High-Level Application Development is Realistic for Wireless Sensor Networks

Marcin Karpiński and Vinny Cahill
Distributed Systems Group
School of Computer Science and Statistics
Trinity College Dublin
{karpinsm, vinny.cahill}@cs.tcd.ie

***Abstract*—Programming Wireless Sensor Network (WSN) applications is known to be a difficult task. Part of the problem is that the resource limitations of typical WSN nodes force programmers to use relatively low-level techniques to deal with the logical concurrency and asynchronous event handling inherent in these applications. In addition, existing general-purpose, node-level programming tools only support the networked nature of WSN applications in a limited way and result in *application* code that is hardly portable across different software platforms. All of this makes programming a single device a tedious and error-prone task.**

To address these issues we propose a high-level programming model that allows programmers to express applications as hierarchical state machines and to handle events and application concurrency in a way similar to imperative synchronous languages. Our program execution model is based on static scheduling what allows for standalone application analysis and testing. For deployment, the resulting programs are translated into efficient sequential C code. A prototype compiler for TinyOS has been implemented and its evaluation in described in this paper.

I. Introduction

Finding a suitable set of programming abstractions that address the specific nature of WSN applications is currently an active research topic. Many approaches have been proposed in the literature and they can be roughly grouped in to two categories: *macroprogramming*, where the operation of the whole network is defined as a single program, see for example [1], [2], and *node-centric* programming, a more traditional approach where the focus is on the behaviour of a single sensor node. Our programming model falls into the second category.

The severely constrained resources of a typical WSN node prohibit the use of traditional, high-level programming models, and features like automatic garbage collection or full-featured object orientation are out of reach of WSN application programmers. On the other hand, WSN applications, due to their concurrent nature, where hardware events may arise asynchronously in an unspecified order require more suitable programming tools than classical programming languages such as C/C++ or Java.

Currently the most popular WSN software development platform is TinyOS, an event-driven, lightweight sensor-node operating system written in the nesC [3] programming language. TinyOS applications are partitioned into modules which are wired together using bidirectional interfaces and communication between the modules is asynchronous, which means that time-consuming interface method calls are split (so called, *split-phase* execution) into an invocation part, a *command* call, and an acknowledgement part, an *event* notification. Building programs from chains of calls and callbacks, which is the essence of any event-driven programming framework, can be a very difficult and error-prone exercise for non-trivial programming tasks as discussed in [4].

Other node-centric programming frameworks that avoid the event-driven programming model focus on different aspects of WSN application development. They include resource-optimised operating systems that offer traditional multi-threaded programming environments with blocking system calls for hardware access (e.g., [5]), virtual machines [6] for easier node reprogramming that provide a low-level intermediate language which hides the split-phase execution model, or a higher-level approach [4] where applications are composed of hierarchical state machines and split-phase execution is avoided by defining application states and input/output actions.

The contribution of this paper is thus threefold. Firstly, we propose a high-level programming model that structures event handling through the use of a fine-grained concurrency operator in a similar way to imperative

synchronous languages [7]. In this model split-phase execution is avoided and programs are hierarchically composed of states that are treated as computational routines rather than sets of program variable values. Secondly, we give a precise program execution model that defines the semantics of the programming model and, as opposed to the rigorous semantics of synchronous languages, allows program execution on highly resource-constrained computing platforms. Thirdly, we show how such a programming model can be layered on top of a wide range of event-driven middleware and operating systems.

The paper is organised as follows: we start by describing the program execution model in Section II while the application programming model is introduced in Section III. In Section IV we show how the programming model can be integrated with existing event-driven systems and in Section V we present a prototype compiler implementation for our model designed for the TinyOS platform. We investigate the applicability of our model by analysing properties of the code generated by the compiler in Section VI. We conclude the paper with Discussion, Related Work and Summary sections.

II. Program Execution Model

A. Concurrency

The essential characteristic of our execution model is suppression of the split-phase program execution scheme. This means that all time-consuming hardware or operating system (OS) calls, which from now on will be referred to as *event calls*, are blocking. Since being blocked while awaiting for an event call to complete would render an application non-responsive for arbitrarily long periods of time, some form of parallelism and resumption from such situations needs to be provided. For that purpose, inspired by imperative synchronous languages such as [7], we use the concurrency operator [... || ...] which we allow to be freely nested and mixed with other constructs of the programming model in order to syntactically structure event handling code. The general intuition behind its operation is simple: a statement $[A||B]$ is interpreted as A and B executing in parallel and we refer to both statements A and B as *threads* of this operator. Parallel operators can have an arbitrary number of threads.

To allow for parallel execution of program fragments while keeping in mind the highly-constrained resources of WSN nodes we use static program scheduling. This means, in particular, that the execution order of all concurrently running parts of the program is determined at compilation time. Hence programs can be regarded as deterministic given that the order in which event calls are completed is set. We divide, therefore, all statements of the model with regards to their behaviour during concurrent execution into two categories: *atomic* and *non-atomic*.

Non-atomic statements include all event call statements plus a special statement yield. Upon execution, they act as points of thread interleaving (as in cooperative multithreading models) and we assume that event calls can take an unbounded amount of time to complete whereas the yield statement, generally speaking, completes shortly after it is executed. We can decompose, therefore, the execution of non-atomic statements into two independent operations: the invocation and the completion, and say that thread interleaving always takes place after such statement has been invoked and before it has completed. Intuitively, non-atomic statements are introduced to model split-phase execution since at the OS level event calls are often composed of two activities, i.e., call invocation and OS notification (a callback). In the case of the special-purpose yield statement no action is taken during its invocation and completion.

Atomic statements, on the other hand, include all other statements of the model. They are executed sequentially and without interruption until a non-atomic statement is encountered which, in turn, is invoked and the control is passed to another thread or to the OS. When the non-atomic statement completes the thread is resumed and continues execution. In order to define the complete program scheduling scheme we need to specify how parallel operators are executed. We say that:

1) The parallel operator executes all its threads in a round-robin manner according to the order of their declaration in the program.
2) The execution of threads proceeds in *steps*: atomic statements are executed until a non-atomic one is encountered. This statement is then invoked, the thread is suspended and control is passed to the next thread of the operator (if there is any).
3) A thread resumes execution when it receives control and its previous non-atomic statement has completed.
4) The parallel operator terminates when all its threads terminate or when it is interrupted.

This procedure is applied recursively when parallel operators are nested. To make it clearer we use the following example where statements A_i are atomic and $E = \{E_i, E_c\}$, $F = \{F_i, F_c\}$ are two event calls with their

invocation and completion components respectively:

$$A_0;\ \Big[A_1;\ E;\ A_2 \ \Big\|\ A_3;\ \Big[A_4 \ \Big\|\ A_5;\ F;\ A_6\Big]\ A_7\Big]$$

The execution order of the program is the following:

$$A_0,\ A_1,\ E_i,\ A_3,\ A_4,\ A_5,\ F_i,\ E_c,\ A_2,\ F_c,\ A_6,\ A_7.$$

Atomic statements will be executed until a non-atomic one is encountered. This means, in particular, that any infinite loop consisting only of atomic statements will not be suspended preventing other threads from further execution. The yield statement has been included in the model to provide a way of handling such situations. This statement is used simply to allow a thread to be suspended in cases where no event call is to be made and it completes in the next step of the thread's execution. Situations potentially leading to infinite atomic loops can be statically detected by the compiler and a warning can be issued.

Communication between threads can be realised through the use of local variables. This, however, could possibly result in race conditions but, on the other hand, this is not likely since threads are interleaved only at precisely defined points (non-atomic statements) and shared variable access is always realised as an atomic statement. Static program scheduling allows us to also detect the possibility of concurrent access to the same OS resource. For that, we define a *device* as a group of event calls related to the same underlying OS or hardware component and we prevent any two threads running in parallel from making event calls to the same device. The code snippet below shows how the above definitions work in practice.

```
1  TOS_Msg *msg = next_buffer(0), *send_buf;
2  int n=0;
3  [
4    while (true) {
5      request msg->data[n++] = Sensor.data;
6      request Timer.start(150);
7    }
8  ||
9    while (true)
10     if (n==10) {
11       send_buf = msg;
12       msg = next_buffer(msg);
13       n=0;
14       emit Radio.message(send_buf);
15     }
16     else yield;
17 ]
```

In this example one thread is periodically querying a sensor and the other one is sending this data out on the radio when ten readings have been taken; Sensor, Timer and Radio are devices that define data, start and message event calls respectfully (we omit parentheses with event calls that carry no parameters). Additionally, our programming model distinguishes between two types of event calls: *incoming*, which result in the application receiving some data, and *outgoing*, which only transfer data from the application to an underlying OS component. We use the emit and request keywords respectively to specify them in programs. Also, note that if the 16^{th} line were removed the second thread would block the whole application infinitely.

B. Interruption

The need for terminating an activity in progress often arises in the domain of embedded systems and is even more common in the case of networked embedded systems such as wireless sensor networks where messages from network neighbours cause a node to abandon its current task and proceed to another one. For instance, in the code example from the previous section one might be interested in adding another thread that is waiting for a particular radio message in order to stop sensor sampling and initiate a sensor calibration routine.

In our execution model, interruption is realised by breaking the execution of the whole parallel operator thus stopping all its threads. Such an action might lead, however, to inconsistencies. If a thread was interrupted while waiting for an event call to complete, this event call would not have a chance to be handled by the application. Moreover, if the same event call was attempted again immediately after the thread was interrupted, the underlying OS component might still be processing the previous invocation resulting in unpredictable behaviour. In order to prevent such situations the compiler reserves a flag (one bit) for each event call used in the program to decide at runtime weather a given call has completed. With this information two types of actions are taken upon executing an event call that is still in progress:

- *late synchronisation* - completion of the previous call is awaited and, as soon as this happens, the new call is invoked.
- *event call cancellation* - if the event call has been defined (see section IV) as capable of being cancelled, an appropriate OS call is made to do so and the new call is invoked immediately.

Our programming model defines a number of constructs that are able to interrupt the execution of parallel operators and they all share the same behaviour, i.e., the interruption takes place immediately after they are executed. One of them is the classic exception raising statement (we use the raise keyword), which together

with the try/catch block can be used as a universal way of stopping activities in progress. The following code presents a situation in which the application is constantly monitoring the radio for heart-beat messages from a nearby node. If no message is received for more than $1.5s$ an action is performed. The example shows also a general way of implementing watchdogs in our model. Note also, that the timer event call must be capable of being cancelled in order for the code to work.

```
1  while (!finish){
2     try{
3        [
4           request Radio.heartbeatMsg;
5           raise HeartBeat;
6        ||
7           request Timer.start(1500);
8           raise Timeout;
9        ]
10    }
11    catch Timeout {finish = true;}
12    catch HeartBeat {}
13 }
```

III. Programming Model

WSN applications due to their networked nature are often best expressed in terms of states and transitions between them, e.g., [8]. Therefore, inspired by the STATECHARTS [9] formalism, we propose that applications be hierarchically composed of *states*. We distinguish two kinds of states: basic states and conglomerates of states which we call *superstates*. The state hierarchy is constructed in a bottom-up manner, firstly, by composing superstates from basic states. Then, subsequent superstates are built from other basic states and superstates. The top-most state in the hierarchy is called the application. Figure 1 graphically presents an example application main which consists of a basic state S and a superstate super_st which, in turn, is a composition of two basic states P and Q.

Basic states, as opposed to in STATECHARTS or other similar formalisms, such as [4], are defined as ongoing activities. These activities can be thought of as classical imperative programming functions that can make use of the constructs presented in Section II and that do not return any values. Superstates can also be thought of as ongoing activities but, in contrast to basic states, these activities are expressed in terms of the way the states that comprise them are composed. Thus, basic states and superstates can be viewed as different levels of granularity describing the same thing, i.e., application activity.

State transitions can be triggered only from within basic states and are explicitly specified using the moveto

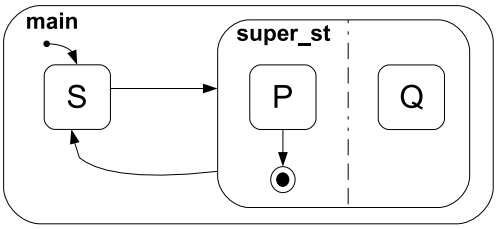

Fig. 1. Graphical representation of the example superstate definition.

statement. This statement interrupts in the same sense as the raise statement, therefore, all threads that might possibly be running at the time of its execution are stopped and the application proceeds to the next state (basic or not). Below we present an example basic state in which the application is periodically sampling a sensor. When the sampled values exceed a certain threshold, a transition is made to the state in which object tracking is performed. At the same time the application is waiting for a radio message from another node that forces it to abandon object detection and calibrate its sensor.

```
1  state detect_obj{
2    [
3      int val;
4      while (true) {
5        request val = Sensor.data;
6        if (val > THRESHOLD) moveto track_obj;
7        request Timer.start(50);
8      }
9    ||
10     request Radio.recalibrationMsg;
11     moveto sensor_calibration;
12   ]
13 }
```

There are essentially two ways of building superstates: by composing states sequentially or in parallel. In order to specify parallel state composition the parallel operator is used and individual states of the composition are treated as separate threads of the operator. The execution scheme of the operator remains the same as presented in the previous section. We also allow superstates to declare local variables to provide means of communication between their constituent states.

For the sake of code modularity, we define superstates as separate and independent entities that can be tested and analysed on their own. This approach, however, results in two important limitations. Firstly, no direct transition can be made from a superstate to a state defined in another superstate because this state would have to be explicitly referenced contradicting code modularity. Secondly, for the same reason local

variables of a superstate cannot be accessed from within other superstates. In order, however, to allow transitions between separate superstates we use named terminal states. Terminal states can be declared in a superstate and when transitioned to, transfer control to another state defined in a different superstate. The mapping between terminal and non-terminal states is done statically. The following code shows how this is done for the example superstate definition presented in Figure 1. The body of the superstate main only *declares* the state super_st while mapping, at the same time, its terminal state T to main's local state S. The superstate super_st is defined separately as a parallel composition of two basic states P and Q. Note that upon transition from state P to T state Q is interrupted.

```
1  superstate main{
2    initial state S{
3      while (true)
4        if (condition) moveto super_st;
5        else /* do something else */
6    }
7    superstate super_st{
8      super_st.T -> S;
9    }
10 }
11
12 superstate super_st{
13   terminal T;
14   int val=0;  /* a superstate variable */
15   [
16     state P{
17       /* do something */
18       moveto T;
19     }
20   ||
21     state Q{
22       while (true)
23         emit Actuator.doSomething(val++);
24     }
25   ]
26 }
```

IV. Integration with existing event-driven systems

Devices, as we mentioned in Section II, define points of contact between applications and the operating system. Event calls are meant to abstract away the three general communication schemes that are used to exchange information between applications and the OS: a single OS function call, a function-callback pair and a single callback invocation (e.g., an interrupt handler). Our model does not try, however, to completely hide the operating system from applications. In order to achieve resource efficiency and generality of our approach we closely relate event calls to the semantics of exposed OS interfaces. This means, in particular, that we want applications that are designed for a given OS to use its data types so we do not need to introduce any additional data translation layer. Although it may seem that applications are not portable across operating systems, this is true only to some extent. Using our model, one can build applications on top of almost any operating system (OS calls must not be blocking), middleware or software library and this fact allows us to pass the task of providing a unified WSN application programming interface to the underlying software layers.

We expect, generally speaking, the information required to integrate our programming model with a given operating system to come from two sources. Firstly, the compiler needs to have certain knowledge about the target execution platform and secondly, all other information regarding the mapping of particular OS components to their abstract application representations (such as radio messages, sensors or actuators) has to be specified as event calls and devices in the application code. Event call and device definitions are, therefore, mixtures of abstract concepts that belong to the programming model and of OS specific information interpreted by the compiler in the context of the target execution platform.

A device definition, specifies, apart from its event calls, the name of the OS component to which it maps. This name, depending on the compilation target, can refer, for example, to a C header file, a Java class or a TinyOS component, and the compiler needs to be able to recognise this object. Initialisation, starting and stopping (for hardware components) routines can be specified in the same way for a given device and the meaning of these definitions will be relative to the OS component to which they refer. We divide abstract OS components into subunits that we call *modules*. In reality, modules that comprise a component might correspond to different interfaces that this component exposes (as in Java or TinyOS) and that information can be important for the compiler to properly translate event calls that correspond to particular parts of the modules.

Each event call may specify, therefore, the module to which it belongs, the name of the OS function and the callback that comprise it, as well as, its cancellation routine. Event call parameters, however, are not specified. Instead, they are meant to be the same as those of the related OS function (if the event call specifies it). Additionally, incoming event calls need to specify what will be regarded as their return value and a callback return expression may need to be provided in cases where it is required by the OS API.

The OS integration scheme allows specification of erroneous event call conditions so that appropriate error handling code can be generated. We confine event call error checking to two cases. First, the invocation part (OS function call) may generate an error condition. This condition can be signalled at the OS level through the OS function call's return value or a value of a particular system error-status variable or function. Secondly, the failure information might be signalled by the OS at the time of event call completion by means of the callback parameters or by other means, as described before. In both cases the condition, expressed in terms of the host OS API, can be specified. Programmers can check in the application code for event call failure using the iferror keyword in the following way:

```
emit Radio.message(msg)
     iferror <error handling code>;
```

Below we present a device definition that specifies two event calls (one incoming and the other outgoing). The event call definitions specify how sending and reception of a particular radio message is mapped to appropriate TinyOS components, interfaces and methods. The outgoing event defines a split-phase communication scheme whereas the incoming one receives data from the OS through a callback named sendDone and uses the m callback parameter as the return value to be transferred to the application. Note that all double-quoted strings are TinyOS specific.

```
1  device Radio{
2    component = "GenericComm";
3
4    emit message{
5      module = "SendMsg[APP_MSG]";
6      function = "send",
7         iferror = "{return}==FAIL";
8      callback = "sendDone",
9         iferror = "success==FAIL",
10        return = "success";
11   }
12   request message{
13      module = "ReceiveMsg[APP_MSG]";
14      callback = "receive",
15         return = "m",
16         eventval = "m";
17   }
18 }
```

V. Implementation

Our programming model has been implemented as a combination of the ANSI C programming language and the concepts described in previous sections. The prototype implementation is based on the idea presented in [10] where the authors show how lightweight threads can be realised in a generic way using C switch statements and a set of preprocessor macros, as well as, how they can be used in application development. The prototype compiler generates ready-to-compile NesC code that can be immediately compiled and installed on the motes.

```
state S{
  int x=1;
  yield;
  x++;
}
```

```
int state_S(S_data *data){
  BOOL finished=FALSE;
  switch(data->PC_0){
    case 0:
      data->x=1;
      data->PC_0=1;
      break;
    case 1:
      data->x++;
      finished = TRUE;
  }
  if (finished) return 0;
  else return 1;
}
```

Fig. 2. A simplified translation of a single state example application.

The implemented static scheduling scheme is based on reentrant functions and nested switch statements to represent application states and specify program execution order. Translated programs are therefore *sequential* and they are executed as follows. The execution of the whole application proceeds in steps and each *application step* consists of progressing execution of all threads running at the time by one *thread step* (as described in Section II). After a single application step has been done, the program is re-run (its task is posted for execution at the TinyOS level). A simplified translation of a single application state is exemplified in Figure 2.

A parallel operator with n threads is translated into n switch statements (similar to the one from the example) and they are executed one after another. Nesting of the operator is realised as nesting of the switch statements. For each thread a single byte of memory is used to store its instruction pointer. A thread instruction pointer pool is used to lower the memory usage for cases where a number of parallel operators is placed in sequence (not nested). A single byte of memory is consumed in such cases. Additionally, the result of each split-phase event call is buffered and the application reads it in the next execution step.

Our compiler groups and externalises local state variables making it possible to reuse this memory, since sequential state composition results in exactly one state being executed at a time. This is in contrast with, for example, event-based frameworks where memory assigned to variables used across a number of function-callback calls cannot be easily recovered. State memory

reuse, however, has not been implemented in the current version of the compiler, neither were many other code optimisations that we envision. The future work involves experimenting with alternative thread implementations, for instance, to avoid program structuring with switch statements and isolate all atomic parts of the program into separate functions that are then chained together with event handlers with the help of a simple scheduler. With this approach applications would not need to be continuously checking for event call completion but rather rely on the OS to trigger further execution when an event call completes.

VI. Evaluation

The evaluation of a programming model is always a difficult task because its main contributions cannot be easily quantified. However, in the domain of Wireless Sensor Networks properties of the code generated by programming tools are often crucial to decide about the applicability of these tools. In this section we use the size of compiled program binary code as the main evaluation criteria for our programming model. We use TinyOS as a point of reference, firstly, because it allows for efficient and compact implementations and secondly, because it is becoming a standard WSN application development tool that many researchers are familiar with. We base our analysis on the binary code size because this metric shows the exact memory cost of a given program and we aim at showing the real applicability of our approach. There is, however, a difficulty stemming from the fact that our model offers a completely different programming paradigm than that of TinyOS and the criteria for comparing two different implementations of *the same* program might be questioned. This is why we focus on comparing implementations of some general programming patterns rather than of arbitrary programs. Also, as an additional means of illustration, we implement in our programming model an object tracking application and a set of example programs that are distributed with TinyOS.

The goal of this section is to show that applications written in our model are *realistic*, i.e., they can be run on the existing mote platforms and that compiler optimisation might further improve our results. The prototype compiler under evaluation is by no means efficient. We have implemented only basic optimisations and there is still a large space for further improvement.

In order to compare the performance of our programming model with TinyOS, we evaluate implementations of four general behavioural patterns that can be found in many WSN applications:

1) **Parallel composition** - a popular example might be sampling a number of different sensors at the same time. Often all these logically parallel operations need to finish before a move to a next application state can be made (a synchronisation point). A minimal TinyOS implementation involves invoking n different actions[1], one after another, and synchronising their callbacks on a single integer variable.
2) **Sequence of operations** - very often applications perform a sequence of steps that involve interacting with hardware and the order of these operations is strict, for example, sensor calibration needs to be done before measurements are taken or the ordering of radio message exchanges in a network protocol. A typical TinyOS implementation sequences event calls by invoking the next action in the sequence from the callback of the previous action.
3) **Splitting lengthy operations** - a technique that allows to split time-consuming computational operation into parts so they do not block the processor for too long. TinyOS does that by partitioning such operations into *tasks* which are queued in the appropriate order.
4) **Event loop** - is a pattern where an application waits for a fixed number of events (e.g., ten sensor samples) and then proceeds to another task, e.g., sending the samples out over the radio. In a typical TinyOS implementation a counter is used to track the number of action-callback invocations and the next action in the sequence of invocations is executed from the previous callback.

The experiments were performed with the TinyOS 1.1.15 package installed on the Redhat 9 operating system. The programs were compiled for the Mica2 mote family and successfully run on the motes. Figure 3 shows the resulting binary code sizes of the first three behavioural patterns implemented in TinyOS and in our model. The results for the Event Loop pattern are presented in Table I (only a single experiment was performed because increasing the number of loop repetitions involved changing only a single boolean condition). These results show the differences in the cost of program structuring according to the above patterns, since all data-related

[1]To avoid confusion, TinyOS' commands and events will be referred to in this section as actions and callbacks.

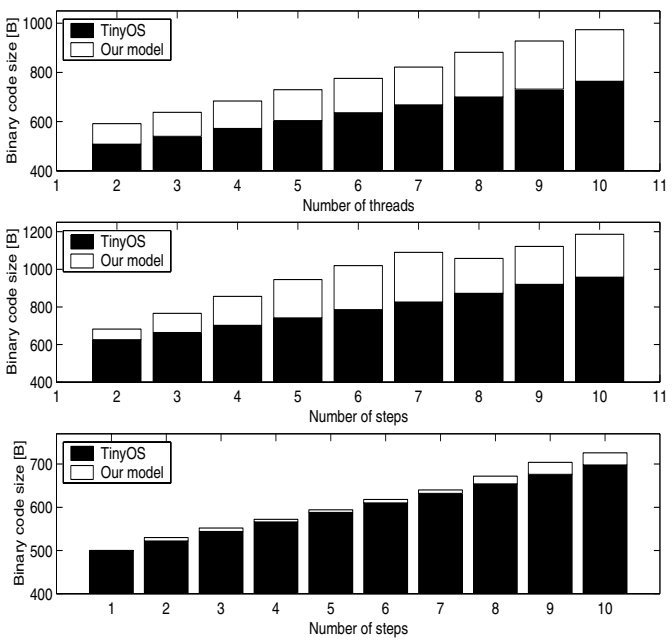

Fig. 3. Binary code size of TinyOS and our model implementations of the first three behavioural patterns.

Our Model		TinyOS	
ROM	RAM	ROM	RAM
718	27	580	23

TABLE I
BINARY CODE SIZE OF TWO IMPLEMENTATIONS OF THE EVENT LOOP BEHAVIOURAL PATTERN.

code was exactly the same in both implementations. As can be seen, adding threads increases the compiled code size by a constant factor. The current version of the compiler introduces 16 bytes of program memory overhead per thread compared to the TinyOS implementation. Additionally, a single byte of RAM memory is used for each thread's instruction counter and in the case of the first three behavioural patterns our model implementations use only a single byte of RAM memory more than their TinyOS counterparts. The results for the other two behavioural patterns also show linear growth in the code size (the small variation in the case of the second pattern was caused by some difficult to explain gcc compiler optimisations). This fact is important, because it shows that more complex programs can be tractable in terms of running them on hardware-constrained sensor nodes.

These results, however, should not be used to draw general conclusions on the performance of our programming model. They rather show certain properties of the code that can be generated from it. Our programming

Application Name	Our model		TinyOS	
	ROM	RAM	ROM	RAM
Blink	1636	50	1518	49
CntToLeds	1868	53	1678	51
Sense	3948	109	3688	102
CntToRfm	10820	452	10712	448
CntToLedsAndRfm	11052	453	10904	448
RfmToLeds	10520	399	10356	390
OscilloscopeRF	12358	526	11752	510
Surge	16924	1937	16570	1929
Tracking application	14304	557	—	—

TABLE II
CODE SIZES OF THE IMPLEMENTED EXAMPLE APPLICATIONS.

model features concepts that are not present in TinyOS such as, for example, error handling or interruption which add to the total program size when used. Also, all incoming event calls statically buffer the data which is transferred from the OS to the application. On the other hand, state management and synchronisation that need to be manually implemented in case of all non-trivial TinyOS applications reduce the differece in code size. Table II gathers the results for a number of example applications that are included in the TinyOS 1.1.15 package. To make the results comparable, the same data types and variables were used in both implementations. In the case of the Surge application we had to additionally write a TinyOS wrapper module that hid all protocol configuration details behind a single TinyOS interface because Surge uses a multi-hop routing component and its configuration lies outside of the scope of our programming model.

We have also implemented a simple tracking application in our model. The operation of the algorithm is depicted in Figure 4 and it proceeds as follows: after performing hardware initialisation (calculation of the mean signal strength) all sensor nodes begin listening for a presence of an object (moving average of the last five light sensor readings is used and the object is an artificial light source). The node whose detection threshold is exceeded becomes a local group leader claiming to be the closest node to the object. It starts sending out periodic heart-beat messages that contain the object's signal strength. Upon reception of such a message, nearby nodes enter the alert state, sample their sensors at higher frequency and send aggregates of these values to the leader. A node that senses a stronger signal than the current leader takes on his role and becomes the new leader while the previous leader enters the alert state (we did not secure the algorithm against the case

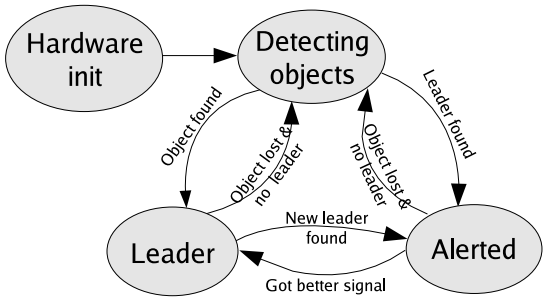

Fig. 4. An example of a simple tracking application.

of multiple leaders but the algorithm always stabilised when the object moved closer to one of the nodes).

Each state of the algorithm was directly translated to an application state in our model, Leader and Alerted states were implemented with three threads each while the DetectingObjects state declared two threads. Also, the implemented application is full-proof in the sense that all possible state transitions are accounted for (for example, when alerted nodes miss two consecutive heart-beat messages from the current leader they trigger election of a new one). The resulting code size for this application is presented in Table II. A TinyOS version of this application was not implemented to avoid bias.

VII. Discussion

As time progresses, the wide spread adoption of WSN technology will require establishing reliable programming abstractions, middleware and code libraries. This software base will act as a component base upon which new WSN applications will be built. In this paper, we show that a general high-level *application* programming model which can be layered on top of many different software components, as opposed to blurring the boundary between the application and the OS, is realistic in the domain of wireless sensor networks. At the current stage we focus mostly on reusing event-based software, firstly, because currently the most popular WSN software development platform (i.e., TinyOS), for which the majority of novel tools are written, is event-based and secondly, because writing low-level software in the event-based manner results in more compact and faster code.

Separating application code from the underlying OS or library components allows, on one hand, for code portability, but on the other, it allows the application code to be analysed independently. The second argument is even more important because the constrained resources of WSN nodes and the usually difficult maintenance of once deployed sensor networks require WSN software to be both efficient and as predictable as possible.

Applications written in our programming model, thanks to the static scheduling technique, can be translated to *sequential* code. This is particularly important property because no threading support is needed from the underlying OS. Sequential programs are much easier to debug and analyse since OS scheduling decisions need not be taken into consideration. We envision, therefore, our compiler generating code for a number of target platforms including, in particular, a simulation framework. Network-level simulation and debugging would be much more tractable since OS and the environment interfere with application's behaviour only when event calls are executed. Note that this is in contrast to, for example, simulating TinyOS applications [11] since they are often so closely entangled with the rest of the OS components that the whole OS and the application need to be simulated together.

VIII. Related Work

Our work has been initially influenced by imperative synchronous languages such as Esterel [7] that impose a rigorous mathematical execution model in which time progresses in discrete instants and events are treated as abstract signals that are *instantaneously* broadcast to all threads in their scope. Although very successful in critical embedded system development Esterel has been reported [12] to generate code that is not always linear in the size of its original specification. Another synchronous language that provides a mix of various formalisms, including state machines and object orientation is presented in [13] but, although very versatile, it is still a fully-featured synchronous language so the code efficiency question arises. In [14] the authors propose a language for embedded hardware/software system programming that introduces a process-network-like programming model and programs are translatable to a fast sequential C code. The language targets, however, hardware/software co-design and does not, therefore, support in any way the networked nature of WSN applications.

Within the Wireless Sensor Network domain, [4] proposes a state-driven model for WSN application development. Their model, however, makes "purely sequential parts of the program flow [...] tedious", which is not the case for our model. A number of event-driven programming frameworks such as [15]–[17], as well as, some more traditional approaches like [5], [18] where the classical thread-based model with independent threads

communicating with the OS kernel through blocking system calls have been proposed. The drawbacks of event-driven programming models have been presented, e.g., in [4], whereas the classical threaded model does not support the specifics of WSN application development.

A state-driven model was proposed by [19]. Their system, however, is mostly inspired by tracking applications and focuses on providing them with a suitable set of networking abstractions. In addition, execution of programs written in the model is event-based. A different WSN programming approach is presented in [20] where physical-world objects are specified in a special language and user computation is executed in an object's proximity, as determined by the object tracking middleware. The applicability of this work is confined therefore to the domain of applications related to object tracking. Another application programming paradigm that is also based on states is presented in [21]. The authors focus on an algorithm that allows to automatically notify applications about program state changes according to externally defined configurations. Their work can be regarded as complementary to ours since we focus on the application layer. It would be interesting to investigate the possibility of integration of these two approaches.

IX. Summary

We have presented a general-purpose and high-level WSN application programming model that hides the split-phase program execution scheme and provides programmers with a fine-grained concurrency model to structure event handling. In this model, applications are hierarchically composed of states and a precise application execution model is defined in such a way that static scheduling techniques can be used to generate efficient sequential programs that can also be tested and analysed on their own. A general method for integrating applications written in the programming model with event-based operating systems and middleware has been developed.

We implemented a prototype compiler for the model and we showed that our model, despite being high-level, can be compiled to efficient code that can be run on the currently available mote platforms. The future work will include optimising the compiler and experimenting with large-scale network simulation of applications written in the model.

References

[1] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," in *DMSN*, 2004.

[2] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The abstract task graph: a methodology for architecture-independent programming of networked sensor systems," in *EESR 2005*, 2005.

[3] 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 2003*. New York, NY, USA: ACM Press, 2003, pp. 1–11.

[4] O. Kasten and K. Römer, "Beyond event handlers: Programming wireless sensors with attributed state machines," in *IPSN 2005*, Los Angeles, USA, Apr. 2005, pp. 45–52.

[5] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys 2005*, 2005.

[6] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks," in *ASPLOS*, Oct. 2002.

[7] G. Berry and G. Gonthier, "The esterel synchronous programming language: design, semantics, implementation," *Sci. Comput. Program.*, vol. 19, no. 2, pp. 87–152, 1992.

[8] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic, "An entity maintenance and connection service for sensor networks," in *MobiSys 2003*, 2003.

[9] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *SenSys 2006*, Nov. 2006.

[11] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *SenSys 2003*, 2003.

[12] S. A. Edwards, "An esterel compiler for large control-dominated systems." *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 169–183, 2002.

[13] R. Budde, A. Poigné, and K.-H. Sylla, "synerjy an object-oriented synchronous language." *Electr. Notes Theor. Comput. Sci.*, vol. 153, no. 4, pp. 99–115, 2006.

[14] S. A. Edwards and O. Tardieu, "Shim: a deterministic model for heterogeneous embedded systems," in *EMSOFT*, 2005.

[15] E. Cheong, J. Liebman, J. Liu, and F. Zhao, "Tinygals: a programming model for event-driven embedded systems," in *SAC*, 2003.

[16] B. Greenstein, E. Kohler, and D. Estrin, "A sensor network application construction kit (snack)," in *SenSys 2004*, 2004.

[17] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN*, 2004.

[18] 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." *MONET*, vol. 10, no. 4, 2005.

[19] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao, "State-centric programming for sensor-actuator network systems," *IEEE Pervasive Computing*, vol. 02, no. 4, pp. 50–62, 2003.

[20] T. F. Abdelzaher, B. M. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. H. Son, J. Stankovic, R. Stoleru, and A. D. Wood, "Envirotrack: Towards an environmental computing paradigm for distributed sensor networks." in *ICDCS*, 2004, pp. 582–589.

[21] K. Römer, C. Frank, P. J. Marrón, and C. Becker, "Generic role assignment for wireless sensor networks," in *EW11: Proc. of the 11th workshop on ACM SIGOPS European workshop: beyond the PC*, 2004.