and the program must wait for a certain time before moving to the next ActState.

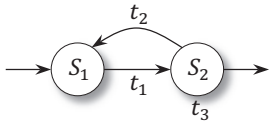

Figure 1: SENSEACT Model

Execution of an ActState may need to retain a set of ActState variables called *state variables*. The scope of a state variable is either *transition-level* or *global-level*. The former includes all values returned by an ActState for the use of next ActState(s), while the latter is the set of variables globally shared by all ActStates.

We formulate the definition of SENSEACT as follows:

Definition 1. A SENSEACT automaton is a 8-tuple $(Q, \Sigma, \delta, V, S, \zeta, E, q_0)$ where

1) Q is the finite set of ActStates
2) Σ is the finite set of ActState execution results
3) $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ is the transition function
4) V is the finite set of state variables
5) $S \subseteq Q \times \mathcal{P}(V) \times Q$ is a set of edges for local state variables transition
6) $\zeta : Q \rightarrow t$ assigns time constraints to states
7) $E \subseteq Q \times t \times Q$ is a set of edges for delayed transitions
8) q_0 is the start ActState

The above formal definition precisely describes what we mean by an activity-based automaton. Obviously, Q denotes the set of all ActStates in a design scenario. Execution of ActState q_i returns a value from Σ. The transition function δ takes as arguments an ActState and an input from Σ and returns the next ActStates which belong to $\mathcal{P}(Q)$, the power set of Q. Local state variable transition is specified by the set S, *e.g.*, $(q_i, \{var1, var2\}, q_j)$ identifies the state variables that should be retained for transition $q_i \rightarrow q_j$. Global state variables are accessible by all states, thereby, we do not need to consider them in transitions. Timing is the key design principle of SENSEACT, shared by both transitions and ActStates. While ζ identifies the time constraint of delayed ActStates, E denotes the set of all delayed transmissions. For example, an edge $(q_i, 20ms, q_j)$ is a transition from state q_i to q_j with 20 ms delay.

Figure 2 depicts a simple SENSEACT, called A. The set of ActStates of A is defined as $Q=\{q_{init}, q_{send}, q_{listen}\}$, where q_{init} is the start ActState and $\Sigma=\{yes, no\}$ is the set of activity execution outputs. The transition function in this example is quite simple, *e.g.*, for $q_{init} \rightarrow q_{send}$ it should ensure that q_{init} has been done successfully (*i.e.*, equal to yes). We define two global state variables ch_no and seq_no to retain the radio channel number and the sequence number of data transmission, respectively. The two time constraints in this example are $(q_{send}, 50\ ms, q_{listen})$ and $\zeta(q_{listen}) \rightarrow 20\ ms$. The former specifies that we need to wait for 50 ms after completing q_{send}, while the latter denotes that ActState q_{listen} should continue listening for 20 ms.

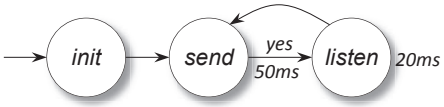

Figure 2: An Example of SENSEACT

Transitions are central to SENSEACT as they basically represent a high-level execution flow of major software components. Therefore, their specifications should be generic enough to meet all execution patterns between activities. Parallel transition is an important aspect of transitions, enabling parallel execution of activities when the function $\delta(Q, \Sigma)$ returns more than one ActState. For example, if the sensor initialization state is performed successfully, the program may require to initiate two different ActStates: listening to the data traffic and polling the temperature sensor regularly. To formulate this, assume that SENSEACT M is in ActState q_m with the possible ActState results of $\Sigma=\{r1, r2\}$. If the execution of ActState q_m returns $r1$, then ActStates $P=\{q_i, p_j, ..., q_n\}$ are executed simultaneously if $(q_m, r1) \rightarrow P$, except for the transition $(q_m, r1) \rightarrow q_k$ with the time constraint of t ms (*i.e.*, delayed transition). Putting the parallel ActStates in a horizontal layout can increase the expressiveness of the model, as shown in Figure 3a. If the execution order is important, parallel transitions are described like $(S_1, y) \rightarrow (\{S_2, S_{22}\}, \trianglerighteq)$. $\{S_2, S_{22}\} \in \mathcal{P}(Q)$ is a totally ordered set with ordering $\trianglerighteq$, which represents the execution order, *e.g.*, $S_2 \trianglerighteq S_{22}$.

The timing feature of SENSEACT is further extended with the concept of *parametric repetition*. Repetition support in SENSEACT is one of its important features as it allows the programmer to abstract operations performed repetitively and maybe with different delay times in each stage of execution, *e.g.*, the sensor node may make three consecutive attempts with different time windows to listen to an incoming data packet. Generally speaking, WSNs are inherently error-prone (*e.g.*, message losses and crashes) and the program execution model is often intertwined with code that might need to re-run a function for fault handling purposes. Therefore, ActState S can be label with the repetitive sequence of executions $\mathcal{T}=\{t_1, t_2, ..., t_n\}$, where t_i denotes the waiting time for the execution of activity S, after the execution at time t_{i-1} (see Figure 3b). Therefore, we complete the description of E in Definition 1 with $\omega : Q \rightarrow \mathcal{T}$, where ω defines the repetitive ActStates and $\mathcal{T}$ denotes the ordered set of time values for each repetition.

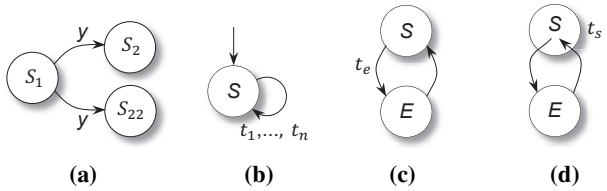

Figure 3: a) Concurrent Transitions, b) Repetition, c) Pull-based Event, and d) Push-based Event

Incorporating events to SENSEACT is of high importance as WSN applications are event-driven by nature and the control flow may change by event occurrence. Two types of events should be considered for modeling: *pull-based* and *push-based* events. While the former is triggered by a piece of code polling the system for events regularly, the latter are events that originate from hardware. As shown in Figure 3c, pull-based events can simply be described by two ActStates: one for event polling in regular interval t_e, and another ActState for event generation. However, push-based events interrupt execution of the running ActState S, transit temporarily to an event processing state E and return to the original ActState S (see Figure 3d). When returning to ActState S, the model either: *i)* resumes the execution of ActState S until t_s is expired, or *ii)* exits from S and transits to another ActState based on the output of ActState E. As a result, push-based ActStates should have mutual transition edges with ActStates that follow the second scenario. To formulate push-based transitions, we extend the notations of Definition 1 with $\delta_e : E \rightleftharpoons \mathcal{P}(Q)$, where E is the finite set for push-based ActStates.

IV. SENSEACT-BASED MODELING OF WSN APPLICATIONS

In this section, we demonstrate SENSEACT's method for design and programming of WSN applications based on the principles presented in the previous section. This method is built up on a design perspective in which each node of the network is in a behavioural state at a given point in time, can stay in the same state for a given period of time, and will transit to another state either immediately or with a certain amount of delay. As shown in Figure 4, the outcomes of SENSEACT are two artefacts: *i)* a SENSEACT-based design model with notations that visualize all design issues discussed in the previous section, and *ii)* a set of programming guidelines and templates that assist the programmer to map the elements of the design model to real programming constructs of the target language.

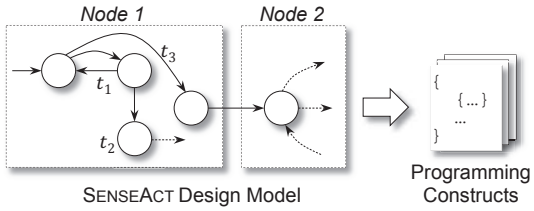

Figure 4: Overall view of SENSEACT's artefacts

The first design step is to identify the behavioural states that the application software can enter over its lifetime, including the transition model between different states. Each ActState represents an operation that takes part in implementing a use case, and at the same time it denotes a state of the application when performing the operation. The transition model is described at the network level in such a way that the end user can gain an overall view on all possible states of the application, as well as their transition map. The obtained ActStates, including the transitions, should be examined with respect to timing, repetitive execution and events according to the principles presented in Section III.

Considering fault modeling is also a significant design issue as faults are likely to occur frequently and unexpectedly in WSNs. SENSEACT models faults as normal ActStates that are responsible to handle the faults with the goal of consolidating similar fault handling functions. In this way, the design model will be enhanced with a set of well-defined fault-handler ActStates that can be reused by other ActStates. It also allows the programmer to verify the fault management flows in different conditions.

Modeling Motivation Application. We revisit here the motivation application scenario and demonstrate the effectiveness of SENSEACT for modeling the activities and flow of control. Note that we focus only on a part of scenario dealing with the propagation of positioning data and temperature sensing. Figure 5 depicts the ActStates and corresponding transitions involved in the scenario, where the right side shows part of model running on the gateway unit. controlling is the central ActState triggering the execution of two core activities: sending the positioning data and polling the temperature sensor. ActState execution can be either successful or failed, thus Σ=$\{yes, no\}$ (For simplicity, we have removed yes labels in the figure). Two types of time constraints should be considered: *i)* on transition: *e.g.*, positioning data should be sent every 3 minutes and data listening should be started 200 ms after sending a packet, and *ii)* on ActState execution: *e.g.*, staying in the data listening ActState for 100 ms. temperature reading also follows the pull-based event pattern, where controlling polls the temperature sensor every one minute.

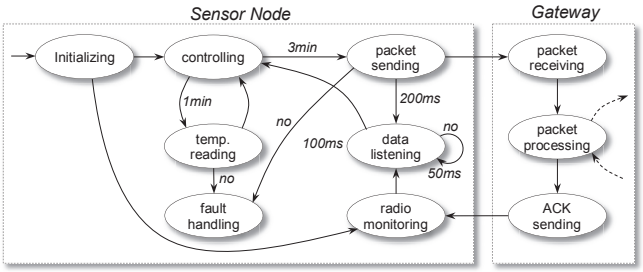

Figure 5: Modeling motivation application with SENSEACT

A. SENSEACT-*Driven Programming and Runtime Support*

As discussed before, one main goal for proposing SENSEACT is to facilitate programming of behavioural aspects of WSN applications. In this section, we discuss how to interpret the elements of a SENSEACT model to a set of programs that can be assembled according to the transitions in the design model. Note that the program templates presented in this section are not tied to any particular programming language, rather the goal is to explore the programming concerns and generic modules.

ActStates are central to the process of transforming SENSEACT models to implementation models. Each SENSEACT entity is considered as a programming construct (*e.g.*, a component or struct) that exposes an interface for life-cycle management (*i.e.*, initiating, executing, and terminating), as well as interfaces for interaction to other ActStates. Figure 6 represents a generic view of an ActState entity along with the required functions. As shown, it may contain local state variables which should be transferred to next ActStates in the sequence. `stayIn(Time duration)` is one of the key interfaces, allowing the programmer to set the time period to stay in a state.

Transitions between ActStates can happen internally and externally. The *internal* transition is introduced for situations where we need to transit in the middle of one ActState to another ActState, *e.g.*, the controlling ActState in Figure 5 requires to trigger sending packet, then proceeds with creating next ActStates in the sequence like radio monitoring. The *external* transition is defined as moving to the next ActState

```
contruct ActState{
  stateVarType stateVar1;
  ...
  stateVarType stateVarN;
  create(params);
  execute(params);
  destroy();

  void stayIn(Time duration);
  ResultType getStateReturnValue();
  ...
}
```

Figure 6: Generic description of ActState

when the execution of current state is completed, *e.g.*, in Figure 5 the transition from initializing to controlling is an external transition.

The other important aspect of SENSEACT-driven programming is the runtime system that hosts ActStates and transitions. Figure 7 illustrates the design model of the runtime support system and its main components. Processing and translating the model (Model Processor), managing the life-cycle of ActStates (ActState Manager), handling the transitions between ActStates and state variables (Transition Handler), and scheduling the timed ActStates and transitions (Generic Scheduler) are the key features of the runtime system. The latter separates all timing concerns from the application code and offers a generic timing service for delaying transitions and ActState execution.

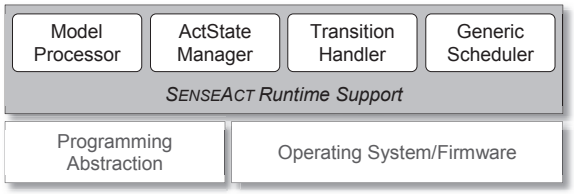

Figure 7: Core components of SENSEACT runtime system

B. Preliminary Evaluation

The evaluation of SENSEACT should be considered from two aspects. The first part is the programming abstraction and the runtime model. This will assess if the overhead of the runtime system is acceptable in terms of resource usage and runtime performance metrics. Since this component of SENSEACT is platform-specific, we may gain different results on different software platforms. For example, if the operating system or firmware exposes a well-defined timer event interface, Generic Schedular will become a lightweight delegator component.

In this paper, we focus only on the second aspect of the evaluation which is the assessment of the model itself. Table I summarizes the comparison between SENSEACT and two set of approaches (see Section V). We include the main evaluation metrics according to the principles of SENSEACT and ignore detailed aspects like parametric repetition. The table shows that MDD approaches pay special attention to code generation, while the formal approaches in the second category address low-level abstraction needs. It indicates that SENSEACT aims to address both, while the code generation is supported to some extent (TSE). By low-level abstraction, we mean that the modeling technique can describe any detailed aspects of system behaviour, which is not the case in high-level ones.

Table I: A comparison between SENSEACT and existing behavioural modeling approaches

Approach	Abstraction Level	Modeling Target	Delayed States	Delayed Transition	Code Generation	Event Modeling
UML2Act.[7]	High	Net.	No	Maybe	Yes	No
MDD [16]	High	Net./Node	No	No	Yes	N/A
MDD [18]	High	Net./Node	No	No	Yes	N/A
MDD [9]	High	Net.	No	Yes	Yes	N/A
T-PetriNets	All	Any	No	Yes	N/A	Yes
T-Automata	All	Any	Yes	No	N/A	No
StateCharts	Low	Object	No	No	N/A	Yes
SENSEACT	Low	Net./Node	Yes	Yes	TSE	Yes

V. RELATED WORK

In this section, we discuss three main categories of approaches abstracting the behaviour of WSN applications: structural design models, MDD approaches, and state-based formalisms.

WSN programming models allow the programmer to obtain a structural design model, visualizing the structure of software modules and their behaviour at a very detailed level. For instance, component-based solutions [8], [5], [17] convert the sensor software to a set of self-contained blocks of functionality that can communicate to each other via interfaces. Abstract Task Graph [15] introduces the notions of Abstract Task and Channel to enable data-centric design of WSN applications. These approaches, in fact, simplify the design of software, but modeling the dynamic behaviours such as activities, delayed interactions, states and flow of control needs a new abstraction atop them.

The second category is related to applying standard software modeling techniques to describe the logic of WSN applications. Some initiatives have been taken to employ behavioural UML diagrams, such as Activity diagram, for visualizing and implementing the software as a set of activities. In [7], UML Activity Diagrams are extended to introduce control structures in the execution flow of software deployed on sensor nodes. Glombitza et al. in [9] propose using State Machines to orchestrate Web services and control flows on sensor nodes. However, the bridge between models and the detailed behavioural aspects of application logic still remains unsolved in the above approaches. Furthermore,

there exist studies that have exploited the concept of MDD to reduce the cost of application development [16]. In [19], a framework is introduced to apply MDD and evolutionary algorithms to select the optimal model of agent-based WSN applications based on non-functional optimization criteria. Inspired by the same framework, in [18] the authors present a model-driven approach, including an automatic model transformation process, for programming data-centric sensor networks. RuleCaster [3] and Flow [14] propose Domain-specific Languages to model WSNs. All these approaches focus on high-level modeling abstractions for complex application logics. We believe that SENSEACT can serve as a complementary model to these approaches with its more concrete abstraction for WSN applications' behaviour.

State machine formalisms are also applied in modeling WSNs. In [13] and [12], techniques are proposed to optimize programming and formulate interaction between TinyOS components respectively, using StateCharts [10]. However, state diagrams in these approaches model the different states of a single object in the system and do not address timing constraints. Timed Automata [1] define the concept of time as a set of real-valued clocks which measures the passage of time and can be reset. According to the formal definition of Timed Automata, an edge (q, a, c, r, q') of an automaton E is a transition from state q to q' with action or alphabet a, clock constraint c and clock resets r. This indicates that the timed automaton spends time only in states, not in edges. However, we need the capability to distinguish timing constraints in states from transitions, where in $(q_m, r) \rightarrow q_n$ both q_m and r are delayed (see motivation application). The global real-valued clocks is also different from the relative and local time calculation model (*i.e.*, state-to-state) in this paper.

VI. CONCLUSION AND FUTURE WORK

In this paper, we presented SENSEACT, a modeling approach to fill the gap between the behavioural description of WSN software and the structural design models proposed by typical programming models. SENSEACT is built on the principles of finite automata and introduces new notations and formalisms to allow modeling of the unique behavioural aspects of WSNs, including timed activities, states, events, and delayed transitions. It also proposes an approach for interpreting the behavioural design elements to a set of coarse-grained programming constructs implementing the dynamic behaviour of WSNs, both at the node level and the network level. Implementing the runtime system of SENSEACT on one of popular sensor operating systems and evaluating its efficiency are in the plan for our future work.

REFERENCES

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2), April 1994.

[2] A. Bachir, M. Dohler, T. Watteyne, and K.K. Leung. Mac essentials for wireless sensor networks. *Communications Surveys Tutorials, IEEE*, 12(2), quarter 2010.

[3] U. Bischoff and G. Kortuem. A state-based programming model and system for wireless sensor networks. In *Pervasive Computing and Communications Workshops, PerCom Workshops '07*, pages 261–266, 2007.

[4] Paolo Costa et al. Programming wireless sensor networks with the teenylime middleware. In *Middleware '07: Proc. of the ACM/IFIP/USENIX Conf. on Middleware*, Newport Beach, California, 2007. Springer-Verlag New York, Inc.

[5] Geoff Coulson et al. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1), 2008.

[6] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proc. of the 4th Conf. on Embedded sensor systems*, Boulder, Colorado, USA, 2006. ACM.

[7] Gerhard Fuchs and Reinhard German. Uml2 activity diagram based programming of wireless sensor networks. In *Proc. of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, SESENA '10, NY, USA, 2010.

[8] David Gay et al. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proc. of the ACM SIGPLAN Conf. on Programming language design and implementation*, San Diego, California, USA, 2003. ACM.

[9] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. Using state machines for a model driven development of web service-based sensor network applications. In *Proc. of the Workshop on Software Eng. for Sensor Network Applications*, SESENA '10, NY, USA, 2010.

[10] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3), 1987.

[11] JeongGil Ko et al. Wireless sensor networks for healthcare. *Proceedings of the IEEE*, 98(11):1947 –1960, nov. 2010.

[12] Volker Menrad and Miguel Garcia. Improving tinyos developer productivity with statecharts, 2009.

[13] M. Mura and M.G. Sami. Code generation from statecharts: Simulation of wireless sensor networks. In *11th EUROMICRO Conf. on Digital System Design Architectures, Methods and Tools (DSD)*, sept. 2008.

[14] Tomasz Naumowicz, Benjamin Schröter, and Jochen Schiller. Prototyping a software factory for wireless sensor networks. In *Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems*, SenSys '09, NY, USA, 2009. ACM.

[15] Animesh Pathak et al. A compilation framework for macro-programming networked sensors. In *Proc. of the 3rd Conf. on Distributed computing in sensor systems*, DCOSS, 2007.

[16] Ryo Shimizu et al. Model driven development for rapid prototyping and optimization of wireless sensor network applications. In *Proc. of the 2nd Workshop on Software Eng. for Sensor Network Applications*, SESENA '11, 2011.

[17] Amir Taherkordi et al. Programming sensor networks using REMORA component model. In *DCOSS '10: Proc. of the 6th Conf. on Distributed Computing in Sensor Systems*, Santa Barbara, CA, USA, 2010. Springer.

[18] Nguyen Xuan Thang et al. Model driven development for data-centric sensor network applications. In *Proc. of the 9th Conf. on Advances in Mobile Computing and Multimedia*, MoMM '11, New York, NY, USA, 2011. ACM.

[19] Nguyen Xuan Thang and Kurt Geihs. Model-driven development with optimization of non-functional constraints in sensor network. In *Proc. of the Workshop on Software Eng. for Sensor Network Applications*, SESENA '10, 2010.

[20] E. Vicario. Static analysis and dynamic steering of time-dependent systems. *Software Engineering, IEEE Transactions on*, 27(8), aug 2001.