FlowTalk: Language Support for Long-Latency Operations in Embedded Devices

Alexandre Bergel, William Harrison, Vinny Cahill, Siobhán Clarke

Abstract—Wireless sensor networks necessitate a programming model different from those used to develop desktop applications. Typically, resources in terms of power and memory are constrained. C is the most common programming language used to develop applications on very small embedded sensor devices. We claim that C does not provide efficient mechanisms to address the implicit asynchronous nature of sensor sampling. C applications for these devices suffer from a *disruption in their control flow*. In this paper, we present FlowTalk, a new object-oriented programming language aimed at making software development for wireless embedded sensor devices easier. FlowTalk is an object-oriented programming language in which dynamicity (*e.g.,* object creation) has been traded for a reduction in memory consumption. The event model that traditionally comes from using sensors is adapted in FlowTalk with *controlled disruption*, a light-weight continuation mechanism. The essence of our model is to turn asynchronous long-latency operations into synchronous and blocking method calls. FlowTalk is built for TinyOS and can be used to develop applications that can fit in 4 kB of memory for a large number of wireless sensor devices.

Index Terms—Embedded systems, object-based programming

1 Introduction

The use of wireless sensor networks has been constantly increasing in various domains (*e.g.,* automotive [28], civil engineering[1], biology [27]). Whereas wireless embedded devices are getting smaller and more sophisticated[2] [41], programming languages used to develop software for small sensor devices have not significantly evolved during the last decade. Most embedded software is written in C [23] (or in one of its derivatives [17]). By being very close to the execution platform, the C language is not appropriate to model complex control flow for an application in the presence of an underlying platform that is event-based.

Although several approaches based on virtual machines like Java Micro Edition[3], SunSPOT[4] or Resilient [1] exist, the target platform must provide enough memory to host a virtual machine. The amount of memory is required to be in the range of 128 to 512 kilo-bytes [38]. This paper focuses on embedded wireless sensor devices, with greater constraints on resources for which the device's memory ranges from 0.5 to 4 kilo-bytes.

These limited resources have an impact on the programming model used to program these devices [33]. One difficulty in sensor programming is the implicit asynchrony of long-latency operations like sensor sampling and radio communications [9]. Sampling the environment (*e.g.,* light, sound or magnetic field) and emitting a radio packet are tasks for which its length in time cannot be predicted [35]. No prediction can be made on how long the sampling or the radio emission will last. Moreover, the central processing unit cannot be interrupted or put into a sleep mode during that time in order to make the device reactive. The traditional approach used in sensor programming is to emit an event, signifying a request for a sampling, and to then wait for notification that will subsequently trigger a callback. This approach has the disadvantage of disrupting the application control flow. A callback, when triggered, does not have the dynamic context in which the event (*i.e.,* request to a sensor) was emitted [39]. Passing data from the main control flow to callbacks necessitates global shared memory allocation, implying the use of guarding locks and preemption needs to be prevented. Although these techniques might be acceptable for a small application, they bring a significant complexity that reduces program source code readability and hampers maintenance.

In this paper, we present FlowTalk, a new programming language aimed at reducing the cost of software development for wireless embedded systems by offering linguistic constructs to model event-based application control flow. FlowTalk is an object-oriented programming language syntactically close to Ruby and Smalltalk in which dynamicity and flexibility has been sacrificed to fit very limited resource constraints. For example, classes cannot be instantiated at run-time. To deal with the asynchrony of sensor sampling FlowTalk uses a technique that we call *controlled disruption*. A computation is cut down into small pieces, which are then inserted into a first-in-first-out (FIFO) queue making them ready to be processed when long-latency operations (*i.e.,* sensor sampling and radio packet emission) are completed. Each of those pieces of computation has the knowledge about the dynamic context of when and how a long-latency operation was triggered. Whereas events are still used by the hardware to request sampling and to be notified when

• *A. Bergel is with the PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Santiago, Chile. W. Harrison is with the Software Structure Group, V. Cahill and S. Clarke are with Lero and the Distributed Systems Group, Trinity College Dublin, Ireland.*
E-mail: abergel@dcc.uchile.cl, Bill.Harrison@scss.tcd.ie, Vinny.Cahill@scss.tcd.ie, Siobhan.Clarke@scss.tcd.ie

1. www.coe.berkeley.edu/labnotes/1101smartbuildings.html
2. robotics.eecs.berkeley.edu/ pister/SmartDust/
3. java.sun.com/javame
4. www.sunspotworld.com

this sampling is done, FlowTalk abstracts this mechanism by making long-latency operations synchronous and blocking.

The contributions of this paper are twofold: (i) it first points out challenges in the computational model offered by wireless sensor networks, (ii) it proposes an approach called *controlled disruption* to deal with the implicit asynchrony of sensors. To ease a software engineering effort when building an application, the benefits of FlowTalk are: (i) applications for embedded sensor devices are built with classes and objects rather than shared global variables and functions, (ii) use of sensors and the radio is driven by method invocations, rather than using signals and callbacks.

The structure of this paper is as follows: Section 2 presents the issues related to long-latency operations and their asynchrony. In Section 3 we describe the FlowTalk programming language, present its properties and show how FlowTalk removes issues resulting from this asynchrony. Section 4 discusses the design choices for FlowTalk and describes its limitations. In Section 5 we briefly present the internals of our compiler. Section 6 gives an evaluation of FlowTalk in terms of memory and battery consumption. In Section 7 we enumerate the principal related works in the field of programming models for embedded devices. Section 8 concludes the paper.

2 Long-Latency Operations and their Asynchrony

A mote[5] is a small, flexible and low-cost embedded device enabling external sampling of the environment (light, magnetic field, sound, ...) and radio communication. The motherboard of a mote contains a micro-controller, a small quantity of memory (anywhere from 0.5 and 4Kb), a radio unit, three colored leds and a collection of sensors. Wireless sensor networks are composed of possibly large numbers of such motes or similar devices. Such a network allows for measurement of environmental values like the light and magnetic field intensity. This data is propagated through the network in order to be processed. Lifetime requirements for such devices scale from months to a year.

This section describes general issues when programming wireless sensor devices. We use a simple application as an illustration (Section 2.1), then show how these issues are reflected in nesC and TinyOS [6], [17], one of the most common programming platforms for embedded sensor networking.

2.1 Disruption in the Application Control Flow

To illustrate the difficulty of programming a wireless sensor device, let us consider a simple behavior that consists of measuring the light and magnetic field intensity, and then sending these values to remote devices. This sensing application is composed of fives steps: (1) sampling the light intensity, (2) sampling the magnetic field intensity, (3) sending the adjusted sum of these two sampled values to a second remote device (operators $<<$ and $>>$ performs bit shift operations), (4) sending the value of the light intensity (obtained from the first step) to a third device, and (5) sending the magnetic field intensity (obtained from the second step) to a fourth device. The control flow of this behavior is depicted in Figure 1. The benefit of summing the two samplings is to emit only one radio packet while the receiver device has to wait for only one package. Numbers on the left-hand side denote the different steps of the computation. This application is intended to be installed on a single wireless sensor device. Programs that run on the three other devices are not discussed.

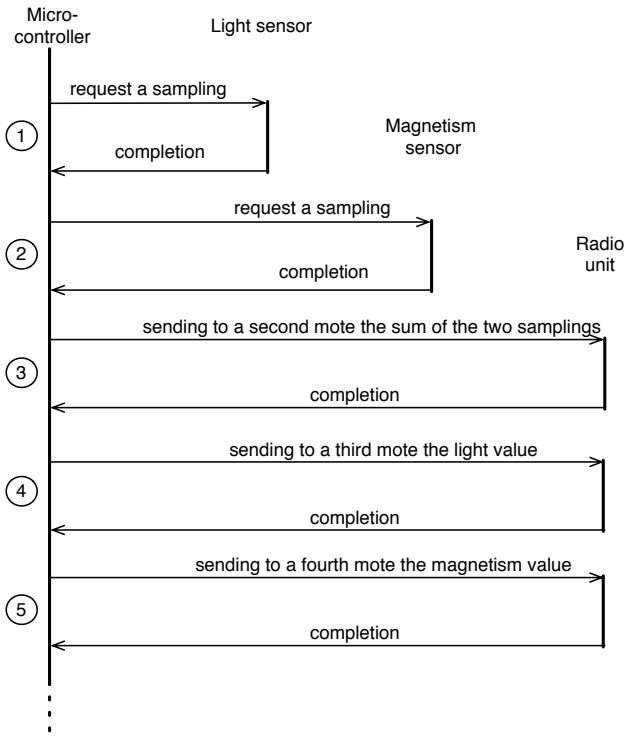

Fig. 1. Control flow of the Sensing application using two sensors and the radio unit.

Operations like sampling a sensor and emitting a radio packet are qualified as *long-latency operations*. Those operations are non-blocking and are realized on the executing platform by sending requests to different electronic component units. Moreover, the time necessary to process a long-latency operation cannot be predicted. The control flow of an application under execution is not suspended or halted while such an operation is processed, leading to an *asynchrony between the application and the sensors*. From this fact, we have identified two main issues:

- *Data-passing.* Passing data between different steps of a computation has to be realized through shared memory. Whereas a stack is used in traditional languages such as Java and C, a heap has to be used to pass values over different steps of the computation since the run-time function stack does not contain any reference to long-latency operation calls.
  In Figure 1, the value resulting from Step 1 is used in Steps 3 and 4. This value has to be stored in the heap, leading to issues stemming from concurrent access as

5. http://www.xbow.com/Products/productdetails.aspx?sid=158

illustrated below.

- *Sequence of operations.* Long-latency operations are non blocking, and the time needed for them to be processed cannot be predicted. Whereas instructions defining program code are naturally ordered in memory in a linear fashion, proper ordering of long-latency operations is not generally supported as illustrated in Section 2.2. Invocations of these operations are driven by the program code, and are therefore ordered, but no assumption can be made about the order of their completion.
  Sometimes, it is perfectly acceptable for long-latency operations to complete in an out-of-order manner. This is illustrated in Figure 1. Although we do not make any assumption on the three remote motes, it is probably not necessary to have Steps 3, 4 and 5 executed sequentially. However, Step 1 must be executed *before* Step 2, itself executed *before* Steps 3, 4 and 5. Usually requesting a sampling of the magnetic field intensity takes a longer time than emitting a radio packet. Although the sampling can be triggered before the radio emission, the latter may still complete before the former.
  A programer should be free to specify sequences of operations so that operation completions reflect the order of their invocation.

Whereas in traditional languages such as Java, C and Smalltalk the application control flow is driven by the runtime stack and instruction sequence, in a wireless sensor device this stack is free from any traces of long-latency operations. We call this situation a *disruption in the control flow* and its consequences are the two issues described above. Figure 1 illustrates a control flow of an application that current event-based programming models for wireless sensor devices do not express in a satisfactory way.

This problem is not singular to wireless sensor network. In fact, most of event-based mechanism suffers from the *inversion of control* issue making the main application control flow moves from places where event handlers are set. This paper innovates on formulating and illustrating the inversion of control issue in wireless sensor network, and provides an adequate solution to it. The inversion of control is mentioned further down in the related work section, when talking about the Scala programming language.

2.2 Illustration with nesC

TinyOS [39] is an open-source operating system designed for wireless embedded sensor networks. Applications for TinyOS are written in nesC, a derivative of C dedicated to motes. This subsection shows how the issues mentioned above in Section 2.1 appear in this popular programming platform for motes, TinyOS. nesC [6] uses an event/callback mechanism to handle long-latency operations. The nesC approach to representing long-latency operations is to make them *split-phase* [6], [17]: operation request and completion are separate functions. In essence, the sensing application, described above, is written in nesC as the following:

```
// Sensing application written in nesC
// It consists of reading two sensors, and sending their
// values to other motes
uint16_t lightIntensity;
uint16_t magneticIntensity;
uint16_t step;

// Step 1
task void senseAndSend() {
  call LightSensor.getData ();
}

async event result_t LightSensor.dataReady (uint16_t value) {
  atomic {lightIntensity = value;}
  post taskForMagnetism();
}

// Step 2
task void taskForMagnetism() {
  call MagneticSensor.getData();
}

async event result_t MagneticSensor.dataReady (uint16_t value) {
  atomic {magneticIntensity = value;}
  post sendToRadio();
}

// Step 3
task void sendToRadio() {
  ...
  atomic {
    msg->val = (lightIntensity>>7) + (magneticIntensity>>7)<<8;
    step = 4;
  }
  // 2 identifies the second mote
  call SendMsgRadio.send (2, ..., msg);
}

// Invoked by the runtime when task sendToRadio has completed
event result_t SendMsgRadio.sendDone(...) {
  ...
  atomic {
    // Step 4
    if (step == 4) {
      msg->val = lightIntensity;
      step = 5;
      // 3 identifies the third mote
      call SendMsgRadio.send (3, ... , msg);
  } else
    // Step 5
    if (step == 5) {
      msg->val = magneticIntensity;
      step = 0;
      // 4 identifies the fourth mote
      call SendMsgRadio.send (4, ... , msg);
    }
  }
  return SUCCESS;
}
```

The nesC code contains 3 global variables. lightIntensity and magneticIntensity are intended to store the light and magnetic intensity, respectively. The step variable is used to indicate the current stage of the computation. Because these variables are global, their access has to be declared as atomic to prevent concurrent calls. This is realized using atomic blocks. The nesC compiler cannot guess that no race condition will occur. nesC checks to see whether variables aren't protected properly and issues warnings when this is the case. The rule for when a variable has to be protected by an atomic linguistic construct is: if the variable is accessed from an async function, then it must be protected. Async functions are functions that can run preemptively, they run asynchronously with regards to tasks.

"Tasks" are provided by nesC to prevent re-entrance and preemption. A task is intended to be posted, making the execution of the code encapsulated by this task deferred. (**Step 1**) The task senseAndSend() is the entry point of the program. It simply requests light intensity sampling. (**Step 2**) When the value is computed by the light sensor, the callback LightSensor.dataReady (uint16_t value) is invoked. The variable value holds the measurement result. The callback stores this value in the global variable lightIntensity, and posts the task taskForMagnetism(). Without entering into the technical details, a callback execution has to be short [17, Section 2.1]. This is the reason why steps consist of callbacks and tasks. The task taskForMagnetism() requests a magnetic field intensity sample. (**Step 3**) Once completed, the callback MagneticSensor.dataReady (uint16_t value) is invoked. It stores the value in the global variable magneticIntensity and emits a radio packet to the mote 2 with the sum of the two samplings. The numeric value 4 is assigned to the global variable step. (**Step 4**) When this radio emission has completed, the callback SendMsgRadio.sendDone (...) is invoked signaling the completion of the radio emission. The step variable is used to request the radio emission to mote 3 with the light intensity value. It also sets the step variable to 5. (**Step 5**) The callback SendMsgRadio.sendDone (...) is again invoked, and finally the last radio emission is performed).

The number of lines of code in the complete application is 136, spread over 3 files.

One of the primary goals of nesC is to provide an efficient mechanism to minimise resource consumption, like memory and power. The main construct is the callback (split-phase). However, this construct makes programming a task more difficult. First, data passing between different steps of the computation is realized by means of global variables. Accesses need to be atomic in order to prevent concurrent executions. Second, the sequence of operations is driven by the callback invocation ordering. Since the same callback is invoked twice, for signaling completion of two operations, a global variable is necessary. As a result, the behavior of the sensing application depicted in Figure 1 is difficult to recognize in the nesC program.

3 FLOWTALK

3.1 FlowTalk in a Nutshell

FlowTalk is a new programming language that makes the programming of wireless sensor embedded devices easier. It abstracts the inherent asynchrony of sensor devices and provides an elegant and simple composition mechanism enabling the reuse of components across different target platforms and embedded software systems.

A program written in FlowTalk is compiled into the native bytecode of the target embedded platform. Programs written in FlowTalk can be run on the Moteiv Telos/TMote and the Crossbow Mica embedded platforms. As described in Section 5, FlowTalk is intended to be used with TinyOS [39]. The executing platforms supported by the FlowTalk compiler are therefore the ones supported by TinyOS. Representing electronic components as objects and the controlled disruption mechanism are the two main characteristics of FlowTalk.

Object-based. Each physical unit on the device (*e.g.,* a sensor or a timer) and each software component (*e.g.,* encryption component) is modeled as an object. Objects are composed together to form executable software. An execution consists therefore of a set of interacting objects. An object is an instance of a class that has its own set of values (*i.e.,* state) and that is able to handle the invocation of methods that are defined in its class (*i.e.,* behavior).

Interaction between a timer, leds and sensors is modeled by message exchanges between objects. Exchanging messages is the only way for objects to interact. Similarly to pure object-oriented languages such as Ruby and Smalltalk, the "everything is an object" paradigm does not prevent one from having global state. Global state in FlowTalk is easily realized by shared a particular object reference to a number of other objects. Variable accessors allow for state access.

Classes can be instantiated only at compile time. Dynamic object creation is not permitted. As explained in Section 4, the rationale behind these strict policies is to comply with limited device resource constraints.

Subclassing enables specialization. New methods can be added in a subclass, and existing methods can be redefined. The original definition of a redefined method is accessible by means of an aliasing mechanism.

Adopting object-orientation unifies the interaction between different software components under a single notion, message passing.

Controlled disruption. To cope with the asynchrony of long-latency operations, the controlled disruption mechanism makes these operations blocking. Method calls triggering long-latency operations are statically detected by the compiler. Each method is cut down into small pieces, each called a fragment. At runtime, method invocation inserts fragments into a global queue leading to a sequential processing order. Each fragment has a reference to its method context containing the value of local variables.

Design goals. FlowTalk offers an approximation of the traditional object paradigm for embedded device. The primary objective of FlowTalk is to provide syntactical constructs to express application control flow in presence of an underlying event-based executing platform. This is achieved by turning long-latency operations into blocking operations. Compared to traditional object paradigm, we have the following four restrictions:

- Method call is the primary mechanism to express control flow. Recursive calls are permitted only if (i) the called method does not perform any long-latency operation, or (ii) if this call is a tail call, *i.e.,* no further operation is performed with the result of the call except than returning it to another caller. A method performing a long-latency operation cannot be active more than once at the same time. The reason for this limitation is that long-latency operations should be realized without consuming stack frames.

- A variable that holds an object on which long-latency operations may be performed is final. This implies that no other value may be assigned to this variable. It is essential to statically locate long-latency operations in a program to cut the program into small pieces of code.
- Objects on which long-latency operation may be directly invoked (*e.g.,* sensor or a radio object) cannot be passed as arguments when calling methods. This restriction is necessary to enable the compiler to statically locate long-latency operations.
- Objects cannot be dynamically created. Dynamic memory allocation is widely recognized as a very expensive feature difficult to cope with wireless sensor network's constraints [37], [42], [43]. Objects are statically created, at composition time.

These restrictions will be explained in a more exhaustive fashion in the remaining of this paper.

3.2 Sensing Application in FlowTalk

The application described in Section 2 is defined in FlowTalk by means of a class SensingApplication, and two methods main and senseAndSend. In FlowTalk, method definitions are syntactically separated from the class declaration. The class SensingApplication is defined as the following:

```
RTObject subclass: #SensingApplication
  variableNames: 'leds timer sensor1 sensor2 radio'
```

SensingApplication is a subclass of RTObject and defines 5 instance variables. RTObject belongs to the runtime provided by FlowTalk and is required to be the top-level class in every class hierarchy composing an application. The main method, the starting point of the application, is defined on this class:

```
SensingApplication main {
  timer invoke: #senseAndSend every: 250.
}
```

Once downloaded into the embedded device, the program will immediately start executing from this method. The main method simply initializes the timer. It sends to the object referenced by the variable timer the message invoke:every:[6] with two arguments, the symbol #senseAndSend and the integer 250. The timer is set up to repeatedly invoke the method senseAndSend every 250 milliseconds.

The method senseAndSend defined on the class SensingApplication defines the behavior described in Figure 1 augmented with the use of leds to indicate the current step of the computation:

```
SensingApplication senseAndSend {
  | v1 v2 |
  leds display: 1.          "Step 1"
  v1 := sensor1 read.

  leds display: 2.          "Step 2"
  v2 := sensor2 read.

  leds display: 3.          "Step 3"
  radio send: ((v1>>7) + ((v2>>7) << 8)) to: #mote2.

  leds display: 4.          "Step 4"
  radio send: v1 to: #mote3.

  leds display: 5.          "Step 5"
  radio send: v2 to: #mote4.
}
```

The method senseAndSend first defines two temporary variables, v1 and v2. They are used to collect the sensor samples. The expression leds display: 1 displays (using a binary format) the value 1 on the leds. The expression v1 := sensor1 read performs a reading on the sensor1 and assigns the result to the first temporary variable, v1. This reading is a blocking operation. The second step is very similar to the first one. The value 2 is displayed and a reading from the second sensor is performed. The third step makes the mote, on which the program is executing, emit a radio packet toward another mote which has the id #mote2. The value transmitted is the sum of the two samples. The fourth step sends the value held by v1 to the mote #mote3, and the last sends the value v2 to #mote4. The three mote identifiers #mote2, #mote3, #mote4 are values used to designate physical mote. Such a value is used when deploying an application on a device.

In order to form an executable, ready to be installed into a mote, the SensingApplication class needs to be instantiated and its variables bound to objects that describe the leds, the timer and the sensors:

```
SensingApplication composeWith:
    {#leds -> Leds .
     #timer -> Timer .
     #sensor1 -> LightSensor .
     #sensor2 -> MagneticSensor .
     #radio -> Radio }
```

The composeWith: directive is interpreted at compile time, before the application deployment therefore. It hooks together the different application components by instantiating the class SensingApplication and "filling" this new instance with instances of the classes Leds, Timer, LightSensor, MagneticSensor and Radio. The practical result of the composeWith: operation is the generation of nesC files. These classes are part of the FlowTalk runtime. Each of them is instantiated, the created object is then assigned to the corresponding instance variable of the object issued from the class SensingApplication. A same instance may be assigned to more than one variable using nested arrows such as: v1 -> v2 -> Component.

The code given above describes the complete Sensing application. Whereas the nesC version has 136 lines of code, the FlowTalk version of this application has 24 lines only. The code FlowTalk is translated into nesC by our compiler. The number of generated lines of code is in the same order of magnitude, with a minimal memory and battery overhead, as explained below (Section 6).

3.3 Controlled Disruption

The previous subsection gave an example of using a blocking long-latency operation with FlowTalk. Accesses to sensors and the radio suspend the control flow of the application. It is resumed later on, when the suspending operation has

6. The reader familiar with the Smalltalk programming language will immediately recognize the similarity with the perform: Smalltalk reflective facility. In FlowTalk however, invoke: accepts only a symbol as immediate value. General expression cannot be provided as a parameter to invoke:.

completed. During the suspension, the micro-controller can execute another part of the code, for example, if a second timer is running.

Making long-latency operations blocking is non-trivial because of very limited resources. Since thread scheduling is a complex mechanism that requires a significant amount of memory, providing threads would come at too high a cost to be supported by the motes. To make long-latency operations blocking, we developed a mechanism named *controlled disruption*.

Long-latency operation identification. At compilation time, long-latency operations are statically identified in the program. As explained below, this is achieved through constraints on variables holding a sensor or a radio. Each method is cut down into small pieces named *fragments*. Each fragment contains a sequence of instructions which ends with a long-latency operation. Methods that do not contain any long-latency operation (for example, performing a computation that does not involve either the sensors or the radio), are assimilated as one single fragment. The sequence of long-latency operations is reflected in the order of fragments composing the method.

At runtime, when a method is invoked, the first fragment of the method is placed in a *fragment queue*. Fragments contained in this queue are sequentially processed. When the first fragment has completed, the following fragment is inserted into the fragment queue. The execution of the method ends when all fragments have been executed. Only one fragment queue is embedded into each mote, it can, therefore, contain references to fragments issued from different methods.

The division of the method senseAndSend into fragments is illustrated in Figure 2. Each fragment ends with a call to a sensor or to a radio. This division is performed at compile time, by means of composeWith:. Each class provided by FlowTalk has the knowledge about long-latency operations that it provides (*cf.*, Section 5). For instance, the class LightSensor declares the method read to trigger a long-latency operation. This is specified by the mapping between a FlowTalk class and a TinyOS component.

Let's assume an application in which three methods may run concurrently:

```
RTObject subclass: #SensingApplication
  variableNames: 'timer1 timer2 timer3 sensor1 sensor2 sensor3 radio'

SensingApplication main {
  timer1 invoke: #senseAndSend every: 250.
  timer2 invoke: #m1 every: 1000.
  timer3 invoke: #m2 every: 2000.
}
```

The three methods senseAndSend, m1, and m2 will therefore run concurrently. The scheduling is determined by the completion of long-latency operations contained in them. Figure 3 gives an example of an execution of these three methods. As illustrated in Figure 2, senseAndSend is composed of 5 fragments (a, b, c, d, and e). Methods m1 and m2 consists of x, y, z and o, p, q, respectively. We focus on the execution of senseAndRead.

(1) Invoking the senseAndRead method inserts the fragment a into the queue.

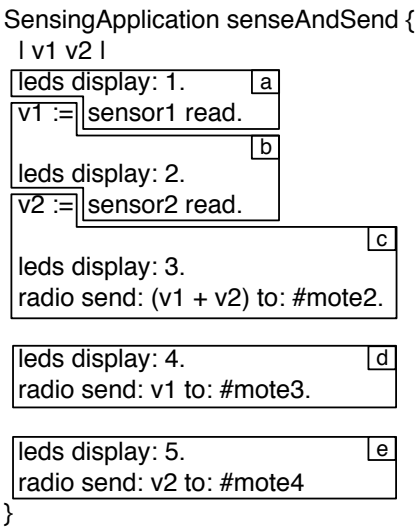

Fig. 2. The method senseAndSend is cut down into small pieces, called fragments (bit shift operations elided for conciseness).

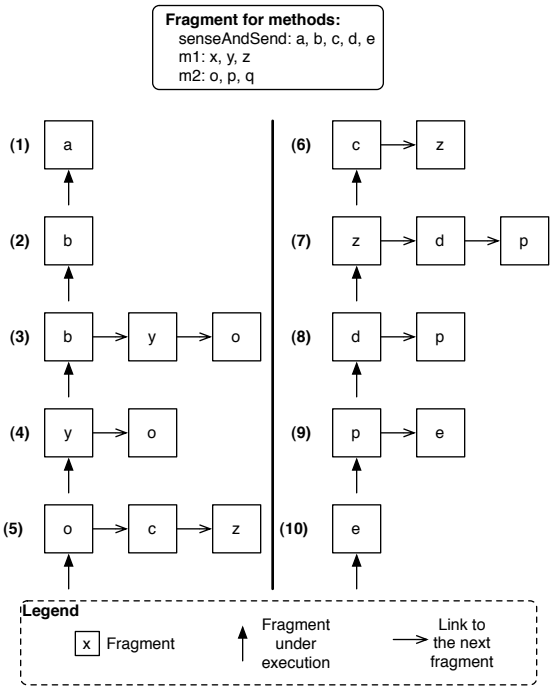

Fig. 3. Example of an execution of the senseAndSend method. Fragments y and z result from other method executions.

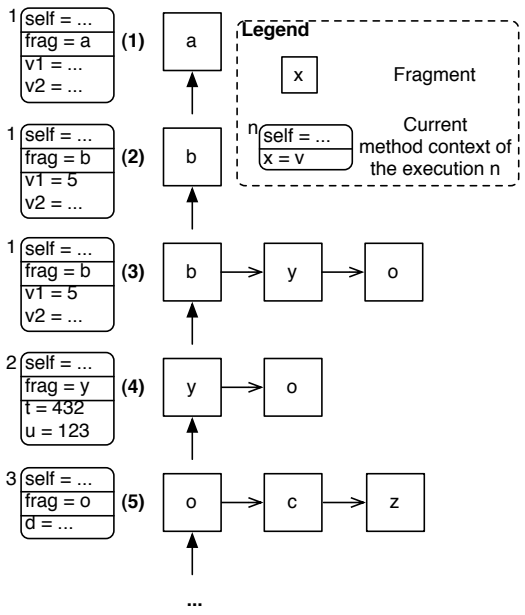

Fig. 4. Two methods are under execution, two contexts are therefore in use.

(2) When this fragment has completed and sensor1 has finished its sampling, it is removed from the queue. And the fragment b is then enqueued.
(3) While b is executing, fragments like y and o issued from m1 and m2 are inserted. These two methods may be executed by some timers.
(4) When b has completed and sensor2 has *not* finished its sampling, it is removed from the queue, and y is being executed.
(5) y has completed, and o is being executed. In the meantime, sensor2 has completed its sampling, which results in an insertion of the c fragment. The fragment z that follows y is also inserted.
(6) The o fragment has completed. c is now under execution.
(7) z is being executed. In the meanwhile, the radio has completed its emission (in fragment c), the fragment d is inserted. p is inserted as well.
(8) d is being executed.
(9) The e fragment is inserted.
(10) e is being executed. Once completed, the method senseAndSend has been executed. Note that the fragment q remains to be processed, however we do not represent it since it is not related to senseAndSend.

Fragments. Methods are cut down into several pieces, called *fragments*. A fragment contains code statements and an environment called a *method context*. A method context is shared among fragments issued from the same method. The code contained in a fragment ends with a long-latency operation, except for the last fragment (*e.g.,* another fragment f might simply use the leds, which is not a long-latency operation). The method context contains a reference to the current receiver, denoted by the self pseudo-variable, the list of values held in temporary variables and the identifier of the current fragment under execution.

When a method is called, the statically allocated method context is initialized with a reference to the self variable and some empty storage location for variables. It is left to the programmer to initialize those variables.

Each fragment ends with a long-latency operation (except for the last fragment of the method). When a fragment under execution has completed, it is removed from the queue and the long-latency operation is performed (*i.e.,* sensor sampling or use of radio). In the meantime, other fragments might be processed. When the long-latency operation has completed, the following fragment is inserted into the queue. The new fragment shares the same method context as the previous fragment.

Figure 4 shows three methods contexts. The first one corresponds to the senseAndSend method, the second one to the m1 method, and the third one to m2. The method context 1 is filled during the execution of fragments a and b at steps (1), (2), and (3). When the long-latency operation associated with fragment a has completed, b is inserted into the queue. The method context is passed from a to b, and the fragment id corresponding to b is set. In (3), new fragments are inserted. In (4), another method is under execution.

Declaring a long-latency operation. Designing a new class to model a sensor may necessitate declaring a method as a long-latency operation. For example, a class TemperatureSensor will need to define a read method and annotate it as a long-latency operation. This is done via a method annotation (< ... >):

```
TemperatureSensor read {
  <LongLatencyOperation> }
```

The annotation LongLatencyOperation declares the read method to be a long-latency operation. This method is therefore blocking and it leads to a new fragment insertion at execution time. Annotating a method as a long-latency operation is static information used by the compiler to identify operations that are inherently asynchronous from the executing platform point of view. As described in Section 5, the read method will have to be mapped to some low-level component.

The LongLatencyOperation annotation is put only on function that are directly long-latency. There is no need to transitively annotate functions that may invoke the long-latency one.

Branching instruction. Use of branching instructions such as a conditional (condition ifTrue: [...] in FlowTalk) needs some special care since the program control flow might not be constant over time. For example, consider the following piece of code:

```
ApplicationClass aMethod {
  | v |
  ((lightSensor read) >> 7) > 2)
    ifTrue: [
      v := soundSensor read.
      radio send: v to: 2.
      radio send: v to: 3].
  radio send: (magnetismSensor read) to: 4.
```

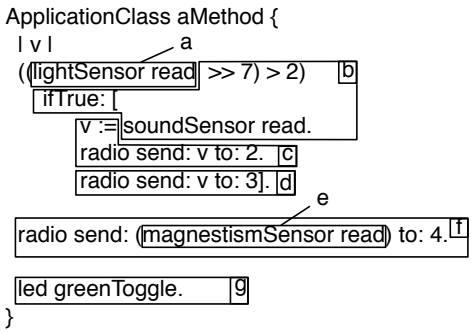

Fig. 5. Controlled disruption applied to a branching instruction.

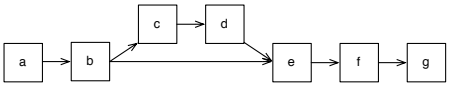

Fig. 6. Control flow of the branching example.

```
  led greenToggle.
}
```

If the first sample of the light intensity is greater than 2, then the intensity obtained from the sound sensor is sent to two motes. After this condition, the magnetic intensity is sampled and sent to a fourth mote. The green led is toggled as the last instruction.

The controlled disruption mechanism divides this method into 7 fragments, as depicted in Figure 5. The execution sequence of those fragments depends on the intensity of the light. Figure 6 illustrates the fact that after the b fragment, the following fragment can be either c or e.

Loops. Iterations in FlowTalk are achieved through recursive method calls. Loops constructs are currently not supported in FlowTalk, but this is simply a matter of syntactic sugar.

Benefits of FlowTalk. In Section 2.1 we identified three problems stemming from the disruption in the sensing application control flow, namely: (i) the difficulty with transmitting data along each step of a computation, (ii) the need to minimise resource consumption, especially regarding memory and power, and (iii) the ability to express sequences of operations. The FlowTalk version of the Sensing application has the following properties:

- Data is passed throughout the execution of a method by using local variables. By being in the same lexical scope, variables are accessible from any part of the method that defines them. Since fragments are sequentially processed and they cannot preempt each other, concurrent local variable access cannot occur.
- A fragment is inserted in the fragment queue only when a long-latency operation has completed. If no other fragment has to be processed while waiting for a fragment's completion, the micro-controller may be put in a sleep mode by the operating system (TinyOS in our case). The overall management of fragments keeps the activity of the micro-controller to a minimum.
- The sequence of operations is reflected in the flow of instructions contained in a method. By making long-latency operations blocking, sensors are always synchronized with the main application.

As a result, the FlowTalk version of the Sensing application reflects the different steps of its control flow (depicted in Figure 1). As we shall see later (Section 6), the fragment management incurs a negligible amount of extra memory and bettery.

The innovation of the language goes well beyond inversion of control that may result from long-latency operations. A programmer will not have to think in terms of asynchronous operation anymore, which bring down the complexity of programming WSNs.

4 Discussion

This section discusses the design choices taken by FlowTalk and their limitations. Beside syntactic differences, FlowTalk differs from languages such as Java on several important points, most especially regarding dynamic allocation in order to cope with limited power and memory resources.

Object creation. Most object-oriented programming languages allow classes to be instantiated at runtime. Objects stored in a heap are under a common memory management policy possibly dictated by a garbage collector. In FlowTalk, objects can be created only at compile time, when different software components are assembled using composeWith:. Dynamic object creation is not permitted. Although statically identifying different objects needed at the execution is a strong constraint, it is widely recognised [37], [42], [43] that dynamic memory management is difficult to cope with while preserving battery power.

Most of the classes in FlowTalk can be freely instantiated. However classes that represent electronic components can be instantiated only once. It would not make sense to instantiate a class RadioEmitter when the execution platform offers only one radio emission unit. This applies to the classes Leds, Radio and for all the different sensors.

Inheritance. In order to reduce the number of entities defined in memory, the class hierarchy is flattened into one single definition. This is enabled by using an alias mechanism for methods. The super pseudo-variable does not exist in FlowTalk. When a subclass is created, methods can be aliased, enabling invocations of the superclass behavior. Creating an alias for a method adds a new name in the class method dictionary. The aliased method can be invoked through its former name and its alias. For instance, refining the class SensingApplication by emitting a sound at each timer invocation is written:

```
SensingApplication subclass: #SensingApplicationWithBeeper
  variableNames: 'beeper'
  alias: {#sense -> #senseAndSend }

SensingApplicationWithBeeper senseAndSend {
  self sense.
```

```
  beeper emitBeep.
}

SensingApplicationWithBeeper
  composeWith: {#leds -> Leds . ... . #beeper -> Beeper }
```

The alias: directive creates an alias sense for the method senseAndSend. At this stage the method senseAndSend can be invoked by sending the message senseAndSend or sense to self. The new definition of the method senseAndSend replaces the previous definition (shown in Figure 2). This new method calls the original definition by means of a message sense sent on self, the current object. The expression beeper emitBeep emits a beep.

This aliasing mechanism of FlowTalk is essentially the same as the alias composition operation in the Traits models [14, Section 4.3]. This formulation of class inheritance allows a class hierarchy to be flattened.

Threads and fragments. One motivation of the controlled disruption mechanism is to make long-latency operations blocking. Section 3.3 mentions thread-support would come at an unacceptably high cost for an embedded device with a very small amount of memory. One might therefore wonder what is the difference between a fragment and a thread. Whereas a thread traditionally comes with an individual stack, a fragment contains a reference to a method context that is shared among fragments issued from the same method invocation. Statements contained in a fragment use the globally unique runtime stack. Fragments cannot be preempted, and their insertion into the fragment queue is driven by the completion of implicitly defined existing fragments.

Controlled disruption as coroutines. Coroutines can be used as the basis for transferring control [10], [12]. With coroutines, a blocking operation is separated into two parts: one before the block and one after. Dynamic information is stored in a context. The pre-block part contains a reference (*i.e.,* function name) to the post-block part. The coroutine is defined by this reference.

Controlled disruption is very similar to coroutines. Long-latency operations are used to delimit fragments, each fragment is uniquely and statically identified and fragment are not preemptively scheduled. However, a number of differences exist.

The state of a coroutine is determined by its current entry and exit points. The dynamic state of a fragment is stored into a method context. This means that the lifespan of a fragment and a method call is dictated by last-in, first-out; in contrast, the lifespan of coroutine is dictated entirely by their use and need.

A fragment does not have an explicit and unique *yield* instruction. It rather uses long-latency operations to delimit its entry and exit points.

Fragment and continuation. A continuation is a procedure that resumes a remaining computation when giving a value. Continuations are constructs that give a programming language the ability to save the execution state at any point to return to that point later on. Intuitively, a continuation represents the "rest of the computation". Fragments and continuation are both mechanisms to suspend and resume a program execution. However, fragments differ from continuation on several points.

Continuations allow you to resume a computational state only if you have visited it before [25, Page 186]. Only back leap may be expressed using continuation therefore. Fragments allow one to do forward jump. This is used to express a control between different software component. Back leaps are also permitted in FlowTalk, only if a recursion is tail-call. The selling point of continuations is to permit leap on non tail-call.

The second difference is about first-class entities. Continuations need to be first-class entities in order to be stored in a variable and retrieved later on. In FlowTalk, a programmer cannot obtain the reference of a fragment. Only the run-time has access to this.

Although similar on the surface, the two differences mentioned above make fragments distinct from continuations.

Static fragment allocation. One aspect of controlled disruption is that fragments need to be statically defined. Consequently, some variables are single assignment only. As explained above, fragment identification is achieved by identifying calls representing long-latency operations in the source code. To achieve this, *it is necessary to statically determine which long-latency operations can be invoked for each variable*. As a consequence, a variable that holds a sensor or a radio cannot hold a value different from the one given at composition time. Objects representing sensors or a radio cannot be passed as method arguments, and no object can be assigned to a variable that has been declared to refer to a sensor or a radio with composeWith:.

However, variables that are bound to any object that does not trigger a long-latency operation can be initialized and freely rebound at execution time. Restricting some instance variables to be single assignment is a consequence of statically identifying fragments. Although dynamic definition of fragments would be conceivable, it would probably come at a too high cost in term of resources.

Recursive method calls are permitted only if the call is an *tail call* (either returns a value without making a recursive call, or returns directly the result of a recursive call) or if the method does not contain any long-latency operation. As with most programming languages for wireless sensor languages, dynamic memory allocation is not permitted. Supporting recursion or reentrance for non-tail calls necessarily implies method context allocation. Our experience with FlowTalk shows that not supporting a general schema for recursion and reentrance does not represent a major obstacle.

Probably the restriction to use sensor or radio objects as message arguments may be partially softened by determining whether method variable arguments are polymorphic or not. In that case, data flow analysis needs to be applied. We plan to investigate this as future work.

Parallelism. In some situation, some operations may be re-

alized in parallel. For example, taking the scenario example given in Figure 1, it will be perfectly legitimate to perform the two sensors samplings at the same times since the light and magnetism sensors are two different physical components on the motherboard. This is a behavior that may be programmed in nesC and is supported by TinyOS [22].

However, the controlled disruption makes long-latency operation blocking, preventing them to be executed in parallel. Our experience based on the realization of several applications involving sensors sampling and data sending did not make parallelism an important wish. On the other side, we suspect parallelism to reduce the time the micro-controller is active, hopefully leading to a reduced battery consumption. Currently, FlowTalk does not support parallelism but will definitely be investigated in the future.

Interaction with nesC. As it will be described in detail in the coming section, a FlowTalk application is mapped into a set of TinyOS components. A FlowTalk class may implement a TinyOS interface either to be seen by the operating system in case of events, or for being interfaced with a nesC application. By "implementing a TinyOS interface" we mean that each function declared in the interface has to be associated to a method defined in FlowTalk. The fact that FlowTalk is translated into nesC makes the interaction between these two languages trivial: a TinyOS component generated from a FlowTalk application may be interfaced with a nesC application as any plain nesC component.

Speed overhead. Unfortunately, time profiling techniques for wireless sensor networks have not gained a significant acceptance. We therefore have not been able to conduct any experiment to measure speed overhead since we haven't found any usable tools. However, since the majority of applications for wireless sensor spend most of their time in waiting for incoming network message or timer events, the usability of FlowTalk should not be impacted, even in the case of a significant overhead.

Applicability to other languages. Coroutines and continuations have been traditionally employed to model the control flow of applications in the presence of multiple entry and exist points in various settings (*e.g.,* Operating systems [12], Web servers [13], [29]). Although we have not demonstrated the applicability of controlled disruption to other languages, we believe there is no major obstacle to have controlled disruption in nesC and in Java.

TinyOS/nesC specificity. FlowTalk offers a linguistic construct to deal with long-latency operations in a blocking fashion without incurring significant overhead. The root of the problem solved by FlowTalk is callback execution ordering. TinyOS cannot order callback execution. As we illustrated earlier (Section 2), the only way for a nesC programmer to palliate this is to have a *reification* of the execution control flow, traditionally realized with global variables. TinyOS/nesC cannot ensure callback ordering because several instances of the same events must be bound to the same callback. As exemplified in Section 2.2 the callback associated to the radio (SendMsgRadio.sendDone(...)) is invoked at each radio transmission, even if the transmissions occur at different stage in the scenario.

5 Implementation

The FlowTalk compiler performs a global whole program analysis. This section describes different parts of the implementation and code generation of the FlowTalk compiler. Our compiler is freely available[7]. It is intended to be used with TinyOS, and generates nesC code as an intermediate language, before being compiled into machine code. We compile FlowTalk to nesC for reasons of convenience.

FlowTalk based on TinyOS and nesC. TinyOS [39] is an open-source operating system designed for wireless embedded sensor networks. It provides an abstraction of the executing platform to enable applications to be easily ported to various embedded device models. FlowTalk generates executable code for TinyOS.

The FlowTalk compiler generates executables intended to be run using TinyOS. As nesC [17] is the language meant to be used with this operating system, our compiler produces nesC source code. nesC is an extension to the C programming language with some linguistic modularity constructs and makes long-latency operations split-phase. An *interface* specifies a set of function signatures that are provided by, and expected from, a module. A *module* is a set of functions and variable definitions. A module can implement several interfaces. The TinyOS task queue is used by FlowTalk to schedule the fragments.

Fragment and method context. A static interpretation multiple fragments composing a program. Those fragments are then mapped into nesC event handlers and tasks. We created a complete meta-model of the nesC language. Compiling a FlowTalk program creates an instance of this meta-model, which is then translated into nesC source code. Each class is translated into a nesC module, and wiring between modules is inferred from binding rules provided at composition-time (*cf.*, end of Section 3.2). A fragment is implemented in nesC as a task, augmented with a method context. A method context is statically allocated.

Translation into nesC. FlowTalk programs are translated into nesC. Parts of the code that do not contain any long-latency operation calls, are directly translated into nesC. For instance, a FlowTalk class defines the method display: used to display a value using a binary format on the leds:

7. http://bergel.eu/flowtalk.html

```
"FlowTalk code"
Leds >> display: value
   ((value bitAnd: 1) > 0)
      ifTrue: [self redOn]
      ifFalse: [self redOff].

   ((value bitAnd: 2) > 0)
      ifTrue: [self greenOn]
      ifFalse: [self greenOff].

   ((value bitAnd: 4) > 0)
      ifTrue: [self yellowOn]
      ifFalse: [self yellowOff].
```

```
// nesC code
result_t display(uint16_t value) {
   if ((value & 1) > 0) {
      call Leds.redOn();
   } else {
      call Leds.redOff(); }
   if ((value & 2) > 0) {
      call Leds.greenOn();
   } else {
      call Leds.greenOff(); }
   if ((value & 4) > 0) {
      call Leds.yellowOn();
   } else {
      call Leds.yellowOff(); }
   return SUCCESS;
}
```

A class representing an electronic component may be instantiated only once (Section 4). The self variable is directly mapped to the corresponding TinyOS component, Leds in the example above.

Polymorphism is supported naturally in FlowTalk since it is an object-oriented programming language. The only restriction applies to variables from which long-latency operation may be invoked, and in that case those variables cannot be rebound or passed as a message argument. No restriction applies to other classes.

```
RTObject subclass: #A
A foo { ^10 }

RTObject subclass: #B
B foo { ^20 }

RTObject subclass: #Application
   variableNames: 'a b leds'

Application bar: object {
   ^object foo
}

Application main {
   leds display: ((self bar: a) + (self bar: b))
}
```

Classes A and B define a method foo that returns 10 and 20, respectively. The caret mark (^) indicates a return statement. The class Application defines three variables, a, b and leds. It also defines a method bar: which requires an argument. It simply returns the result of invoking the method foo on the passed object. The main method displays the sum of the two invocations of foo: on the leds. This example involves a polymorphic variable, object.

When generating nesC code, classes are flattened into a set of nesC functions. Each object contains a numerical value that identifies its class. For instance, the example Application is translated into nesC as follows:

```
typedef struct {uint16_t id;} *Object;
typedef struct {uint16_t id;} ClassA;
typedef struct {uint16_t id;} ClassB;
ClassA t1;
ClassB t2;
ClassA* a = &t1;
ClassB* b = &t2;
uint16_t foo (Object object) {
   if (object->id == 1) {return 10;}
   else if (object->id == 2) {return 20;}
}
uint16_t bar (Object object) {
   return foo(object);
}
```

```
command result_t StdControl.init() {
   a->id = 1;
   b->id = 2;
   // Display 3 on the leds
   display (foo((Object)a) + foo((Object)b));
   return SUCCESS;
}
```

During the compilation, an identifier is given to each class. In this example, the class A has the identifier 1 and B has the identifier 2. Methods that are invoked through polymorphic calls are translated into a set of conditional statements to select the proper method to invoke (*e.g.,*foo(Object object)). The function init() in the component StdControl is the entry point for a nesC application. This function is invoked when the program is downloaded to the embedded device. The main method is translated into the init() function.

Translating a FlowTalk application that contains long-latency operations inserts global variables to represents linearized instance variables, temporary variables and the application control flow. Each fragment is associated with a numerical identifier. The resulting output of the sensing application is presented in Section 2.2.

Mapping to TinyOS components. A FlowTalk class does not have to be mapped to a TinyOS component if it does not model an electronic component. Each class that describes a TinyOS component has to declare methods that trigger a long-latency operation. Moreover, it has to specify to which component it has been mapped. For instance, the class LightSensor is defined as the following:

```
LightSensor initialize {
   self implementInterface:
         ((Interface @ #StdControl) aliasedName: #StdControlPhoto)
   self implementInterface:
         ((Interface @ #ADC) aliasedName: #ADCPhoto)
}
LightSensor tinyOSComponentName { ^#Photo }
```

The methods initialize and tinyOSComponentName define the mapping of the FlowTalk class LightSensor to the TinyOS component Photo. Calls to the light sensor are made through two interfaces, StdControl and ADC.

The StdControl interface is defined as:

```
Interfaces at: #ADC put:
   ((Interface new
      addMethods: {
         #read ->
               (MethodPrototype for:
                  'async command result_t getData()') .
         #dataReady: ->
               (MethodPrototype for:
                  'async event result_t dataReady(uint16_t value)')
      }) addCallback: #dataReady: for: #read; yourself).
```

Interfaces is a dictionary that contains all the nesC interfaces. The interface ADC declares two methods, read and dataReady:. The method read triggers a long-latency operation because of the callback declaration. The method read is mapped into a nesC command which has the prototype async command result_t getData(). The method dataReady: is never called in a FlowTalk program, it is only used to indicate that read is a long-latency operation and it designates the prototype of the nesC function callback.

Application name	Executable size (FlowTalk)	Executable size (nesC)	Ratio
CounterToLeds	1'532+46	1'570+46	0.97
SensorToLeds	2'382+81	2'348+67	1.01
RadioToLeds	8'994+326	8'790+326	1.02
CounterToLedsAnd-Radio	9'674+354	9'398+384	1.02
SensorToRadio	10'088+356	9'618+386	1.04

Fig. 7. Ratio between the nesC and FlowTalk version of a number of programs

Operation sequencing. In FlowTalk, long-latency operations complete in the order of their invocation. As illustrated in Section 2, nesC does not provide a dedicated mechanism for this ordering. The fragment identifier contained in method contexts is used to achieve operational sequencing.

Two fragments constituting a method cannot be present in the fragment queue at the same moment. Only when a fragment has completed, will the following fragment be inserted into the queue. The execution of a method is represented by the sequential execution of fragments, where the fragment id numbers are increasing.

6 Memory and Battery Consumption

We ported a number of applications from nesC to FlowTalk and measured the memory and battery consumption against the original nesC versions.

Memory consumption. The table below shows for 5 applications the memory consumption once deployed in the embedded device. A part of the application is stored in the read only memory (ROM) and another part in the random access memory (RAM). The ROM contains static data such as program code and the RAM contains volatile information such as variables.

The column titled *FlowTalk executable* indicates the size of the applications[8]. The value 1532+46 means that the application *CounterToLeds* consumes 1532 bytes of ROM and 46 bytes of RAM. The *nesC executable* indicates the size of the application written in nesC.

Those results shows that the controlled disruption has a very light cost in terms of memory consumption. The largest application (∼10 kB) has a penalty of only 4% compared to the original nesC application.

The FlowTalk version of the CounterToLeds application is slightly smaller than the nesC version. Although the reason for this difference might be found deep into the nesC compiler,

8. The nesC version of those applications are CntToLeds, CntToLedsAndRfm, SenseToLeds, SenseToRfm, RfmToLeds. Their source code is accessible online on www.tinyos.net/tinyos-1.x/apps. The FlowTalk version is accessible in the distribution available online.

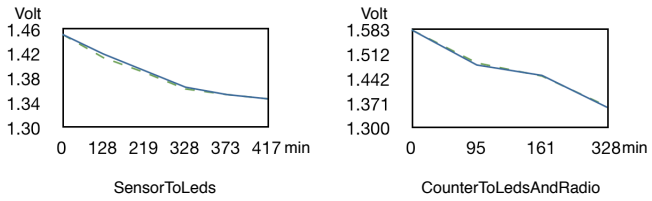

Fig. 8. Battery consumption

we believe that there is no apparent reason for this. There is probably a missed optimization in the nesC compiler.

The controlled disruption mechanism has practically no memory overhead mainly because only a low number of them are necessary in the example we provided.

Memory consumption. Measuring battery consumption accurately is a non-trivial task since no support are provided by TinyOS. We used a battery voltmeter for that purpose.

We conducted the experiment as follows. We took 4 CrossBow wireless sensor networks mica mote2:

- Mote A and B are powered each with 2 batteries Duracell ProCell MN1500 LR6, 1.5 V, AA. The SensorToLeds application provided with the TinyOS distribution runs on Mote A. On Mote B we run the FlowTalk version of it.
- Mote C and D are powered each with 2 batteries Ducacell Rechargeable 1700 mAh, AA HR6, DC1500 Ni-MH, 1.2 V. These batteries are available in any drugstore. The CounterToLedsAndRadio application distributed from the TinyOS distributions runs on Mote C. Similarly, we run the FlowTalk version of the application on Mote D.

We use different set of batteries to cover most typical usage set and situations.

We tuned the application to a high regime (very short timer interval), leading to a quick consumption. Mote A and B stopped working after 417 minutes, and Mote C and D after 328 minutes. Figure 8 shows the battery voltage against the time in minutes. Each motes has two batteries. Since the consumption of the two batteries in each mote is almost identical, we use 1 graph to represent the 2 mote batteries. Actually, few millivolts may differ between the batteries in a mote, but we are reaching the limit of accuracy of measuring tools at that level.

For each of the two applications, the nesC and FlowTalk versions have a consumption almost identical. According to the figures, the FlowTalk version consumes slightly more (2 %) than the nesC version for the SensorToLeds application. The reason is probably due to some cycle to handles frames. This experiment shows that fragment manipulation does not incur any unnecessary cycle-consuming addition that would be avoided in nesC.

7 Related Work

Being fragile to control flow disruption is a property shared by most event-based systems. Programming Internet [13] and concurrent programming [7] drag their own bag of similar

problems. FlowTalk focuses on the particular issue of long-latency operation in Wireless Sensor Networks.

Several works in the field of programming models for embedded devices are related to FlowTalk. The section groups the related piece of work by the topic they share with FlowTalk:

- *Programming languages:* AmbientTalk, Java Card, RTSJ, TaskJava, Virgil
- *Concurrency:* Clarity, galsC, Mantis, Protothread
- *Component model:* Pecos
- *Global programming:* DESEJOS, Maté, Pleiades, SpatialViews
- *Reducing memory consumption:* Jepes
- *Hardware:* SunSPOT
- *Synchronous languages:* Esterel, Reactive objects

7.1 Programming languages

AmbientTalk. Mobile networks surround a device equipped with wireless technology, and are demarcated dynamically as users move. AmbientTalk [11] is a new programming language making programming of mobile networks easier. It provides linguistic constructs to deal with (i) volatile connections, (ii) context-awareness based on the surrounding environment, (iii) serverless autonomous computing, and (iv) concurrency.

The main focus of AmbientTalk is on the definition of mobile network services. FlowTalk has different priorities. While AmbientTalk aims at managing highly distributed, dynamic connections, FlowTalk provides an efficient way of dealing with long-latency operations in the presence of strong resource restriction. AmbientTalk employs a purely event-driven concurrency framework, founded on actors enabling asynchronous and non-blocking message passing. Actors are usually tied to strong resource consumption, assuming the presence of a thread mechanism and a scheduler. There is currently no perspective in applying AmbientTalk's ideas to wireless sensor networks.

Java Card. Java Card technology[9] enables smart cards (devices with very limited memory) to run small applications that employ Java technology. It provide smart card manufacturer with a secure execution platform. The Java Card 2.1 Virtual Machine Specification defines a subset of Java that fits very constrained resources. Among other things, threads are not supported.

The range of applications supported by Java Card differs from the one of wireless sensor networks since a Java Card program is not meant to be at the heart of an highly event-based environment.

Real-Time Specification for Java (RTSJ). An important concern for meeting hard real-time constraints in Java is the interaction of automatic memory management with real-time operations. The time the garbage collection process will take to free all un-necessary objects cannot be predicted. Real-Time Specification for Java [2], [44] uses a memory model based on scoped types. Scoped types enable timely memory reclamation and allow for predictable performance.

9. java.sun.com/products/javacard

Whereas several experimental cases were conducted with major industrial companies in aeronautics, the example embedded devices used contained fairly large amounts of memory (256 Mb SDRAM and 32 Mb FLASH [3]). FlowTalk is intended to be used with systems that offer far smaller amounts of memory.

Tasks. It has been widely acknowledged that the event-driven programming style severely complicates program maintenance and understanding as it requires a fragmentation of the logical flow of control into multiple and independent callbacks.

Fisher *et al.* [16] proposed *Tasks*, a variant of cooperative multi-threading, that allow "each logical control flow to be modularized in the traditional manner, including usage of standard control mechanisms like procedures and exceptions." They produced a backward compatible extension of Java, called *TaskJava*.

The TaskJava type system tracks the set of methods whose execution might yield using the `async` method annotation. The compiler translates annotated methods into Java code that uses a continuation-passing style. This mechanism is similar to controlled disruption: fragments are statically located at compile time and the code is cut into smaller pieces following a reification of the control flow. We did not use a type system since FlowTalk does not annotate variables and methods with types, but the annotation mechanism is indeed similar. The major difference lays in this reification: TaskJava assumes full continuation to be modeled in the base language, whereas FlowTalk restricts the operations that can be performed on a method that refers to long-latency operations.

Virgil. An approach to use object-orientation with embedded microcontrollers has been recently proposed with Virgil [40]. Virgil is a lightweight object-oriented language designed with careful consideration for resource-limited domains. The main contribution of Virgil is to decouple the initialization time from runtime which allows an application to run on a bare hardware. Although FlowTalk proposes an object model for the same range of hardware, FlowTalk differs from Virgil by explicitly supporting execution control flow over long-latency operations.

7.2 Concurrency

Clarity. A new programming language has been recently proposed to tackle issues related to asynchronous systems components. Clarity [8] enables analyzable design of asynchronous software components. Compared to classical event-based systems, Clarity has three features: (i) *nonblocking* function calls which allow event-driven code to be written in a sequential style. If a blocking statement is encountered during the execution of such a call, the call returns and the remainder of the operation is automatically queued for later execution; (ii) *coords*, a set of high-level coordination primitives to encapsulate common interaction between asynchronous components and make high-level coordination protocols explicits; (iii) *linearity annotations* delegate coord protocol obligation to exactly one thread at each asynchronous function call,

transforming a concurrent analysis problem into a sequential one.

When a nonblocking call is performed, if the hardware is not ready, the caller returns a particular value and the remainder of the computation is automatically converted into a closure and put into a queue. The executing platform must allow for stack reification. This is achieved by using threads. In FlowTalk, what constitutes method contexts is statically known since the location of long-latency operations is known by the compiler. As a result, FlowTalk does not need to attach a stack to fragments. FlowTalk is designed for wireless sensor networks, whereas Clarity tackles issues on asynchrony assuming the presence of enough resources to hold continuations.

galsC. The galsC [9] programming language extends nesC with advanced abstractions for concurrency. In galsC, components are linked to each other to form *actors*. Messages that are exchanged within an actor are synchronous. Actors communicate with each other asynchronously via message passing, which separates the flow of control between actors. Actors are typed, and those types are inferred based on the graph formed by the actors.

Whereas the problem tackled by galsC is similar to the one in this paper, the approach is different. In galsC, the programmer has to manually specify where asynchrony may occur by specifying junction points between actors. In FlowTalk, a long-latency operation (which is asynchronous from the point of view of the operating system but not from FlowTalk) is statically located by the compiler. In FlowTalk, when a method is defined, the annotation LongLatencyOperation needs to be added depending on whether it directly invokes a primitive of the operating system that is asynchronous. Contrary to galsC, each call does not need to be annotated. Moreover, FlowTalk assumes that long-latency operations are synchronized.

Mantis. A multithreaded embedded operating system for wireless sensor network has been recently proposed to alleviate limitations of the TinyOS event-based architecture. Mantis [5] is a sensor operating system that supports a thread mechanism with a lightweight RAM footprint making it possible to fit in less than 500 bytes of memory.

The multithreading of Mantis is useful to prevent one long-lived task from blocking execution of a second time-sensitive task, mitigating the bounded buffer producer-consumer problem. Mantis is meant to be used with time consuming processing such as composition and encryption. The goal of Mantis differs from the one of FlowTalk since long-latency operations remains unchanged with Mantis whereas they are made blocking with FlowTalk.

Protothread. FlowTalk focuses on the issue of implementing sequential control flow on top of an event-driven system. This issue has been partially addressed by Protothread [15]. Protothreads are an extremely lightweight, stackless threads that provide a blocking context on top of an event-driven system, without the overhead of per-thread stacks. Protothreads provides conditional blocking inside a C function.

All protothreads run on the same stack and context switching is done by stack rewinding. This is advantageous in memory constrained systems, where a stack for a thread might use a large part of the available memory. A protothread requires only two bytes of memory per protothread.

A protothread runs within a single C function and cannot span over other functions. A protothread may call normal C functions, but cannot block inside a called function. Blocking inside nested function calls is instead made possible by spawning a separate protothread for each potentially blocking function. The advantage of this approach is that blocking is explicit: the programmer knows exactly which functions that may block and which functions are not able to block.

Protothread differs from FlowTalk on several points: (i) FlowTalk fragment are not user specified, but inferred by the FlowTalk compiler according to the declared long-latency operations whereas protothreads need to be explicitly declared; (ii) it is not possible for a regular function called from a protothread to block inside the called function whereas FlowTalk allows a fragment to invoke a long-latency operation to complete the fragment execution.

FlowTalk fragments are mapped into TinyOS tasks. TinyOS tasks are very similar to prototalk. Mapping FlowTalk fragments to protothreads is therefore realizable.

Scala. The "inversion of control" is a negative effect that may be found in most of event-driven programming models ranging from web applications [36] to concurrent based models [21]. The inversion of control happens when an execution environment dispatches events to the installed handlers, resulting in an inversion of the control over the execution of program logic since the program never calls these event handlers itself.

The inversion of control issue is at the root the control flow disruption we formulated: the main application control flow necessarily moves to places where event handlers are located.

The Scala[10] general purpose programming language provides *event-based actors*, "an implementation technique for lightweight actor abstractions on non-cooperative virtual machines such as the JVM. Non-cooperative means that the virtual machine provides no means to explicitly manage the execution state of a program." The idea is as follows: an actor that waits in a receive statement is not represented by a blocked thread but by a closure that captures the rest of the actor's computation. The closure is executed once a message is sent to the actor that matches one of the message patterns specified in the receive. The key is that the execution of a closure is carried by the thread of the sender.

Haller and Odersky [21] have considered the case of actor based system using a queue between each interacting actors. Moreover, the presence of closure is a necessary condition for the Scala scheme to work. FlowTalk addresses this issue differently by restricting operations associated to long-latency operations. This relax the condition on supporting closure.

10. www.scala-lang.org

7.3 Component model

Pecos. The goal of the Pecos (PErvasive COmponent Systems) project [32] was to enable component-based technology for a class of embedded systems known as "field devices" such as temperature, pressure, and flow sensors, actuators, and positioning devices. Pecos proposes a component model for field devices that captures a range of non-functional properties and constraints.

Pecos uses a discrimination of components and necessitates realtime constraints to schedule components. Pecos is made to ease gaining a better software architecture. FlowTalk focuses on a finer grain, application control flow.

7.4 Global programming

Design space exploration tool. DESEJOS [30] (DEsign of Software for Embedded Java with Object Support) is a tool that enables an automatic selection of the best organization for objects in a program written in Java. This deals with the objects after the programmer has coded the application, and before the code execution. The DESEJOS tool automatically transforms dynamically created objects into statically allocated ones when possible.

The DESEJOS approach assumes the use of a virtual machine and a target platform supporting a memory management function with garbage collection. FlowTalk runs on platforms having much stronger resource constraints.

Maté. Deployment of wireless sensor application becomes a critical task when the networks of devices is large or when not easily physically accessible. The Maté virtual machine [26] enables programs to be acquired from a wireless sensor network using ad-hoc routing and data aggregation algorithms. The set of Maté bytecode instructions is designed to reduce the size of programs in order to reduce energy consumption when wirelessly transmitted. FlowTalk provides high level language constructs to better express application control flow, which is not the main priority of Maté.

Pleiades. A wireless sensor application is traditionally written from the point of view of one particular node in the network. Pleiades [24] is a new programming language for which an application is implemented as a central program that conceptually has access to the entire network. Pleiades augments the C language with constructs for addressing the nodes in a network and accessing local state from individual nodes. The Pleiades programming model borrows from a previous work on Kairos [18], an extension to Python that also provides support for iterating over nodes and accessing node-local state.

FlowTalk retains the node-centric view and tackles the problem of asynchrony within one node instead of focusing on a network-centric view as promoted by Pleiades.

SpatialViews. The programming model offered by SpatialViews [31] allows for the specification of virtual ad-hoc networks by describing nodes. A collection of virtual nodes is described as a view over the real, physical network. Each node may have various characteristics such as a set of provided services and a location. A service is described in terms of a set of methods, and are provided by a node by means of a concrete implementation.

SpatialViews provides high level operators to manipulate networks such as iteration, migration, and service offering. SpatialViews places itself as a global management system, whereas FlowTalk adopts a local view. Instead of specifying global properties, FlowTalk allows for a behavior description of one single mote.

7.5 Reducing memory consumption

Jepes. Static analysis of Java programs for embedded devices is one of the most distinctive features of Jepes [37]. Jepes targets embedded devices having a memory ranging from 0.5 kB to 5 kB by reducing Java program executables up to 75%. Jepes supports two aggressive optimisations for a Java program: (i) Assembly macros can be embedded into Java methods and (ii) Objects may be allocated in the stack rather than in the heap. Those optimisations are driven by annotating the program where those optimisations are likely to occur.

The initial motivation of Jepes is different than the one of FlowTalk. Jepes takes standard Java and a set of annotations as input. FlowTalk constrains the writing of programs by forbidding dynamic object creation and restricting objects passing.

7.6 Hardware

Java can be used to write applications for embedded sensor networks thanks to SunSPOT, based on the Squawk virtual machine [38]. SunSPOT is written in Java and does not need an operating system to be run on. It supports thread scheduling and garbage collection. SunSPOT's features are very close to that of a modern virtual machine. SunSPOT favors large amount of memory ranging from 512 kB up to 4,096 kB. This goal differs from FlowTalk's, where platforms may support amounts of memory 1000 times smaller than the amount intended to be used with SunSPOT.

However, studying the API[11] remains interesting. The Java API used to program SunSPOT employs asynchronous and synchronous calls to talk to physical components located on the motherboard. For example, EDemoController.accelerometerScaleChanged(int newScale) is called by the runtime when a variation is detected by the accelerometer. This method has to be overridden in subclasses. ILightSensorController.getLightSensorValue(...) is a blocking operation. ITemperatureInput.addITemperatureInputThresholdListener(...) adds a specified temperature sensor threshold listener to receive callbacks from a temperature sensor. This method is used to register a callback, which is an instance of ITemperatureInputThresholdListener. Alternatively, a user mays ask for the temperature value in a synchronous fashion with ITemperatureInput.getCelsius().

11. http://www.sunspotworld.com/docs/javadoc

7.7 Synchronous languages

Synchronous languages [20] provide primitives for achieving a kind of parallelism based on the hypothesis of perfect synchronism. This hypothesis assumes that a computation is performed instantaneously. Esterel [4] is probably the most famous synchronous language. Esterel translates a program into a finite-state machine.

Reactive objects [34], [35] assimilate every object as an autonomous unit of execution. An object is either executing the sequential code of exactly one method, or passively maintaining its state. Reactive objects impose a restriction on method execution by ensuring that methods do not block execution indefinitely.

Assumptions in FlowTalk are different than the ones of synchronous languages. Whereas FlowTalk is meant to produce software that is "reactive enough" to be synchronized with the current environment, it does not provide a guarantee about the duration such operations might take. Moreover, it is the responsibility of the programmer to ensure the computation to be deterministic. FlowTalk helps in modelling an application control flow in a resource restrained context, whereas synchronous languages focus on expressing synchronization and parallelism.

8 Conclusion

With the growing complexity of embedded devices, it is becoming necessary to use current software engineering techniques and methodologies to increase software productivity. Object modeling and design is a widely-known methodology intended to satisfy software portability, maintainability, and to shorten development time.

This paper presents FlowTalk, a new programming language aimed at making the programming of sensor-embedded devices easier by providing an efficient mechanism to express an application control flow. In order to cope with the asynchrony that may occur between different electronic components on the host, FlowTalk provides a technique named *controlled disruption*.

A fault management mechanism is currently lacking in FlowTalk. The majority of the examples and experiments were conducted in a closed environment with adequate battery power supply. In order to meet industrial and real-life situations, it is planned to stress the expressiveness of FlowTalk in different situations that can result from movement or environmental factors (*e.g.,* effect of the wind on the radio transmission). The choice of a particular error management policy is dependent on the ambient external environment. These policies should be easily added and interchangeable and as they are likely to crosscut an embedded application, the use of aspect-oriented techniques in this area is promising [19]. Finally, it is planned to use simulation and tracing techniques to obtain knowledge about the dynamic execution, in order to ease restriction on instances variables.

The FlowTalk compiler is freely available and the distribution contains every application mentioned in this paper and many more.

Embedded devices and sensor networks have been in constant evolution. However, corresponding programming techniques and methodologies for this particular field have not equally evolved. We hope the work presented in this paper is a positive step towards a better way of programming very small embedded devices.

Acknowledgments. FlowTalk is the result of an intense research effort that begun in 2006. This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

We also would like to thank Shane Brennan, Raymond Cunningham, Stéphane Ducasse (for suggesting "FlowTalk"), David Gay, Ron Goldman, Oscar Nierstrasz, Andreas Polze and his research group, Lukas Renggli and Aline Senart for their valuable comments. We also would like to thank Marcin Karpinski for his precious help with TinyOS. Several technical problems had to be faced when experimenting. Our thanks go to Alain Fargue, Greg Guche, and all the POPS INRIA Project Team.

References

[1] J. R. Andersen, L. Bak, S. Grarup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen. Design, implementation, and evaluation of the resilient smalltalk embedded platform. In *Proceedings of ESUG International Smalltalk Conference 2004*, Sept. 2004.

[2] C. Andreae, Y. Coady, C. Gibbs, J. Noble, J. Vitek, and T. Zhao. Scoped types and aspects for real-time Java. In *Proceedings ECOOP '06*, volume 4067 of *LNCS*, pages 124–147. Springer-Verlag, July 2006.

[3] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbuster, E. Pla, and D. Holmes. A real-time Java virtual machine for avionics. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*. IEEE Computer Society, 2006.

[4] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[5] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[6] E. Brewer, D. Culler, D. Gay, P. Levis, R. von Behren, and M. Welsh. nesC: A programming language for deeply networked systems. http://nescc.sourceforge.net.

[7] D. Caromel. Programming abstractions for concurrent programming. In *TOOLS Pacific '90*, pages 245–253, Sydney, Australia, Nov. 1990.

[8] P. Chandrasekaran, C. L. Conway, J. M. Joy, and S. K. Rajamani. Programming asynchronous layers with clarity. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 65–74, New York, NY, USA, 2007. ACM.

[9] E. Cheong and J. Liu. galsc: A language for event-driven embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1050–1055, Washington, DC, USA, 2005. IEEE Computer Society.

[10] M. E. Conway. Design of a separable transition-diagram compiler. *Commun. ACM*, 6(7):396–408, 1963.

[11] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-oriented programming in ambienttalk. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*, volume 4067, pages 230–254. Springer-Verlag, 2006.

[12] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 122–136, New York, NY, USA, 1991. ACM Press.

[13] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[14] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[15] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[16] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 134–143, New York, NY, USA, 2007. ACM.

[17] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[18] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, New York, NY, USA, 2005. ACM.

[19] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan. Declarative failure recovery for sensor networks. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 173–184, New York, NY, USA, 2007. ACM.

[20] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[21] P. Haller and M. Odersky. Event-based programming without inversion of control. In *In Proceedings of Join Modular Programming Languages (JMLC)*, volume 4228, pages 4 – 22, Sept. 2006.

[22] M. Karpinski and V. Cahill. High-level application development is realistic for wireless sensor network. In *In Proceedings of Fourth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks SECON 2007*. IEEE, June 2007.

[23] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice Hall Software Series, 1978.

[24] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 200–210, New York, NY, USA, 2007. ACM.

[25] S. Krishnamurthi. *Programming Languages: Application and Interpretation*. Shriram Krishnamurthi, 2007.

[26] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[27] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[28] K. Matheus, R. Morich, C. Menig, A. Lübke, B. Rech, and W. Specks. Car-to-car communication - market introduction and success factors. In *In the Proceedings of the 5th European Congress and Exhibition on Intelligent Transport Systems and Services (ITS'05)*, 2005.

[29] J. Matthews, R. B. Findler, P. Graunke, S. Krishnamurthi, and M. Felleisen. Automatically restructuring programs for the web. *Automated Software Engineering: An International Journal*, 2003.

[30] J. C. B. Mattos, E. Specht, B. Neves, and L. Carro. Making object oriented efficient for embedded system applications. In *SBCCI '05: Proceedings of the 18th annual symposium on Integrated circuits and system design*, pages 104–109, New York, NY, USA, 2005. ACM Press.

[31] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. *SIGPLAN Notice*, 40(6):249–260, 2005.

[32] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. V. D. Born. A component model for field devices. In *Proceedings First International IFIP/ACM Working Conference on Component Deployment*, pages 200–209, Berlin, Germany, June 2002. ACM.

[33] J. Noble and C. Weir. *Small Memory Software: Patterns for systems with limited memory*. Addison-Wesley Professional, Nov. 2000.

[34] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, Götebord, Sweden, May 1999.

[35] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black. Reactive objects. In *In Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed computing*, Crystal City, Virginia, USA, Apr. 2002.

[36] C. Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[37] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. L. Knudsen. Compiling Java for low-end embedded systems. *SIGPLAN Notice*, 38(7):42–50, 2003.

[38] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[39] TinyOS: An open-source OS for the networked sensor regime. http://www.tinyos.net.

[40] B. L. Titzer. Virgil: objects on the head of a pin. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 191–208, New York, NY, USA, 2006. ACM.

[41] B. A. Warneke and K. S. Pister. Exploring the limits of system integration with smart dust. In *Proceedings of IMECE'02, ASME International Mechanical Engineering Congreee & Exposition*, Nov. 2002.

[42] O. Zendra. Memory and compiler optimizations for low-power and -energy. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOPLPS'06), co-located with ECOOP'06*, July 2006.

[43] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *Proceedings of the 11th International Conference on Compiler Construction (CC'02)*, volume 2304 of *LNCS*, pages 14–28. Springer-Verlag, 2002.

[44] T. Zhao, J. Noble, and J. Vitek. Scoped types for real-time Java. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04)*, pages 241–251, Washington, DC, USA, 2004. IEEE Computer Society.

Alexandre Bergel is Assistant Professor at the University of Chile. He obtained his PhD in 2005 from the University of Berne, Switzerland, under the supervision of Prof. Nierstrasz and Prof. Ducasse. His thesis focused on a new module system to ease software evolution and extension for large software systems. He was a post-doc at Trinity College Dublin. In 2007 he went to Germany and became a Research Fellow at the Hasso-Plattner-Institut, Potsdam. Until May 2009 he was Permanent Researcher at INRIA, a French based research institute. He worked on defining new programming language constructs and the Moose reengineering platform to ease evolution of program source code. Contact him at bergel.eu.

William Harrison is an SFI Research Professor in the School of Computer Science and Statistics, Trinity College, Dublin. He heads a group researching programming language fundamentals for creating more malleable software. His was previously on the Research Staff of IBM's Thomas J. Watson Research Center exploring software for software development, initially on programming languages and program analysis and optimization and later on software development environments. He is one of the founding

researchers in what has come to be called “Aspect-Oriented Software Development”. He was a member of the IBM Academy of Technology and an IEEE Distinguished Lecturer on Software Environments, and has been a member of IEEE since 1973. Contact him at www.cs.tcd.ie/Bill.Harrison.

Vinny Cahill holds a Personal Chair in Computer Science at Trinity College Dublin where he also serves as Head of the Department of Computer Systems and Director of Research for Computer Science and Statistics. His research addresses many aspects of distributed systems, in particular, middleware and programming models for pervasive and mobile computing with application to intelligent transportation systems and management of critical infrastructure. He has a particular interest in self-organising systems. He has published over 100 peer-reviewed publications in international conferences and journals. Contact him at www.dsg.scss.tcd.ie/∼vjcahill.

Siobhán Clarke is a Senior Lecturer and Fellow of Trinity College Dublin, where she leads the Distributed Systems Group and is a Research Area Leader in Lero: The Irish Software Engineering Research Centre. Her research interests are design and programming models for advanced, adaptable distributed systems. She received her BSc and PhD degrees in Computer Science from Dublin City University. Contact her at www.cs.tcd.ie/Siobhan.Clarke.