Model-Driven Performance Engineering for Wireless Sensor Networks with Feature Modeling and Event Calculus

Pruet Boonma
Department of Computer Engineering
Chiang Mai University
Chiang Mai, 50200, Thailand
pruet@eng.cmu.ac.th

Junichi Suzuki
Department of Computer Science
University of Massachusetts, Boston
Boston, MA 02125, USA
jxs@cs.umb.edu

ABSTRACT

This paper proposes and evaluates a model-driven performance engineering framework for wireless sensor networks (WSNs). The proposed framework, called Moppet, is designed for application developers to rapidly implement WSN applications and estimate their performance. It leverages the notion of feature modeling so that it allows developers to graphically and intuitively specify features (e.g., functionalities and configuration policies) in their applications. It also validates a set of constraints among features and generates application code. Moppet also uses event calculus in order to estimate a WSN application's performance without generating its code nor running it on simulators and real networks. Currently, it can estimate power consumption and lifetime of each sensor node. Experimental results show that, in a small-scale WSN of 16 iMote nodes, Moppet's average performance estimation error is 8%. In a large-scale simulated WSN of 400 nodes, its average estimation error is 2%. Moppet scales well to the network size with respect to estimation accuracy. Moppet generates lightweight nesC code that can be deployed with TinyOS on resource-limited nodes. The current experimental results show that Moppet is well-applicable to implement biologically-inspired routing protocols such as pheromone-based gradient routing protocols and estimate their performance.

Categories and Subject Descriptors

C.2.1 [**Computer Systems Organization**]: Computer Communication Networks; C.4.2 [**Computer Systems Organization**]: Performance of Systems; D.2.2 [**Software**]: Software Engineering—*Design Tools and Techniques*; I.2.3 [**Computing Methodologies**]: Artificial Intelligence—*Deduction and Theorem Proving*

BADS'11, June 14, 2011, Karlsruhe, Germany.

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

As wireless sensor networks (WSNs) have been growing in their scale and complexity, it is complicated, error-prone and time-consuming to rapidly develop and tune WSN applications. Application development and performance tuning can be expensive even for simple applications. This stems from two major issues.

The first issue is a lack of adequate abstractions in application development and performance tuning. The level of abstraction remains very low in developing and tuning WSN applications. For example, a number of WSN applications are currently implemented in nesC, a dialect of the C language, and deployed on the TinyOS operating system, which provides low-level libraries for fundamental functionalities such as sensor reading, node-to-node communication and signal strength sensing. nesC and TinyOS abstract away hardware-level details; however, they do not aid developers to rapidly develop and tune their applications.

The second issue is a lack of sufficient integration between application development and performance tuning. Performance tuning is a process to examine an application's performance through mathematical models, simulations and/or empirical experiments and customize its design/implementation against given performance requirements such as resource consumption. This process is often labor-intensive because most WSN applications are performance sensitive. Few methods and tools exist to seamlessly integrate application development and performance tuning in order to feedback estimated/measured performance results for customizing WSN application design and implementation.

In order to address these two issues, this paper investigates a new model-driven performance engineering framework for WSN applications. The proposed framework, called Moppet, is intended to improve the productivity of developing WSN applications and engineering their performance. Moppet consists of two components: feature modeler and performance estimator (Figure 1). The Moppet feature modeler (Moppet-FM) is a graphical modeling environment to configure WSN applications. It leverages the notion of feature modeling [7], which is a simple yet powerful method to graphically model an application's *features* (e.g., functionalities and configuration policies) and constraints among them. Moppet-FM allows developers to specify each applica-

tion as a set of features and consistently validate constraints among those features. Upon validation, Moppet-FM transforms specified features to application code (nesC code and deployment descriptions for the TinyDDS middleware[1]

The Moppet performance estimator (Moppet-PE) is designed to approximate a WSN application's performance without generating its code nor running it on simulators and real networks. It leverages event calculus [12] to approximate the power consumption on each node. Event calculus is a first-order logical language to represent and reason about events (or actions) and their effects. For example, it can perform deductive inference, which infers *what is true when*, given a set of events at certain time points (i.e., *what events occur when*) and their effects (i.e., *what events do*). By considering individual functionalities performed in each duty cycle on each node as events and describing the power consumption of those functionalities as the effects of the events, Moppet-PE infers when each node runs out of its battery (i.e., what is true when). It is intended to simplify the trial-and-error process for evaluating a WSN application's performance and customizing its features to satisfy performance requirements (Figure 1).

This paper overviews Moppet and evaluates it through empirical and simulations experiments. Experimental results show that, in a small-scale WSN of 16 iMote nodes, Moppet's average performance estimation error is 8%. In a large-scale simulated WSN of 400 nodes, its average estimation error is 2%. Moppet scales well to the network size with respect to estimation accuracy. Moppet generates lightweight nesC code that can be deployed with TinyOS on resource-limited nodes.

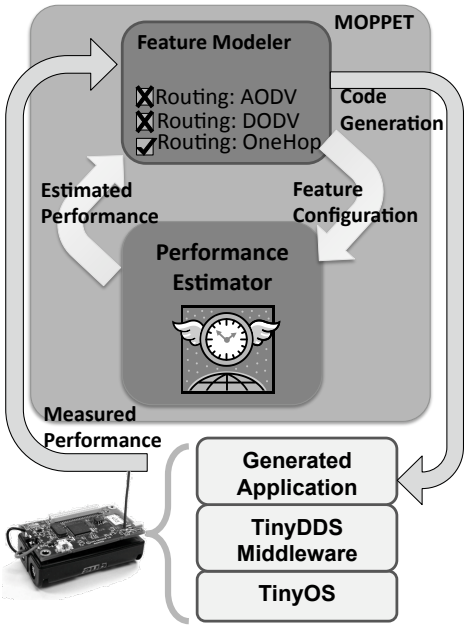

Figure 1: An Overview of Moppet

[1]TinyDDS is a publish/subscribe middleware that is compliant with the standard Data Distribution Service (DDS) specification [17]. See [4, 5] for its design and implementation. It is freely available at http://code.google.com/p/tinydds/.

2. RELATED WORK

Several studies have investigated model-driven development for WSN applications with Unified Modeling Language (UML) [9, 16, 18, 23–25] and domain-specific languages [1, 6, 10,13,15]. Unlike them, Moppet considers performance engineering as well as application development in a model-driven manner. Moreover, it employs feature modeling so that developers (even non-programmers) can build their WSN applications.

Feature modeling has been used to configure the capabilities/parameters in an embedded operating system [14] (e.g., concurrency and interruption policies) and WSN applications [19]. However, these work do not consider performance engineering as Moppet does.

Thang and Geihs intend to augment model-driven development of WSN applications with an evolutionary algorithm [21]. Their approach is to evolve class diagram elements and seek the optimal class diagram with respect to given performance metrics. Performance measures are obtained through simulations. This work is similar to Moppet in that both work generate nesC code from higher-level models and consider performance of WSN applications. However, Moppet is designed to estimate performance without running simulations.

In the area of networks including WSNs, performance engineering often focuses on transforming application designs to program code to be evaluated with simulators [2, 8, 11, 22, 26]. In contrast, Moppet focuses on simulator-less performance estimation without generating program code in order to improve the efficiency of performance estimation.

3. THE MOPPET FEATURE MODELER

Moppet-FM provides a *feature model* that defines a series of features in WSN applications and constraints among the features. The feature model is defined atop the feature metamodel in the Feature Modeling Plug-in (fmp), which is an Eclipse plug-in for editing and configuring feature models[2]. Application developers graphically model a *feature configuration*, which is an instance of Moppet-FM's feature model. It specifies a certain set of features used in a particular WSN application. All modeling artifacts in Moppet-FM are built and maintained with the metameta model (Ecore) in the Eclipse Modeling Framework[3].

Figure 2 shows the feature model in Moppet-FM. It consists of three kinds of features: application-related, network-related and hardware-related features. Application-related features target application functionalities such as application types, data aggregation and data persistence. Network-related features focus on the network layer of applications; e.g., routing. Hardware-related features are used to specify the nodes that applications operate on.

White and black circle icons indicate optional and mandatory features, respectively. For example, `Localization` is mandatory, and `DataAggregation` is optional. Half-filled circle icons represent the features with pre-defined values. A feature can declare its type and have sub-features. When a typed feature is selected in a feature configuration, its value should be specified. In Figure 2, `DataAggregation` has three mandatory sub-features. This means that, if data aggregation is selected in a feature configuration, those sub-features

[2]http://gsd.uwaterloo.ca/fmp/

[3]http://www.eclipse.org/modeling/emf/

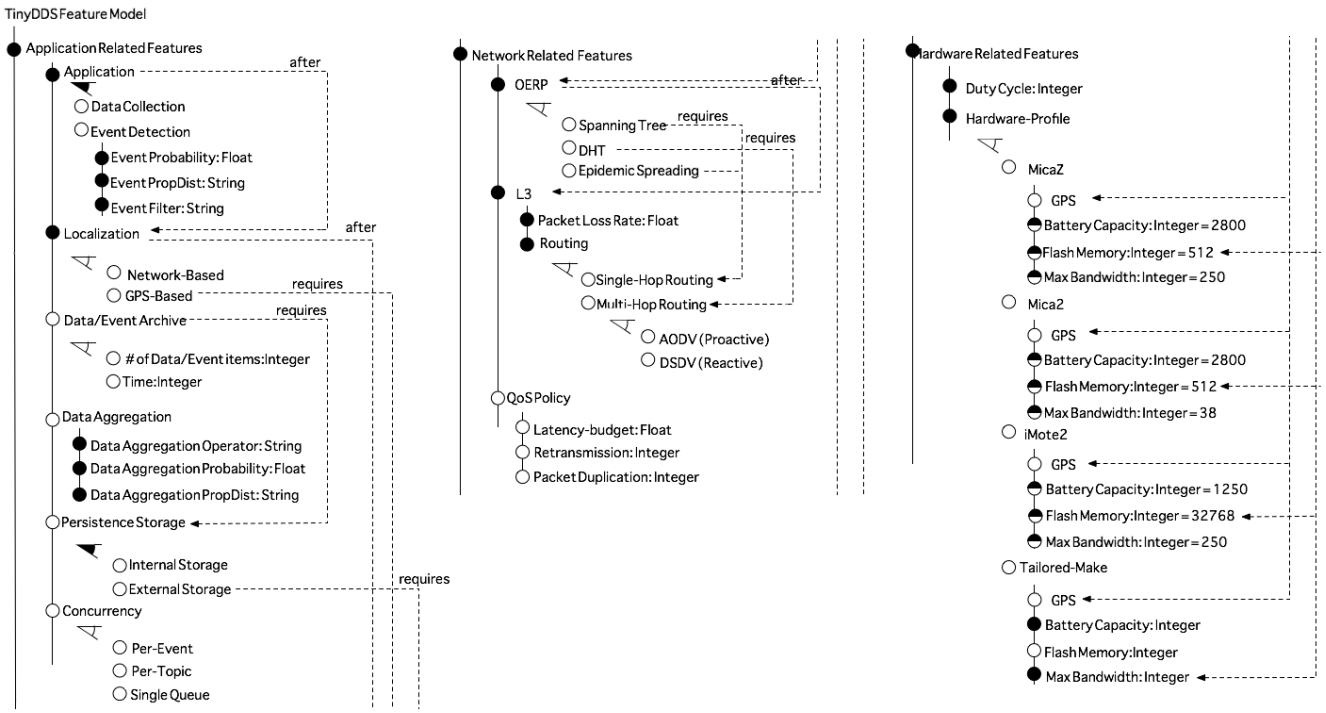

Figure 2: Feature Model in the Moppet Feature Modeler

must be selected as well. One of the sub-features, `DataAggregationOperator`, declares `String` as its type. Thus, if it is selected, its value (e.g., `AVERAGE` or `SUM`) is specified in a feature configuration.

A fork icon with a black sector in a feature model denotes an *inclusive-OR* relationship among sub-features. It requires to select one or more sub-features. In Figure 2, Moppet-FM requires to select `DataCollection` and/or `EventDetection` as the type of an application. A fork icon with a white sector denotes an *exclusive-OR* relationship among sub-features. For example, only one of sub-features is selected at a time for `Localization`.

In addition to the hierarchical structure of features, Moppet-FM uses two types of relationships (dotted lines) among features: *requires* and *after*. A *requires* relationship indicates a dependency among features. For example, when `GPS-based` is selected under `Localization`, `GPS` must be selected as well in a hardware-related feature. An *after* relationship indicates precedence among features in code generation. It instructs which feature's code needs to be generated after which feature's code. In other words, it describes the sequence of features to be performed by generated code. For example, Moppet-FM places the code of `DataCollection` and/or *EventDetection* after the code of `Localization` in code generation.

Figure 3 shows an example feature configuration that is created with Moppet-FM's feature model. As application developers create their feature configurations, Moppet-FM enforces the constraints among features. For example, when `GPS-Based` is selected, Moppet-FM automatically deselects (or deactivates) `Network-Based` and instructs developers to select `GPS` as a required feature. It also warns developers if a property is not defined for `DataAggregationOperator` even when it is selected. Once all constraint violations are resolved, Moppet-FM uses a feature configuration as an input for code generation and performance estimation.

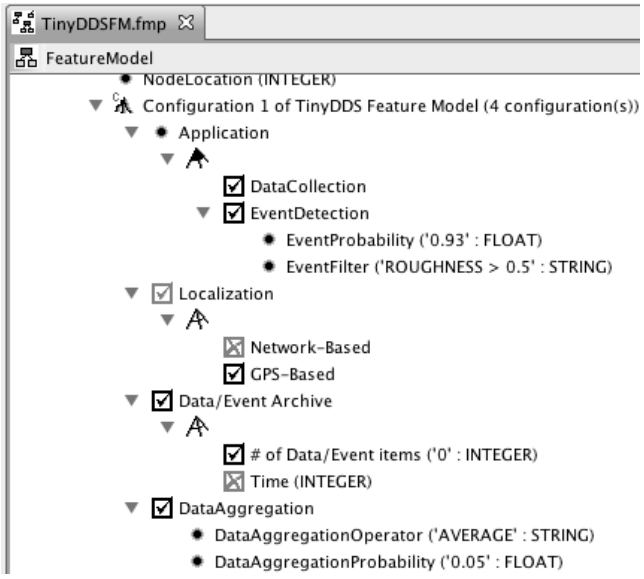

Figure 3: An Example Feature Configuration

4. THE MOPPET PERFORMANCE ESTIMATOR

Moppet-PE receives a feature configuration from Moppet-FM and estimates the lifetime of each node with event calculus. Key components in event calculus are events and fluents. A fluent is a condition (i.e., logical state) that can

Listing 1: An Example Axiom on Power Consumption

```
1  Initiates(Start, Operating, t).
2  Terminates(Stop, Operating, t).
3  Release(Start, BattLevel(level), t).
4  HoldsAt(BattLevel(0), t) → Happens(Stop, t).
5  HoldsAt(BattLevel(level1), t1)  ∧ level2 = level1 - offset →
6          Trajectory(Operating, t1, BattLevel(level2), offset).
7  HoldsAt(BattLevel(level1), t) ∧ HoldsAt(BattLevel(level2), t) → level1 = level2.
8  HoldsAt(BattLevel(100), 0).
9  !HoldsAt(Operating, 0).
10 Happens(Start, 0).
11 ?HoldsAt(Operating, 200).
```

Listing 2: A Snippet of the Axiom for the Initial Parameter Configuration

```
1  !HoldsAt(WakeUp(Mote),0).
2  !HoldsAt(EventDetect(Mote), 0).
3  HoldsAt(Level(Mote,1500000),0).
4  !HoldsAt(EventDetectionApplication(Mote), 0).
5  !HoldsAt(DataOllectionApplication(Mote), 0).
6  HoldsAt(Gps(Mote), 0).
7  Happens(GpsOff(Mote), 6).
8  Happens(Start(Mote),0).
```

change over time. It can appear as an argument of predicates. Basic predicates are shown below.

1. $\tau_1 < \tau_2$: Time point τ_1 is before time point τ_2.
2. `Initiates`(α, β, τ): The fluent β starts to hold (i.e., become true) after the event α occurs at time τ. After τ, β holds until it is altered by any event.
3. `Terminates`(α,β,τ): The fluent β ceases to hold (becomes false) after the event α occurs at time τ. After τ, β does not hold until it is altered by an event.
4. `Happens`(α, τ): The event α occurs at time τ.
5. `HoldsAt`(β, τ): The fluent β holds at time τ.
6. `Releases`(α, β, τ): The fluent β is changeable after the event α occurs at time τ.
7. `Trajectory`(β_1, τ, β_2, δ): If the fluent β_1 is initiated at time τ, the fluent β_2 holds at time $\tau + \delta$

Figure 4 categorizes predicates into three groups. A predicate in the *what happens when* group (`Happens`) describes the relationship between an event and time. The *what actions do* group has the predicates that relate events and fluents. A predicate in the *what's true when* group (`HoldsAt`) describes the relationship between a fluent and time.

Given a set of relationships among events, fluents and time, event calculus can reason about the facts about events, fluents and time. For example, Listing 1 shows an example axiom that can be used to estimate the power consumption of a node. The axiom can conclude whether the node can still operate at the end of a given time period.

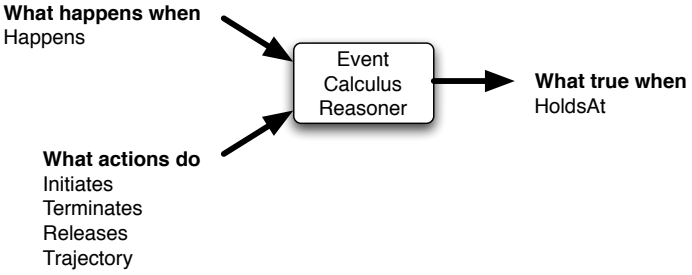

Figure 4: Relationships among Predicates

Lines 1 and 2 describe that, if the `Start` and `Stop` events occur at time t, the node starts and stops to operate, respectively, by changing the fluent `Operating`. Upon the occurrence of `Start` at time t, the fluent on battery level, `BattLevel(level)`), becomes changeable. `BattLevel()` is a function that compares its argument with the current battery level and returns true when they are equal. Line 4 describes that the first predicate (`HoldsAt`) becomes true if `BattLevel(0)` is true at time `t` (i.e., if the current battery level becomes zero at `t`). If this predicate is true, it implies that the second predicate (`Happens`) becomes true as well. As a result, the event `Stop` occurs, and in turn, the fluent `Operating` becomes false due to Line 2.

Line 5 describes a continuous change of battery level over time. If the current battery level is `level1` at time `t1` and the battery level at the next measurement time is `level2`, the fluent `Operating` and an updated battery level of `level2` hold at time `t1+offset`. It is assumed that the battery level decreases by one unit for a unit time period. Line 6 defines

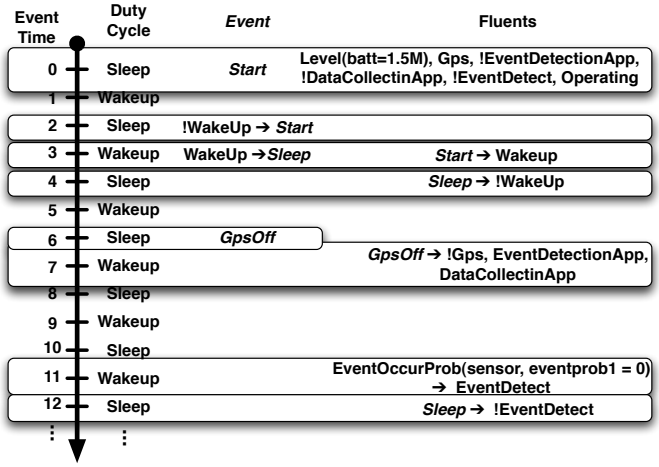

Figure 5: An Example Event Flow

Listing 3: A Snippet of the Axiom for Duty Cycling

```
1  HoldsAt(WakeUp(Mote),t) → Happens(Sleep(Mote),t)
2  Terminates(Sleep(Mote),WakeUp(Mote),t).
3  !HoldsAt(WakeUp(Mote),t)→ Happens(Start(Mote),t).
4  HoldsAt(Operating(Mote), t) → Initiates(Start(Mote),WakeUp(Mote),t).
```

a state constraint which guarantees that only one battery level holds at each time point.

Line 7 defines the initial battery level (100) at the 0th time point. Lines 8 and 9 state that a node in question does not operate and fires the event `Start` at the 0th time point. Line 10 evaluates whether the node can still operate (i.e., whether `small Operating` is true) at the 200th time point.

Figure 5 visualizes the event calculus specification that Moppet-PE uses to estimate the lifetime of each node. It shows how a node changes its state (sleep/off or wakeup/on) over time. It also shows which events occur and which fluents change their states at individual time points. Italic terms indicate events, and non-italic terms indicate fluents.

At the 0th time point, several parameters are initialized according to a feature configuration that Moppet-PE receives from Moppet-FM. For example, the initial battery level is set to 1,500,000 μA/hr. The `GPS` feature is enabled by making the fluent `Gps` hold. The `EventDetectionApplication` and `DataCollectinoApplication` features are disabled by making their corresponding fluents to be false. This is because of an *after* relationship that enforces data collection and event detection after localization. (See Figure 2.) Figure 5 assumes GPS-based localization. Each node's GPS is turned off when `Gps` becomes false at the 6th time point.) Listing 2 shows a snippet of the event calculus axiom that describes the initial parameter configuration.

In Figure 5, the `Wakeup` fluent does not hold at the second time point; a node is sleeping at that time. Accordingly, the `Start` event is emitted. It allows the `Wakeup` fluent to hold at the next time point. Conversely, when the `Wakeup` fluent holds at the third time point, the `Sleep` event is emitted so that `Wakeup` ceases to hold at the next time point. This true-false switching of `Wakeup` controls the duty cycle of a node. Listing 3 shows a snippet of the event calculus axiom that controls each node's duty cycle.

5. EVALUATION

This section evaluates Moppet through simulations and empirical experiments. Simulations are carried out with the TOSSIM simulator. Empirical experiments are conducted with iMote2 nodes. Both simulations and empirical experiments use the the feature configuration shown in Table 1.

This evaluation study considers an oil spill at the sea and simulates a WSN application that detects a spill as an event with sensor nodes, each of which equips a fluorometer. A spill is emulated to contain 110 barrels (approximately 3,100 US gallons) of crude oil and to be spilled in the middle of Dorchester Bay, Massachusetts, in the U.S. The spilled oil moves and spreads out due to the water current and tide. To describe this oil movement, a set of spatio-temporal emulation data is generated with an oil spill trajectory model implemented in the General NOAA Oil Modeling Environment[4]. The emulation data contains timestamped sensor data (i.e., fluorescence reading) at each node. In each simulation and empirical experiment, every node is supplied with this emulation data and emulates its sensor reading in each duty cycle.

This evaluation study uses three different WSNs that consist of 16, 100 200, 300 and 400 nodes deployed uniformly in an observation area. The observation area is 150 meters×150 meters for a WSN of 16 nodes, while 300 meters×300 meters for WSNs of 100 and 400 nodes. An oil spill is emu-

[4]http://response.restoration.noaa.gov/software/gnome/gnome.html

lated to occur at the north-western corner of the observation area. Each node's communication range is 30 meters. A base station is deployed at the center of the observation area. Moppet-PE uses the same power consumption characteristics as the one in TOSSIM [20].

Feature Name	Configuration
Application	Event Detection
- Event Probability	Obtained from GNOME
- Event Prob. Distribution	Gaussian Distribution
- Event Filter	Fluorescence spectrum > 300 nm
Localization	GPS-based
Data/Event Archive	Time: 1 hour
Data Aggregation	
- Data Aggregation Operator	Merge
- Data Aggregation Prob.	0.05
- Data Aggregation Prob. Dist.	Gaussian Distribution
Persistence Storage	External Storage
Concurrency	Per-topic
OERP	Spanning Tree
L3	
- Packet Loss Rate	0.01
- Routing	Single-Hop Routing
QosPolicy	
- Latency-budget	300 seconds
- Retransmission	3
- Packet Duplication	2
Duty Cycle	300 seconds
Hardware-Profile	iMote2
- GPS	Enabled
- Battery Capacity	2800 mA/hour
- Flash Memory	512 KB
- Max Bandwidth	38 kbps

Table 1: A Feature Configuration Used in Simulations and Empirical Experiments

Memory Footprint (KB)	Moppet code	Surge
RAM	7.6	4.9
Flash Memory	120.8	86.9

Table 2: Memory Footprint of Generated Code on an iMote2 Node

Table 2 shows the memory footprint of nesC code that Moppet-FM generates from a feature configuration in Table 1. It consumes approximately 8 KB and 121 KB in RAM and flash memory, respectively. Table 2 also shows the memory footprint of Surge, a simple data collection application bundled in TinyOS. (Serge implements a shortest-path routing protocol based on a spanning tree.) The difference in memory footprint between Moppet-generated code and Surge is 2.7 KB in RAM and 33.9 KB in flash memory. This difference is fairly small, given the fact that Serge does not perform most of the functions of Moppet-generated code (e.g., data aggregation, concurrency and data retransmission). Table 2 demonstrates that Moppet-FM successfully generates lightweight code.

Figures 6 to 10 show the node lifetime that Moppet-PE estimates for the WSNs of 16, 100, 200, 300 and 400 nodes, respectively. Each figure also compares estimated lifetime

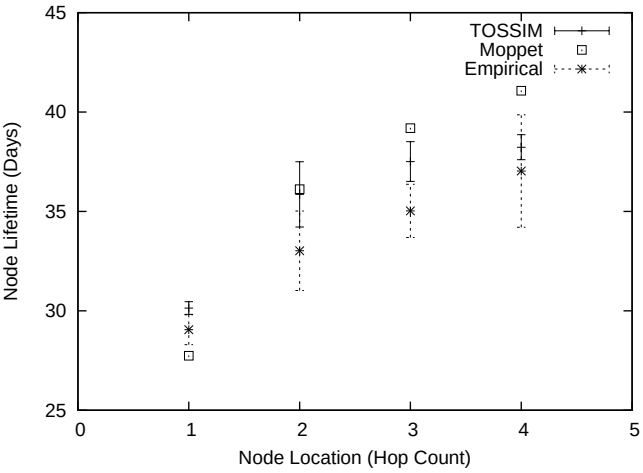

Figure 6: Node Lifetime with 16 Nodes

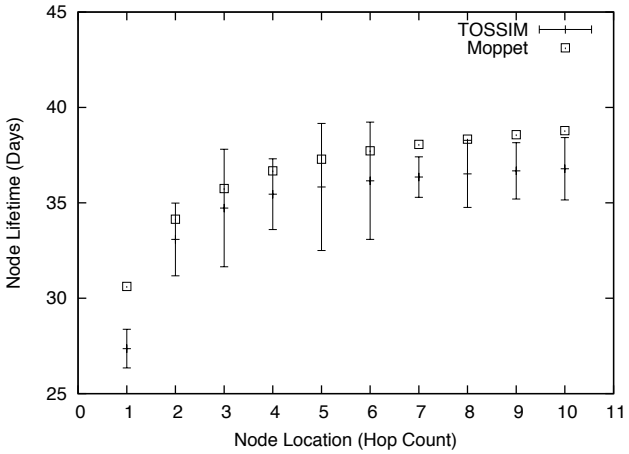

Figure 7: Node Lifetime with 100 Nodes

results with the ones obtained from simulations and empirical experiments. (Empirical experiments are carried out only with 16 nodes.) The X-axis of each figure indicates the location of nodes based on the hop count from the base station. Each of simulated and empirical results shows the average lifetime of nodes with the same hop count from the base station. A range bar denotes a standard deviation.

Figure 6 demonstrates that Moppet's node lifetime estimation is accurate compared with simulated and empirical results. The average estimation error is 5% and 9% against simulated and empirical results, respectively. The maximum estimation error is 8% and 11% against simulated and empirical results, respectively.

Although the number of nodes in the network increases, qualitative trends of estimation error remain similar to the one obtained with the network of 16 nodes. In a WSN of 100 nodes, the average and maximum estimation errors are 5% and 11% (Figure 7). With 400 nodes, the average and maximum estimation errors are 2% and 4% (Figure 10). Figures 6 to 10 illustrate that Moppet-PE scales well to the network size; its node lifetime estimation is accurate and practical in large-scale networks.

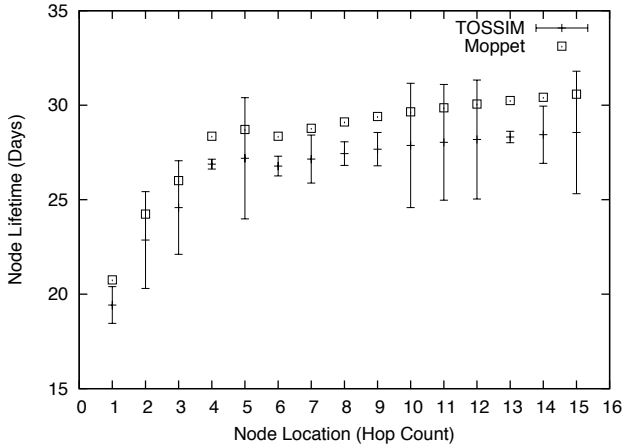

Figure 8: Node Lifetime with 200 Nodes

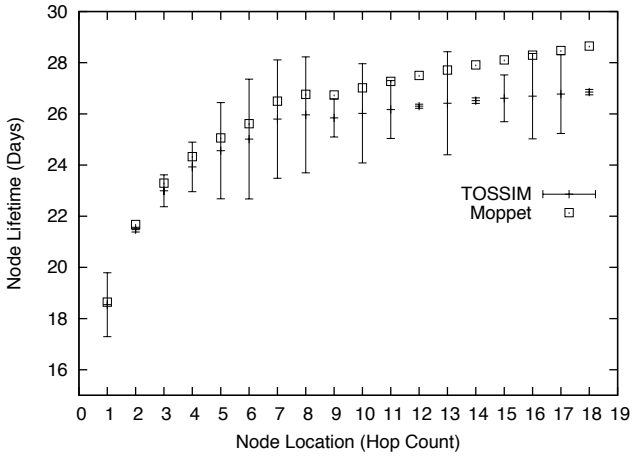

Figure 9: Node Lifetime with 300 Nodes

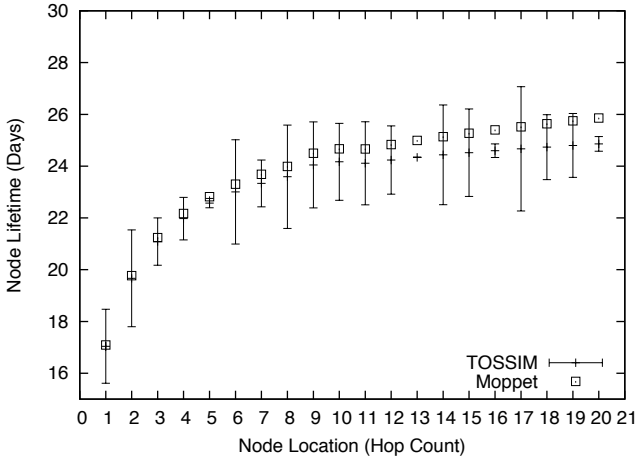

Figure 10: Node Lifetime with 400 Nodes

6. CONCLUSION

This paper proposed and evaluated a model-driven development framework, called Moppet, which aids to rapidly build WSN applications and estimate their performance. Moppet estimates power consumption and node lifetime without generating its code nor running it on simulators and real networks. Its estimation is accurate enough against simulation and empirical experimental results. It scales well to the network size with respect to estimation accuracy. Moppet generates lightweight nesC code that can be deployed with TinyOS on resource-limited nodes.

Several extensions are planned on Moppet. First, Moppet-PE will be extended to support extra performance metrics in its performance estimation, such as data yield, data transmission latency and network lifetime. Secondly, Moppet will be investigated to implement biologically-inspired WSN applications and estimate their performance. Potential applications include the ones using biologically-inspired routing protocols (e.g., [3]).

7. REFERENCES

[1] B. Akbal-Delibas, P. Boonma, and J. Suzuki. Extensible and precise modeling for wireless sensor networks. In *Proc. of the 2nd Int'l Workshop on Model-Based Software and Data Integration*, Sydney, Australia, April 2009.

[2] M. Azgomi and A. Khalili. Performance Evaluation of Sensor Medium Access Control Protocol Using Coloured Petri Nets. *Electronic Notes in Theoretical Computer Science*, 242(2):31–42, 2009.

[3] P. Boonma and J. Suzuki. BiSNET: A biologically-inspired middleware architecture for self-managing wireless sensor networks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 51(16), 2007.

[4] P. Boonma and J. Suzuki. Middleware support for pluggable non-functional properties in wireless sensor networks. In *Proc. of IEEE Int'l Workshop on Methodologies for Non-functional Properties in Services Computing*, July 2008.

[5] P. Boonma and J. Suzuki. TinyDDS: An interoperable and configurable publish/subscribe middleware for wireless sensor networks. In A. Hinze and A. Buchmann, editors, *Principles and Applications of Distributed Event-Based Systems*, chapter 9. IGI Global, 2010.

[6] E. Cheong, E. A. Lee, and Y. Zhao. Joint modeling and design of wireless networks and sensor node software. Technical Report UCB/EECS-2006-150, University of California, Berkeley, November 2006.

[7] K. Czarnecki, U. Eisenecker, and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[8] C. Fok, G. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of IEEE Int'l Conference on Distributed Computing Systems*, June 2005.

[9] G. Fuchs and R. German. UML2 activity diagram based programming of wireless sensor networks. In *Proc. of the 2010 ACM ICSE Workshop on Software Engineering for Sensor Network Applications*, Cape Town, South Africa, May 2010.

[10] N. Glombitza, D. Pfisterer, and S. Fischer. Using state machines for a model driven development of web service-based sensor network applications. In *Proc. of*

the 2010 ACM ICSE Workshop on Software Engineering for Sensor Network Applications, Cape Town, South Africa, May 2010.

[11] Y. Huang, W. Luo, J. Sum, L. Chang, C. Chang, and R. Chen. Lifetime Performance of an energy efficient clustering algorithm for cluster-based wireless sensor networks. In *Proc. of Int'l Symposium on Parallel and distributed Processing and Applications*, August 2007.

[12] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[13] M. Kuorilehto, M. Hännikäinen, and T. D. Hämäläinen. Rapid design and evaluation framework for wireless sensor networks. *Ad Hoc Networks*, 6(6), 2008.

[14] D. Lohmann, F. Scheler, W. S. Preikchat, and O. Spinczyk. PURE Embedded Operating Systems - CiAO. In *Proc. of IEEE Int'l Workshop on Operating System Platforms for Embedded Real-Time Applications*, July 2006.

[15] F. Losilla, C. Vecente-Chicote, B. Alvarez, A. Iborra, and P. Sanchez. Wireless sensor network application development: An architecture-centric MDE approach. In *Proc. of the European Conference on Software Architecture*, September 2007.

[16] M. Mura and M. Sami. Code Generation from Statecharts: Simulation of Wireless Sensor Networks. In *Proc. of EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, September 2008.

[17] Object Management Group. Data Distribution Service (DDS) for real-time systems, v1.2, 2007.

[18] D. A. Sadilek. Prototyping Domain-Specific Languages for Wireless Sensor Networks. In *Proc. of Int'l Workshop on Software Language Engineering*, September 2007.

[19] W. Schröder-Preikschat, R. Kapitza, J. Kleinöder, M. Felser, K. Karmeier, T. H. Labella, and F. Dressler. Robust and efficient software management in sensor networks. In *Proc. of Int'l Workshop on Software for Sensor Networks*, January 2007.

[20] V. Shnayder, M. Hempstead, B. rong Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of ACM Int'l Conference on Embedded Networked Sensor Systems*, November 2004.

[21] N. X. Thang and K. Geihs. Model-driven development with optimization of non-functional constraints in sensor network. In *Proc. of the 2010 ACM ICSE Workshop on Software Engineering for Sensor Network Applications*, Cape Town, South Africa, May 2010.

[22] C. Thompson, J. White, B. Dougherty, and D. C. Schmidt. Optimizing mobile application performance with model-driven engineering. In *Proc. of IFIP Int'l Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, November 2009.

[23] C. Vicente-Chicote, F. Loslla, B. Álvarez, and A. Iborra. Applying MDE to the development of flexible and reusable wireless sensor networks. *Int'l Journal of Cooperative Information Systems*, 16(3/4), 2007.

[24] P. Volgyesi, M. Maroti, S. Dora, E. Osses, , and A. Ledeczi. Software composition and verification for sensor networks. *Journal of Science of Computer Programming*, 56(1–2), 2005.

[25] H. Wada, P. Boonma, J. Suzuki, and K. Oba. Modeling and executing adaptive sensor network applications with the matilda uml virtual machine. In *Proc. of the 11th IASTED International Conference on Software Engineering and Applications*, Cambridge, MA, November 2007.

[26] T. Yang, M. Ikeda, G. De Marco, and L. Barolli. Performance Behavior of AODV, DSR and DSDV Protocols for Different Radio Models in Ad-Hoc Sensor Networks. In *Proc. of IEEE Int'l Conference on Parallel Processing*, September 2007.