ProSe: A Programming Tool for Rapid Prototyping of Sensor Networks*

Mahesh Arumugam[1] and Sandeep S. Kulkarni[2]

[1] Cisco Systems, Inc., San Jose, CA 95134, USA
maarumug@cisco.com
[2] Michigan State University, East Lansing, MI 48823, USA
sandeep@cse.msu.edu

Abstract. We focus on application of abstract network protocols towards prototyping sensor networks. Such abstract programs exist for several applications, e.g., routing, tracking, dissemination, etc. These programs are often specified in terms of event-driven actions where the program responds to actions in the environment or previous actions taken by the program. Hence, they are easy to specify, verify and manipulate. However, they cannot be applied directly in sensor networks as the computation model in sensor networks (write all with collision) differs from that (read/write or shared memory) used in abstract programs. Towards this end, we propose ProSe, a programming tool that enables the designers to (1) specify protocols in simple, abstract models, (2) reuse existing fault-tolerant/self-stabilizing protocols from the literature, and (3) automatically generate and deploy code. ProSe hides the deficiencies of existing programming platforms that require the designers to explicitly deal with buffer management, stack management, and flow control. As a result, we expect that ProSe will enable rapid prototyping and quick deployment of protocols.

Keywords: Programming Tool, Network Protocols, Sensor Networks.

1 Introduction

Sensor networks have become popular due to their applications in border patrolling, critical infrastructure protection, habitat monitoring, structural health monitoring, and hazard detection. Furthermore, due to the development in MEMS technology, tiny low-power sensors can now be manufactured and deployed in large numbers. They have been successfully used in large scale deployments of several applications.

One of the important challenges in deploying sensor network applications is programming. Most of the existing platforms (e.g., nesC/TinyOS [2]) for developing sensor network programs use *event-driven programming model* [3]. As identified in [3–5], while an event-driven programming platform has the potential to simplify concurrency by reducing race conditions and deadlocks, the programmer is responsible for stack management and flow control. For example, in nesC/TinyOS platform, the state of an operation does not persist over the duration of entire operation. As a result, programmers need to manually maintain the stack for the operation. Moreover, the state of the

---

* A preliminary version of this paper appeared as a poster in [1].

operation is shared across several functions. Hence, the designer has to manually control the execution flow. As the program size grows, such manual management becomes complex and is often the source of programming errors.

In addition, typical sensor network platforms require the programmers to manage buffers, contend for access to radio channel, and deal with faults. Hence, as mentioned in [6], programming in nesC/TinyOS platform is "somewhat tricky." In [6], the authors motivate the need for a simpler model that allows one to specify applications in terms of event-driven model (that hides several programming level issues).

To simplify programming sensor networks, several approaches are proposed (e.g., [6–24]). These approaches hide most of the low-level details of the network. (We refer the reader to Section 2 for more details on these approaches.) Existing work on macroprogramming primitives require implementation of such primitives in a target platform (e.g., nesC/TinyOS). Moreover, most of these primitives still require the designer to specify protocols in a target platform (though some intricate details are hidden).

Contributions of the paper. With this motivation, in this paper, we propose *ProSe*, a programming platform for sensor networks that allows designers to concisely specify sensor network protocols. ProSe is based on the theoretical foundation on computational model in sensor networks [25, 26]. In [25, 26], the authors model the computations in sensor networks as a *write all with collision* (WAC) model. In this model, in one atomic step, a sensor can write its own state as well as the state of all its neighbors. However, if two sensors try to update the state of a common neighbor (say, k) simultaneously then, due to collision, the state of k remains unchanged. Thus, this model captures the nature of communication in sensor networks. Moreover, in [25, 26], the authors proposed transformation algorithms that allow the designers to specify programs in abstract models (e.g., read/write model, shared-memory model) and transform them into WAC model (cf. Section 3.1 for a brief introduction to these models).

ProSe enables the designers to (1) specify sensor network protocols and macroprogramming primitives in simple, abstract models, (2) transform the programs into WAC model while preserving properties such as fault-tolerance and self-stabilization [27] of the original programs, and (3) automatically generate and deploy code. An advantage of ProSe is that it will facilitate the designer to use existing algorithms for automating the addition of fault-tolerance to existing programs. Moreover, since abstract models are used to specify protocols, ProSe allows the designer to gain assurance about the programs deployed in the network using tools such as model checkers [28].

Additionally, we observe that the work on distributed computing and traditional networking has focused on problems such as consensus, agreement in the presence of faulty/malicious sensors, reliable broadcast, routing, leader election, synchronization, and tracking. These problems (or variations thereof) also need to be solved in the context of sensor networks, either in the design of sensor network protocols or in the design of macroprogramming primitives. However, existing solutions to these problems are written in abstract models such as read/write model and shared-memory model. Since it is desirable to utilize the vast literature in this area, to speed up the development and

deployment of sensor networks, ProSe enables the designers to reuse existing algorithms by automatically transforming them into WAC model.

Organization of the paper. In Section 2, we discuss the related work. In Section 3, we provide a detailed discussion on ProSe. Then, in Section 4, we discuss two case studies. Finally, in Section 5, we make the concluding remarks.

2   Background on Programming Platforms for Sensor Networks

Related work that deals with programming platforms include [6–24].

Macroprogramming. In [7], *collaboration groups* are proposed that hides the designer from issues such as communication protocols, event handling, etc. In [8–10], *macroprogramming* primitives that abstract communication, data sharing and gathering operations are proposed. These primitives are exposed in a high-level language. However, these primitives are application-specific (e.g., *abstract regions* for tracking and gathering [8] and *region streams* for aggregation [9]). And, in [11], *semantic services* programming model is proposed where each service provides semantic interpretation of the raw sensor data or data provided by other semantic services. In this model, users only specify the end goal on what semantic data to collect.

In [12], macroprogramming model, called *Kairos*, that hides the details of code-generation and instantiation, data management, and control is proposed. Kairos provides three abstractions; (1) node-level abstraction, (2) one-hop neighbor list abstraction for performing operations on the neighbor list, and (3) remote data access. In [13], programming language called *Pleiades* is proposed that extends C language with constructs for addressing nodes in a network and accessing local data from individual nodes. In [14], virtual node abstraction is proposed where the physical nodes in the network emulate the virtual node application (specified by the designer). The emulation is divided among three main components: (1) to elect a region leader in each region of the network, (2) to retrieve the current state of virtual node application, and (3) to keep the virtual node state synchronized with the physical nodes in the region.

While [7–14] are designed for simplifying programming application services such as tracking, aggregation, etc, ProSe is designed to simplify programming both network services (e.g., routing, clustering, leader election, distributed reset, etc) and application services. Furthermore, ProSe hides low-level details such as message collisions, corruption, sensor failures, etc. Moreover, unlike Kairos and Pleiades, ProSe does not require any runtime support. Additionally, ProSe enables reuse of existing algorithms while preserving properties such as self-stabilization of the input program.

Rule-based programming. In [15–18], rule based programming approaches are proposed. These approaches allow designers to specify programs similar to guarded commands format. However, unlike ProSe, approaches proposed in [15, 16] require designers to *explicitly* specify send/receive message actions of the sensors. As a result, the designers have to decide what messages to transmit (e.g., raw data vs. some interpretation of data), when message transmissions are scheduled (e.g., backoff based vs. timeslot based), and when to listen to the medium for new messages (e.g, always on radio vs. schedule based). In [18], a declarative sensor network programming paradigm

called *DSN* is proposed. DSN uses *Snlog*, a high-level specification language based on facts and rules, for specifying programs. DSN provides an easy mechanism for interacting with the lower layers of the stack and components written in systems languages.

The approaches proposed in [15–18] do not facilitate the reuse of abstract protocols from the literature. Moreover, dynamic embedded sensing and actuation language (*DESAL*) proposed in [17] does not provide a mechanism for preserving properties of interest (e.g., fault-tolerance, self-stabilization) in the transformed programs. Additionally, unlike [18], ProSe does not require any runtime support.

Transaction-based programming. In [19], a transactional framework, called *TRANS-ACT*, is proposed for programming wireless sensor/actor networks. In this approach, an execution of a non-local method is of the form: *read*[*write-all*]. In other words, a non-local method consists of: (1) a read operation that reads the state of the neighbors and (2) a write-all operation that updates the state of all the neighbors. This model differs from WAC model as follows. Specifically, read action is modeled in [25] as a write-all action that updates the state of a sensor at all its neighbors. Therefore, expect for the write-all action, each method accesses only local variables. On the other hand, in TRASACT, designers have to specify what variables to read and what variables to update at the neigbhors in every method. As a result, TRASACT introduces unnecessary overheads in the implementation of read and write-all operations (e.g., read request, read reply, write-all, acknowledgment, conflict detection, cancellation, and c ancel acks).

Programming tools. Techniques like virtual machine (e.g., *Maté* [6]), middleware (e.g., *EnviroTrack* [20]), library (e.g., *SNACK* [21], *TASK* [22]), and database (e.g., *TinyDB* [23]) are proposed for simplifying programming sensor network applications. Another interesting approach for development of mobile sensor network applications is *CarTel* [24]. CarTel provides a simple querying infrastructure. However, these solutions are (i) application-specific, and/or (ii) restrict the designer to what is available in the virtual machine, middleware, library, or network. By contrast, ProSe provides a simple abstraction while allowing the designer to specify wide variety of protocols.

3   ProSe: Overview, Architecture, and Features

In this section, we present: (1) the theoretical background of ProSe, (2) the architecture of ProSe, (3) the internals of ProSe, and (4) the features of ProSe that simplify sensor network programming.

3.1   Preliminaries and Theoretical Background

Programs are specified in terms of guarded commands [29]; each guarded command (respectively, action) is of the form:

$$guard \quad \longrightarrow \quad statement,$$

where *guard* is a predicate over program variables, and *statement* updates program variables. An action $g \longrightarrow st$ is *enabled* when g evaluates to true and to execute that

action, st is executed. A *computation* of this program consists of a sequence $s_0, s_1, \ldots,$ where s_{j+1} is obtained from s_j by executing actions in the program $(0 \leq j)$.

A computation model limits the variables that an action can read and write. Program actions are split into a set of processes. Each action is associated with one of the processes. (Note that in this paper, the terms process and sensor are synonymous.)

Shared-memory model. In this model, in one atomic step, a sensor can read its state as well as the state of its neighbors and write its own variables.

Read/Write model. In this model, in one atomic step, a sensor can either (1) read the state of one of its neighbors and update its *private* variables, or (2) write its own state.

Write all with collision (WAC) model. In this model, each sensor consists of write-all actions. In one atomic step, a sensor can update its own state and the state of all its neighbors. However, if two or more sensors simultaneously try to update the state of a sensor, say k, then the state of k remains unchanged. Thus, this model captures the nature of communication in sensor networks (i.e., *local broadcast with collision*).

Transformations for WAC model. Recently, approaches have been proposed for transforming programs into WAC model. They can be classified as: (a) TDMA based deterministic transformation [25] and (b) CSMA based probabilistic transformation [26].

TDMA based deterministic transformation. In [25], Kulkarni and Arumugam proposed algorithms for transforming programs written in read/write model into programs in WAC model. In [25], the action by which a process (say, j) reads the state of process k in read/write model is modeled in WAC model by requiring process k to write the appropriate value at process j. However, if another neighbor of j is trying to write the state of j at the same time then, due to collision, none of the write actions succeed. In order to deal with this problem, in [25], time division multiple access (TDMA) is used to ensure that collisions do not occur during write actions. Specifically, in WAC model, each process executes the enabled actions and writes (broadcasts) its state to all its neighbors in its TDMA slots. Note that with TDMA based transformation, the model of computation does not change. Rather, TDMA avoids collisions during execution. However, if the slots are corrupted then collisions may occur during execution.

If the transformation uses a deterministic TDMA service (e.g., [30–32]) to implement the write-all action, the resulting program in WAC model is also deterministic. Additionally, in [25], the authors propose extensions for transforming programs written in shared-memory model into programs in WAC model.

TDMA based transformation algorithms proposed in [25] preserve self-stabilization property of the original programs. A program is self-stabilizing if starting from arbitrary initial states the program (eventually) recovers to states from where the computation proceeds in accordance with its specification. In [25], it has been shown that for every computation of the transformed program in WAC model there is an equivalent computation of the given program. Therefore, if the transformed program transitions into arbitrary states then there is a corresponding transition in the given program. Now, if the given program is self-stabilizing then it will recover to legitimate states. In the transformed program, if the TDMA slots are not corrupted then the transformed program will recover to

legitimate states. Hence, if the TDMA algorithm is self-stabilizing (e.g., [30, 32]) then the transformation preserves self-stabilizing property.

CSMA based probabilistic transformation. In [26], Herman proposed *cached sensor transform* (CST) that allows one to correctly simulate a program written for shared-memory model in sensor networks. CST uses CSMA to broadcast the state of a sensor and, hence, the transformed program is randomized.

Dealing with lossy channels. The WAC model captures the nature of communication in sensor networks. However, in practice, messages may be lost due to various factors including corruption and varying signal-to-noise ratio. We argue that the transformations proposed for WAC model is valid in the presence of lossy channels.

Towards this end, first, we note that a message loss can be treated as a write action of a sensor did not update one or more its neighbors. This is equivalent to the computation of original program, say, in read/write model, where the sensor did not read the corresponding neighbor(s). Therefore, it follows that, for every computation of the transformed program in WAC model, there is an equivalent computation of the original program in read/write model.

In the presence of lossy channels, if a sensor executes the write-all action infinitely often then the state of the sensor is updated at all its neighbors infinitely often. In case of CSMA based transformation, thus, only probabilistic gurantees about the transformed programs can be provided in the presence of lossy channels. Likewise, in case of TDMA based transformation, although collisions are not a concern, the presence of lossy channels enable only probabilistic guarantees about the transformed programs. Additionally, since the transformation preserves the stabilization property of the original program, eventually, the transformed program in WAC model also self-stabilizes to states where each sensor correctly captures the state of all its neighbors.

3.2   Programming Architecture

The programming architecture of ProSe is shown in Figure 1. ProSe transforms the input guarded commands program into a program in WAC model. Subsequently, ProSe generates the corresponding nesC code (targeted for TinyOS). Furthermore, ProSe *wires* the generated code with a MAC layer to implement the write-all action in the WAC model. The MAC layer provides an interface for broadcasting (i.e., writing all neighbors) and receiving WAC messages. ProSe also wires the generated code with *NeighborStateM.nc* that maintains the state of the neighbors of each sensor. The designer can then use the nesC/TinyOS platform to build the binary of the nesC code that can subsequently be disseminated across the network using a network programming service.

Input guarded commands program. In the input program, the designer has to specify whether a variable is *public* or *private*. Also, the designer has to identify the sensor to which the variable belongs. For example, if sensor j accesses its local variable x, it is specified as $x.j$. Consider the *MAX* program (cf. Program 1). Each sensor maintains a public variable x. The goal of MAX is to eventually identify the maximum value of x across the network. Whenever $x.j$ is less than $x.k$, j copies $x.k$ to $x.j$. This allows j to update $x.j$ and, eventually, $x.j$ holds the maximum value of x.

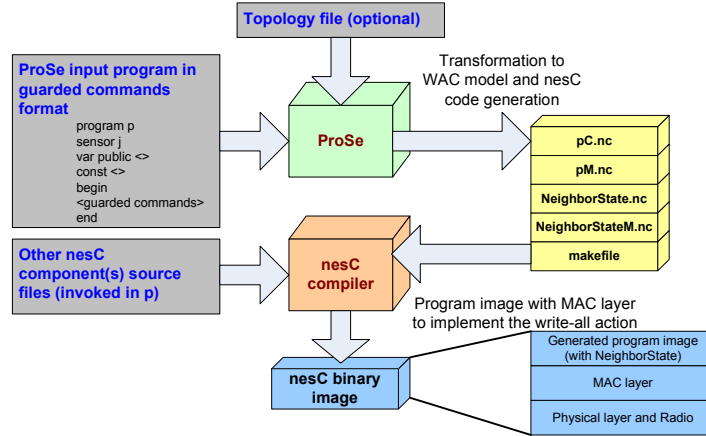**Fig. 1.** Programming architecture of ProSe

---

Program 1. MAX program in shared-memory model

```
1 program max
2 sensor j;
3 var public int x.j;
4 begin
5    (x.k > x.j) -> x.j = x.k;
6 end
7 init state x.j = j;
```

Initial states. The designer also specifies zero or more initial states in the program. If initial states are not specified then the variables are initialized to arbitrary values. And, if the program contains more than one initial state then the variables are initialized to a randomly selected state. In the MAX program example, $x.j$ is initialized to j.

Topology information. ProSe *wires* a component (*NeighborStateM*) that maintains the state of the neighbors at each sensor, with the generated code. Each sensor should identify its neighborhood either dynamically using a neighborhood abstraction layer (e.g., [33]) or statically using a *topology file* which specifies the communication topology [34]. Then, ProSe configures the MAC layer and NeighborStateM.

3.3   Implementation

In the generated program, each sensor maintains a copy vector for each public variable of its neighbor (in NeighborStateM, the module that implements NeighborState interface to get/set copy vectors). Each copy vector captures the value of the corresponding variable at its neighbors. The size of this vector is determined using the neighborhood information of each sensor.

The actions of the input program are executed whenever a timer fires. Then, it marshals all the public variables as a message *wacMsg* and schedules it for transmission (broadcast). Depending on the transformation algorithm and the MAC layer selected by the user, ProSe configures when the timer fires and how *wacMsg* is transmitted. In case of a TDMA based transformation (e.g., [25]), ProSe configures the timer to fire in every TDMA slot assigned to the sensor and broadcasts *wacMsg* using a TDMA service (e.g., [30–32]). In case of a CSMA based transformation (e.g., [26]), it configures the timer to fire in a random interval whenever the sensor receives a message. And, it uses a CSMA service to broadcast *wacMsg*. Thus, as identified in Figure 1, ProSe generates the following nesC files: a *configuration* file, an *interface* file, and 2 *module* files.

Configuration file. Configurations wire components together, connecting interfaces used by components to interfaces provided by others [2]. ProSe generates *pC.nc*, given the input program *p*. *pC.nc* wires *pM.nc*, NeighborStateM.nc, network services (e.g., TDMA, CSMA, etc), and other interfaces required by the module.

Interface and module files. Modules provide the application code and implement one or more interfaces [2]. ProSe generates *pM.nc* (given the input program *p*) and (2) NeighborStateM.nc as outlined below. (For reasons of space, we refer the reader to [34] for the steps involved in the generation of these files.)

- *Initializing nesC modules.* ProSe generates NeighborState.nc that provides get/set functions for public variables of the program. For each public variable, ProSe generates a copy vector in NeighborStateM (with entries for all neighbors of a sensor). NeighborStateM.nc implements NeighborState.nc. In pM.nc, ProSe generates code to (1) initialize components (e.g., TDMA, CSMA, Timer, NeighborStateM) and (2) start network/middleware services (e.g., TDMA, CSMA, Timer).
- *Implementing the guarded commands.* ProSe generates the nesC code for the actions specified in the input program in *Timer.fired()* event. For each action $g \longrightarrow st$, it generates the corresponding nesC code of the form *if(g){st;}*. And, ProSe generates code for implementing the write-all action.
- *Updating the neighbor state.* ProSe generates code for updating NeighborStateM whenever it receives a message. The values of the public variables of the sender are updated in the corresponding copy vectors (in NeighborStateM).

Once code is generated, the designer can use the nesC/TinyOS platform to build the binary image. This image can then be deployed across the network.

3.4 Additional Features

Dealing with faults in protocol design. The normal operation of a network is affected by (1) failure of sensors, (2) state corruption, and (3) message loss. Regarding failure of sensors, ProSe provides an abstraction which allows a sensor (say, j) to determine whether its neighbor (say, k) is alive or failed. In the input program, sensor j can access the public variable $up.k$; if $up.k$ is *TRUE* (respectively, *FALSE*) then k is alive (respectively, failed). ProSe implements this variable using heartbeat protocol (e.g., [35]). For example, if j fails to receive update messages (i.e., WAC messages)

for a pre-determined time interval from its neighbor k then j declares k as failed. Thus, designers can use this abstract variable to simplify the design of programs. Similarly, ProSe also models Byzantine sensors through abstract variables ($b.k$).

Regarding state corruption, ProSe permits arbitrary initial states. This allows the designer to model systems that are perturbed to an arbitrary state. When used in the context of a self-stabilization preserving transformation (e.g., [25, 26]), this feature enables the design of self-stabilizing protocols. Finally, regarding message loss, ProSe allows the designer to provide probability of transmission on any given link.

Priorities of actions. Consider a routing protocol. The actions in the protocol can be classified as either *heartbeat* or *protocol* actions. Heartbeat actions are responsible for checking the status of the neighbors and protocol actions are responsible for construction/maintenance of the routing structure. These two classes of actions may have different priorities, i.e., the frequency of execution of heartbeat actions may be different from protocol actions. Typically, in a network where failures are common, heartbeat actions have higher priority. To represent such actions, ProSe allows the designer to specify priorities for each action. Priority characterizes the frequency with which an action would be executed. And, priorities are specified along with the guarded commands.

Local component invocations. ProSe makes protocol design highly intuitive and concise. However, it is not always desirable to use guarded commands. For example, consider the design of a routing protocol, where the sensors maintain a spanning tree rooted at the base station. In this program, whenever the parent of a sensor fails, it chooses one of its active neighbors for which the link quality is greater than a certain threshold, as its parent. Towards this end, the sensor has to compute the link quality of each of its neighbors. Specifying this action in guarded commands is difficult. Moreover, nesC/TinyOS components may exist that provide the desired functionality.

To enable reuse of existing nesC/TinyOS components, ProSe allows component invocations in guarded commands. For example. in a routing protocol, the designer may invoke the interface *LinkQuality* (implemented by *LinkEstimatorM*) to compute the link quality. Thus, parent update action in the routing protocol can be specified in guarded commands as shown in Program 2. Note that LinkEstimatorM should be implemented in nesC/TinyOS. This component, however, uses only local data (e.g., using NeighborStateM). ProSe wires this component with the generated code.

Program 2. Illustration of local component invocation

```
1 ...
2 component LinkEstimatorM provides LinkQuality;
3 begin
4 ...
5 | (up.(p.j) == FALSE) && (up.k == TRUE) &&
6     (LinkQuality.quality(k) > THRESHOLD)
7  -> p.j = k; quality.j = LinkQuality.quality(k);
8 ...
9 end
```

4   ProSe: Case Studies

In this section, we present two case studies: (1) a routing tree maintenance program and (2) an intruder-interceptor program. (For reasons of space, we refer the reader to [34, 36] for a case study on prototyping a power management protocol.)

4.1   Routing Tree Maintenance Program (RTMP)

In this section, we specify routing tree program (RTMP) [37] in shared-memory model as shown in Program 3. In this program, sensors are arranged in a logical grid. The program constructs a spanning tree with the base station as the root. The base station is located at $\langle 0,0 \rangle$. Each sensor classifies its neighbors as *high* or *low* neighbors depending on their (logical) distance to the base station. Also, each sensor maintains a variable, called *inversion count*. The inversion count of the base station is 0. If a sensor chooses one of its low neighbors as its parent, then it sets its inversion count to that of its parent. Otherwise, it sets its inversion count to inversion count of its parent + 1. Furthermore, to deal with the problem of cycles, if the inversion count exceeds a certain threshold (*CMAX*), the sensor removes itself from the tree.

Program 3. Routing tree maintenance program (RTMP)

```
1 program RoutingTreeMaintenance
2 sensor j;
3 const int CMAX;
4 var
5   public int inv.j, dist.j;
6   public boolean up.j;
7   private int p.j;
8 begin
9 (dist.k < dist.j) && (up.k == TRUE) && (inv.k < CMAX) && (inv.k < inv.j)
10   -> p.j = k; inv.j = inv.k;
11 | (dist.k < dist.j) && (up.k == TRUE) && (inv.k+1 < CMAX) &&
12   (inv.k+1 < inv.j)
13   -> p.j = k; inv.j = inv.k+1;
14 | (p.j != NULL) && ((up.(p.j) == FALSE) ||   (inv.(p.j) >= CMAX) ||
15   ((dist.(p.j) < dist.j) && (inv.j != inv.(p.j))) ||
16   ((dist.(p.j) > dist.j) && (inv.j != inv.(p.j)+1)))
17   -> p.j = NULL; inv.j = CMAX;
18 | (p.j == NULL) && (inv.j < CMAX)
19   -> inv.j = CMAX;
20 end
```

In this program, each sensor (say, j) maintains three *public* variables: (i) $inv.j$, the inversion count of j, (ii) $dist.j$, the (logical) distance of j to the base station, and (iii) $up.j$, the status variable for j (indicates whether j has failed or not). ProSe provides implementation of $up.j$ using heartbeat protocol, as discussed in Section 3.4. Whenever

Table 1. Memory footprint of the generated RTMP program

	Program ROM (in bytes)	RAM (in bytes)
routingM + NeighborStateM	42	106
SS-TDMA	108	586
other components (Timer, FramerM, LedsC, etc)	15934	404

(a)   (b)
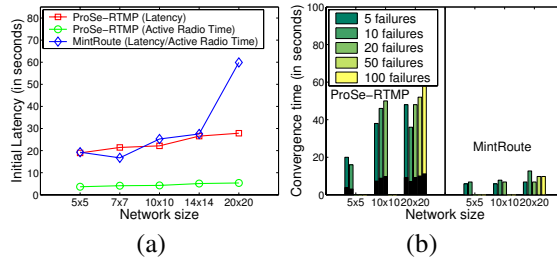
Fig. 2. Simulations results of the generated program. With 90% link reliability: (a) initial latency and (b) convergence time. Note that the black bars in the convergence time graph shows the active radio time during the convergence period. (The result was similar for simulations with 95% link reliability.

j finds a low/high neighbor that provides a better path (in terms of inversion count) to the base station, it updates its *private* variable, $p.j$, the parent of j, and inversion count $inv.j$. Whenever a sensor fails or inversion count is not consistent with its parent, the sensor sets its parent to *NULL* and its inversion count to *CMAX* (i.e., it removes itself from the routing tree). Subsequently, when it finds a neighbor with a better inversion count value, it rejoins the tree.

We used ProSe to transform the program and integrated SS-TDMA [32] with the generated program (cf. Table 1 for memory footprint of the generated program).

Simulation results. We simulated the generated program (ProSe-RTMP) and MintRoute [38] using TOSSIM [39]. (For experimental results, we refer the reader to [34].) In our simulations, the base station is located at $\langle 0,0 \rangle$ (i.e., sensor 0) and the inter-sensor separation is 10 ft. In the absence of any interference, we have observed that probability of successful communication is more than 98% among the neighbors. However, random channel errors can cause the reliability to go down. Hence, we choose conservative estimate of 90% link reliability (that correspond to the analysis in [40, 41]).

In our simulation, each sensor executes the write-all action of the program once in every 2 seconds. And, in MintRoute, the sensors exchange routing information every 2 seconds. Once the initial tree is constructed, we simultaneously fail some sensors and measure the convergence time. The simulation results are shown in Figure 2. The initial latency to construct the routing tree for ProSe-RTMP and MintRoute are similar. MintRoute maintains link estimates of the active links of a sensor and updates the estimate periodically. As a result, the radio is active all the time. By contrast, with ProSe-RTMP, the active radio time of the sensors during this period is significantly less (i.e., around 20% of the initial latency).

Figure 2(b) presents the convergence time of the protocols in the presence of failed sensors. MintRoute converges to a new routing tree quickly. By contrast, ProSe-RTMP converges within 30-50 seconds. We note that this behavior is *not* because of prototyping with ProSe. Rather, it is because of the nature of the original protocol specified with ProSe. More specifically, MintRoute is *pessimistic* in nature, i.e., it maintains a moving average of link estimates of all active links of a sensor all the time. Hence, when sensors fail, it converges to a new tree quickly. By contrast, RTMP is *optimistic* in nature. In other words, whenever a sensor chooses one of its neighbors as its parent, it does not change its parent unless the parent has failed or the tree is corrupted. As a result, when sensors fail, it takes sometime for the protocol to update the tree. On the other hand, the active radio time during recovery is small with ProSe-RTMP.

4.2 Pursuer-Evader Tracking Program

Sensor networks are often used in intruder-interception games, where the sensors guide the pursuer (e.g., a robot, a soldier, etc) to track and intercept the evader (e.g., intruder, hostile vehicle, etc). In this section, we specify the evader-centric program for intruder-interception from [42] in shared-memory model as shown in Program 4. In this program, sensors maintain a tracking structure rooted at the evader. The pursuer follows this tracking structure to intercept the evader. Whenever the pursuer arrives at a sensor (say, k), it consults k to determine its next move. Specifically, it moves to the parent of k. And, since the pursuer is faster than the evader, it eventually intercepts the evader.

Program 4. Pursuer-evader tracking program

```
1 program PursuerEvaderTracking sensor j; var
2   public int dist2Evader.j, detectTimeStamp.j, p.j;
3   private boolean isEvaderHere.j;
4 begin (isEvaderHere.j == TRUE)
5    -> p.j = j; dist2Evader.j = 0; detectTimeStamp.j = TIME;
6 | (detectTimeStamp.k > detectTimeStamp.j) ||
7      ((detectTimeStamp.k == detectTimeStamp.j) &&
8      (dist2Evader.k+1 < dist2Evader.j))
9    -> p.j = k; detectTimeStamp.j = detectTimeStamp.k;
10     dist2Evader.j = dist2Evader.k+1;
11 end
```

Each sensor (say, j) maintains three public variables: (i) *dist2Evader.j*, distance to the root of the tracking structure, (ii) *detectTimeStamp.j*, the timestamp that j knows when the evader was detected at the root, and (iii) $p.j$, the parent of j. Whenever j detects the evader, it sets *detectTimeStamp.j* to its current clock value (using the *TIME* keyword), *dist2Evader.j* to 0 and $p.j$ to itself. If it finds one of its neighbors (say, k) has the latest detection timestamp, then it updates its public variables accordingly and sets its $p.j$ to k. In Program 4, for simplicity, we do not show the actions of the pursuer. Since the pursuer can listen to the messages transmitted by the sensors, whenever the pursuer is near j, it reads the public variable $p.j$ and moves to $p.j$.

Table 2. Memory footprint of the generated tracking program

	Program ROM (in bytes)	RAM (in bytes)
trackingM + NeighborStateM	N/A*	91
SS-TDMA	108	586
other components (Timer, FramerM, LedsC, etc)	14920	404

* The perl script `tinyos-1.x/contrib/SystemC/module_memory_usage` used to obtain
the breakdown of program ROM and RAM used by various components only provides the
RAM usage data for trackingM; it does not report the ROM usage for trackingM. We expect
this value to be small.

(a)   (b)
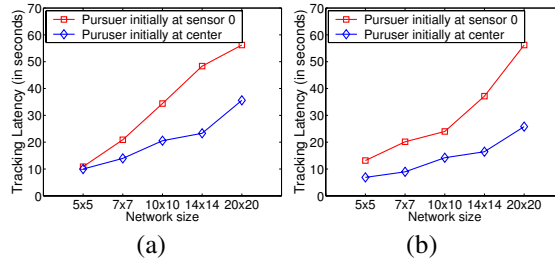
Fig. 3. Tracking latency of the generated program: (a) with 90% link reliability and (b) with 95% link reliability

We used ProSe to transform the program and integrated SS-TDMA [32] with the generated program (cf. Table 2 for memory footprint of the generated program). In this program, we need to *wire* components that detect whether the evader is present near a sensor. For example, if the goal is to intercept vehicles then we need to integrate components that can signal whether a vehicle is present or moving near a sensor (e.g., magnetometer components, accelerometer components). Based on this signal, the variable *isEvaderHere.j* at sensor j is either set or unset. Such components are independent of the design of a tracking service.

Simulation results. We simulated the generated program using TOSSIM [39]. The inter-sensor separation is 10 ft and the TDMA period in SS-TDMA is 0.78 seconds. Similar to Section 4.1, we choose the link reliability to be 90% and 95%. In our simulations, we use a *virtual* pursuer and a *virtual* evader. The evader moves randomly in the network. The variable *isEvaderHere.j* at j is set to *TRUE* or *FALSE* depending on the current location of the evader. The pursuer is twice as fast as the evader. We did two sets of simulations: (1) the initial location of pursuer is at $\langle 0, 0 \rangle$ and (2) initial location of pursuer is at the center of the network. In both scenarios, the initial location of evader is at the corner (i.e., $\langle N-1, N-1 \rangle$ on NxN grid). From Figure 3, we observe that the tracking latency increases as the network size increases. The latency when the pursuer is initially near the center is significantly less than the case where the pursuer is initially at $\langle 0, 0 \rangle$. And, the active radio time is at most 20% of the time required by the pursuer to intercept the evader. Thus, these results demonstrate the potential of ProSe to generate application-level services for sensor networks.

5   Conclusion

We expect that the programs generated by ProSe are competitive to related programs designed manually for sensor networks. Since ProSe hides low-level details from the designer, it allows rapid prototyping of sensor network protocols. Therefore, we expect that the development time of a typical application (composed of several protocols) is small. Furthermore, since ProSe automatically transforms the program in abstract model to generate the corresponding nesC/TinyOS code, it enables quick deployment of applications. We demonstrated this for (1) routing tree maintenance program (cf. Section 4.1), (2) pursuer-evader tracking service (cf. Section 4.2), and (3) power management protocol [36].

We note that program analysis of nesC/TinyOS programs is gaining attention recently as it assists in programmer understanding, error detection, and program validation (e.g., [43]). Specifically, in [43], program analysis is performed on state machines derived from nesC/TinyOS programs. Such analysis is straight-forward in ProSe as the programs are specified in a simple guarded commands format and several analysis tools and model checkers [28] are readily available.

There are several possible future directions to this work. First, we would like to to combine this work with [44] where the sensor network protocols are proposed in a model that is similar to the abstract models used in ProSe. Since [44] focuses on verification aspects of the abstract protocols, combining it with ProSe, will provide assurance guarantees about the deployed programs. Additionally, we are also focusing on integrating ProSe with tools that automatically synthesize fault-tolerant programs from their fault-intolerant versions (e.g., FTSyn [45]).

References

1. Arumugam, M., Kulkarni, S.S.: Prose - a programming tool for rapid prototyping of sensor networks. Poster at the Conference on Sensor, Mesh, and Ad Hoc Communications and Networks, SECON (2007)
2. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Programming Language Design and Implementation (PLDI) (June 2003)
3. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management or, event driven programming is not the opposite of threaded programming. In: USENIX Annual Technical Conference (June 2002)
4. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In: Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys) (November 2006)
5. Kasten, O., Römer, K.: Beyond event handlers: Programming sensor networks with attributed state machines. In: Fourth Internation Conference on Information Processing in Sensor Networks (IPSN) (April 2005)
6. Levis, P., Culler, D.: Maté: A tiny virtual machine for sensor networks. ACM SIGOPS Operating Systems Review 36(5), 85–95 (2002)
7. Liu, J., Chu, M., Liu, J., Reich, J., Zhao, F.: State-centric programming for sensor-actuator network systems. Pervasive Computing 2(4), 50–62 (2003)

8. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: Networked Systems Design and Implementation, NSDI (2004)
9. Newton, R., Welsh, M.: Region streams: Functional macroprogramming for sensor networks. In: Workshop on Data Management for Sensor Networks (2004)
10. Newton, R., Arvind, Welsh, M.: Building up to macroprogramming: An intermediate language for sensor networks. In: International Conference on Information Processing in Sensor Networks (IPSN) (April 2005)
11. Whitehouse, K., Zhao, F., Liu, J.: Semantic streams: A framework for composable semantic interpretation of sensor data. In: Römer, K., Karl, H., Mattern, F. (eds.) EWSN 2006. LNCS, vol. 3868, pp. 5–20. Springer, Heidelberg (2006)
12. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using kairos. In: Prasanna, V.K., Iyengar, S.S., Spirakis, P.G., Welsh, M. (eds.) DCOSS 2005. LNCS, vol. 3560, pp. 126–140. Springer, Heidelberg (2005)
13. Kothari, N., Gummadi, R., Millstein, T., Govindan, R.: Reliable and efficient programming abstractions for wireless sensor networks. In: Programming Language Design and Implementation, PLDI (2007)
14. Brown, M., Gilbert, S., Lynch, N., Newport, C., Nolte, T., Spindel, M.: The virtual node layer: A programming abstraction for wireless sensor networks. In: International Workshop on Wireless Sensor Network Architecture (April 2007)
15. Sen, S., Cardell-Oliver, R.: A rule-based language for programming wireless sensor actuator networks using frequence and communication. In: Workshop on Embedded Networked Sensors (EmNets) (May 2006)
16. Terfloth, K., Wittenburg, G., Schiller, J.: Rule-oriented programming for wireless sensor networks. In: Conference on Distributed Computing in Sensor Networks (2006)
17. Arora, A., Gouda, M., Hallstrom, J., Herman, T., Leal, B., Sridhar, N.: A state-based language for sensor-actuator networks. In: International Workshop on Wireless Sensor Network Architecture (April 2007)
18. Tavakoli, A., Chu, D., Hellerstein, J., Levis, P., Shenker, S.: A declarative sensornet architecture. In: Workshop on Wireless Sensor Network Architecture (2007)
19. Demirbas, M., Soysal, O., Hussain, M.: TRANSACT: A transactional framework for programming wireless sensor/actor networks. In: International Conference on Information Processing in Sensor Networks (April 2008)
20. Abdelzaher, T., et al.: EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. In: International Conference on Distributed Computing Systems (ICDCS) (March 2004)
21. Greenstein, B., Kohler, E., Estrin, D.: A sensor network application construction kit (SNACK). In: Conference on Embedded Networked Sensing Systems (2004)
22. Buonadonna, P., Gay, D., Hellerstein, J., Hong, W., Madden, S.: TASK: Sensor network in a box. In: European Workshop on Wireless Sensor Networks (2005)
23. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. ACM Transactions on Database Systems, TODS (2005)
24. Hull, B., Bychkovsky, V., Chen, K., Goraczko, M., Miu, A., Shih, E., Zhang, Y., Balakrishnan, H., Madden, S.: CarTel: A distributed mobile sensor computing system. In: ACM Conference on Embedded Networked Sensor Systems, SenSys (2006)
25. Kulkarni, S.S., Arumugam, M.: Transformations for write-all-with-collision model. Computer Communications (Elsevier) 29(2), 183–199 (2006)
26. Herman, T.: Models of self-stabilization and sensor networks. In: Workshop on Distributed Computing (2003)
27. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11) (1974)

28. Holzmann, G.J.: The model checker Spin. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
29. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall PTR, Englewood Cliffs (1997)
30. Arumugam, M.: A distributed and deterministic TDMA algorithm for write-all-with-collision model. In: Kulkarni, S., Schiper, A. (eds.) SSS 2008. LNCS, vol. 5340, pp. 4–18. Springer, Heidelberg (2008)
31. Arumugam, M., Kulkarni, S.S.: Self-stabilizing deterministic time division multiple access for sensor networks. AIAA Journal of Aerospace Computing, Information, and Communication (JACIC) 3, 403–419 (2006)
32. Kulkarni, S.S., Arumugam, M.: SS-TDMA: A self-stabilizing mac for sensor networks. In: Sensor Network Operations. Wiley-IEEE Press (2006)
33. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: A neighborhood abstraction for sensor networks. In: ACM International Conference on Mobile Systems, Applications, and Services (MobiSys) (June 2004)
34. Arumugam, M.: Rapid prototyping and quick deployment of sensor networks. PhD thesis, Michigan State University (2006)
35. Gouda, M.G., McGuire, T.M.: Accelerated heartbeat protocols. In: International Confernece on Distributed Computing Systems, ICDCS (1998)
36. Arumugam, M., Wang, L., Kulkarni, S.S.: A case study on prototyping power management protocols for sensor networks. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 50–64. Springer, Heidelberg (2006)
37. Choi, Y.-R., Gouda, M.G., Zhang, H., Arora, A.: Stabilization of grid routing in sensor networks. AIAA Journal of Aerospace Computing, Information, and Communication (JACIC) 3, 214–233 (2006)
38. Woo, A., Culler, D.: Taming the challenges of reliable multihop routing in sensor networks. In: Conference on Embedded Networked Sensing Systems (2003)
39. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: Accurate and scalable simulation of entire tinyOS applications. In: First International Conference on Embedded Netowred Sensor Systems (SenSys), November 2003, pp. 126–137 (2003)
40. Ganesan, D., Krishnamachari, B., Woo, A., Culler, D., Estrin, D., Wicker, S.: An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report IRB-TR-02-003, Intel Research (March 2002)
41. Zuniga, M., Krishnamachari, B.: Analyzing the transitional region in low power wireless links. In: Conference on Sensor and Ad hoc Communications and Networks (2004)
42. Demirbas, M., Arora, A., Gouda, M.: Pursuer-evader tracking in sensor networks. In: Sensor Network Operations. Wiley-IEEE Press (May 2006)
43. Kothari, N., Millstein, T., Govindan, R.: Deriving state machines from TinyOS programs using symbolic execution. In: Seventh Conference on Information Processing in Sensor Networks, IPSN (2008)
44. Gouda, M.G., Choi, Y.-R.: A state-based model for sensor protocols. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 246–260. Springer, Heidelberg (2006)
45. Ebnenasir, A., Kulkarni, S.S., Arora, A.: FTSyn: A framework for automatic synthesis of fault-tolerance. International Journal of Software Tools for Technology Transfer (STTT) 10(5), 455–471 (2008)