Beyond Event Handlers: Programming Wireless Sensors with Attributed State Machines

Oliver Kasten and Kay Römer
Department of Computer Science, ETH Zurich, Switzerland

***Abstract*— Event-driven programming is a popular paradigm for programming sensor nodes. It is based on the specification of actions (also known as event handlers) which are triggered by the occurrence of events. While this approach is both simple and efficient, it suffers from two important limitations. Firstly, the association of events to actions is static—there is no explicit support for adopting this association depending on the program state. Secondly, a program is split up into many distinct actions without explicit support for sharing information among these. These limitations often lead to issues with code modularity, complexity, and correctness. To tackle these issues we propose OSM, a programming model and language for sensor nodes based on finite state machines. OSM extends the event paradigm with states and transitions, such that the invocation of actions becomes a function of both the event and the program state. For removing the second limitation, OSM introduces state attributes that allow sharing of information among actions. They can be considered local variables of a state with support for automatic memory management. OSM specifications can be compiled into sequential C code that requires only minimal runtime support, resulting in efficient and compact systems.**

I. Introduction

Developing a sensor network application typically requires to program individual sensor nodes. Today, programming of individual sensor nodes is mostly performed by programmers directly. An interesting alternative to this approach is *macro programming* (e.g., [1], [2]), where programs for individual nodes are generated from high-level specifications of a sensing problem.

Currently existing frameworks for node programming are typically based on one of two abstractions: *events* and *threads*. With event-based programming, the occurrence of an event triggers the execution of an action. Actions may not block and run to completion without being interrupted by other actions. On the other hand, a multi-threaded system provides threads with independent control flow and stacks, which may be interrupted. Each thread executes a sequential program and may block to await certain conditions.

Both of these models have their advantages and drawbacks, their followers and adversaries [3]. In the context of sensor networks there is a slight bias towards event-based systems, mainly due to the existence of popular event-based programming toolkits such as TinyOS and nesC [4] with a large user base. Hence, many programmers are used to think in terms of events and actions and find it natural to structure their programs according to this paradigm.

Based on our experience with an event-based programming toolkit for sensor nodes [5], we have identified two specific drawbacks of event-based programming. As we will show, these limitations can obscure the structure of programs, hamper modularity, and lead to issues with resource efficiency. This is particularly disadvantageous in larger projects that grow over time and which involve multiple developers. We will show how the event-based programming model can be enhanced in order to alleviate these issues, thereby extending the application domain of event-based programming to cases where this would have been difficult before.

We propose the Object State Model (OSM), which is based on the notion of finite state machines (FSM). FSM concepts have been extensively used for embedded systems design (cf. [6]) and hardware/software co-design. But they have not received much consideration in the domain of sensor networks so far.

While classical FSM-based models can solve some of the issues of event-based programming, they are lacking methods for dynamic data management in resource-constrained settings. The need for dynamic data management shows up at two levels in WSNs. Firstly, a WSN program has to maintain data structures of variable size and structure (e.g., a list of network neighbors, structures for data aggregation from a variable set of sources). Secondly, a WSN program often consists of multiple sub-functions (e.g., tasking, network setup and repair, data collection and processing), each of which maintains data structures with a well-defined lifetime (e.g., a data structure for aggregation is no longer needed after the result has been computed and sent off). These data structures with variable lifetime have to be efficiently allocated to a constrained amount of memory—a problem typically not found in traditional embedded systems.

The contribution of this work is two-fold. Firstly, we consider FSM concepts for programming wireless sensor nodes for the first time. However, rather than using existing FSM-based models, we carefully select time-tested concepts and combine these with semantics that are mostly compatible with well-established event-based programming models. Secondly, we introduce a novel element called *state variables* with automatic memory management to support efficient information sharing among actions.

The remainder of the paper is structured as follows. We review the event-based programming model and two important limitations in Section II. Section III presents existing concepts that are fundamental to OSM. Section IV presents OSM in detail, in particular the semantics of OSM are discussed. Implementation aspects of OSM are examined in Section V. In Section VI we show how OSM can be used to implement a concrete application from the sensor network literature. Section VII discusses how OSM alleviates the identified limitations, also mentioning OSM's advantages and drawbacks. We present related work in Section VIII and conclude in Section IX.

II. Event-Based Programming of Sensor Nodes

A number of research projects have built wireless sensor devices and have developed programming frameworks for them. Many of these frameworks adopt the event-triggered programming model (e.g., the BTnode system software [5], Contiki [7], TinyOS and nesC [4], and Maté [8]). In this section we first review the event-based programming model that is adopted by the above frameworks. Then we show what we believe are its two major limitations.

A. The Event Model

Event-driven programs consist of actions, which are invoked in response to the occurrence of events. Typically a single event can only trigger one action but the same action may be triggered by multiple events. A so-called dispatcher manages the invocation of actions. Dispatchers often use an event queue to hold unprocessed

events. In order to allow other parts of the system to progress, actions are expected to terminate in bounded time (typically less than a few milliseconds, depending on real-time requirements and event-queue size). They run to completion in one atomic step without being interrupted by other actions. Events that occur during the execution of an action are queued for later processing. When the event queue is empty, the system goes into idle mode.

To specify a program, the programmer implements actions (typically as functions of a sequential language) and assigns those actions to events. At system start time, control is passed to the dispatcher. From the programmer's point of view, the control then remains with the system's dispatcher and is only passed to application-defined actions upon the occurrence of events. After the execution of an action, control returns to the dispatcher again.

We have made extensive use of the event-driven model in a number of projects. As part of the Smart-Its [9] and Terminodes [10] research projects we have built the BTnode wireless sensor node platform. The BTnode's system software and many of its applications are based on the event-driven model [5]. However, in working with the BTnode system we have found that the event-triggered model is hard to manage as application complexity grows. Also, [8] reports that programming TinyOS is "somewhat tricky" because of the event-driven programming model. In our work we have identified two issues with this otherwise very intuitive programming model. These issues, which we call *manual stack management* and *manual flow control*, arise because in the event-triggered model many conceptual operations need to be split among multiple actions. As a result, programmers must include additional management code, which obscures the logical structure of the application and is an additional source of error. We will detail these issues in the following section.

B. Events in Practice

Since actions must not monopolize the CPU for any significant time, operations need to be non-blocking. Therefore, at any point in the control flow where an operation needs to wait for some event to occur, the operation must be split into two parts: a non-blocking operation request and an asynchronous completion event. The completion event then triggers an action that continues the operation. As a consequence, even a seemingly simple operation can lead to *event cascades* – an action calls a non-blocking operation, which causes an event to occur, which, in turn, triggers another action. Breaking a single conceptual operation across several actions also breaks the operation's control flow and its state. This has two implications for the programmer. Firstly, as breaking operations into multiple functions effectively discards language scoping features, programmers need to manually manage the operation's stack. This is called manual stack management [11]. Secondly, programmers must guarantee that any order of events is handled appropriately in the corresponding actions. We call this manual flow control. We use the following example to clarify these issues.

```
1  int sum = 0;
2  int num = 0;
3  bool sampling_active=FALSE;
4  void init_remote_compare() {
5      sampling_active=TRUE;
6      request_remote_temp();
7      register_timeout( 5 );
8  }
9   void message_hdl( MSG msg ) {
10     if(sampling_active==FALSE) return;
11     sum = sum + msg.value;
12     num++;
13 }
14 void timeout_hdl() {
15     sampling_active=FALSE;
16     int val = read_temp();
17     int average = sum / num;
18     if( average > val ) /* ... */
19 }
```

The program above compares the local temperature to the average of temperatures sampled by neighboring sensor nodes. To do so, the sensor node running the program sends a broadcast message to request the temperature value from all neighboring sensor nodes (line 6). It then collects the remote samples (lines 9-13). A timeout is used to ensure the temporal contiguity of the sensor readings. Finally, when the timeout expires, the average remote temperature is calculated and compared to the local temperature reading (lines 16-18). As shown in the code above, this relatively simple operation needs to be split into three parts: 1) sending the request, 2) receiving the replies, and 3) processing the result after the timeout.

Manual stack management. In the example, the data variables `sum` and `num` are accessed within two actions. In a traditional, purely procedural program, automatic (i.e., local) variables serve the purpose of keeping an operation's local data. They are automatically allocated on the local runtime stack upon entering a function and are released on its exit. However, automatic variables cannot be used for event cascades. Since the local stack is unrolled after every execution of an action, the state does not persist over the duration of the whole operation. Instead, programmers must manually program the task's state. They can do so either using global variables or by programming a state structure stored on the heap. Both approaches have drawbacks. The first approach permanently locks up memory, also when the operation is not running. The second approach requires manual memory management, which is error-prone. It also requires system support for dynamic memory management, which is rarely found in embedded systems.

Manual Flow Control. Depending on the node's state and history of events, a program may need to behave quite differently in reaction to a certain event. In our previous program, for example, replies from remote sensors should only be regarded until the timeout expires and the timeout action is invoked. After the timeout event, no more changes to `sum` and `num` should be made (even though this is not critical in the concrete example). To achieve this behavior, the timeout action needs to communicate with the radio-message action so that no more replies should be regarded. We call this manual flow control. In the example, code for manual flow control is highlighted. The code introduces a global boolean flag `sampling_active` (line 3), which is used in all three functions (lines 5, 10, and 15). Coding the flow control manually requires operating on state that is shared between multiple functions, such as the flag `sampling_active` in our example. Again, this state needs to be managed manually.

In the small toy example we have just presented, manual management of flow control and of the stack seems to be a minor annoyance rather than a hard problem. However, even in such a simple program, a significant part of the code (4 lines) is dedicated to manual flow control. A similar example in nesC can be found in [4]. As application complexity grows, these issues become more and more difficult to handle. In fact, in our applications that implement complex networking protocols (e.g., a Bluetooth stack), significant parts of the code are dedicated to manual flow control and stack management. The code is characterized by a multitude of global variables, and by additional code in actions to manage program flow. The excess code obscures the logical structure of the application and is an additional source of error.

III. Basic Concepts

To attack the problems described in the last section we propose OSM. The semantics and representation (i.e., language) of OSM are based on finite state machines and concepts introduced by other state-based models, such as Statecharts [12]. Hence, before discussing OSM in the next section, we briefly review concepts related to finite state machines in this section.

Finite state machines are based on the concepts of *states*, *events*, and *transitions*. A FSM consists of a set of states and a set of transitions, each of which is a directed edge between two states, originating form the source state and directed towards the target state. Transitions specify how the machine can proceed form one state to another. Each transition has an associated event. The transition is taken (it "fires") when the machine is in the transition's source state and its associated event occurs. FSMs can be thought of as directed, possibly cyclic graphs, with nodes denoting states and edges denoting transitions. As in event-based programming, *actions* specify computational (re)actions. Conceptually, actions are associated with transitions or states. For the definition (i.e., implementation) of actions, most models rely on a host language, such as C.

Even though a program could be fully specified with those four concepts explained above, [12] argues that the representation of complex programs specified in this naive fashion suffers from state explosion. To alleviate this problem, [12] introduces *hierarchy* and *concurrency* to finite state machines. A state can subsume an entire state machine, whose states are called *substates* of the composing state. The composing state is called *superstate*. This mechanism can be applied recursively: superstates may themselves be substates of a higher-level superstate. The superstate can be seen as an abstraction of the contained state machine (bottom-up view). The state machine contained in the superstate can also be seen as a refinement of the superstate (top-down view). Though a superstate may contain an entire hierarchy of state machines, it can be used like a regular (i.e., uncomposed) state in any state machine. Finally, two or more state machines can be run in parallel, yet communicate through events.

A concept which has received little attention so far are *state variables*, which hold information that is *local to a state or state hierarchy*. Instead, some models rely entirely on their host language to provide variables. In most other models that encompass variables, the variable scope is global to an entire FSM.

IV. The Object State Model

OSM is based on the conceptual elements presented in the previous section. Through the explicit notion of states, the association of events to actions is no longer static. Rather, the invocation of actions becomes a function of both the event and the current program machine state. State variables allow information sharing among a set of actions that collectively implement a certain system function. Their scope is limited to a state (and its substates), thus allowing to reclaim memory upon leaving the state. Modular programming is supported by embedding existing state machine definitions via hierarchical composition. Parallel state machines can handle events originating from independent sources. For example, independently tracked targets could be handled by parallel state machines. While events are typically triggered by real-world phenomena (e.g., a tracked object appears or disappears), events may also be emitted by the actions of a state machine to support loosely-coupled cooperation of parallel state machines.

Besides *modeling* a program in terms of states, transitions, actions, and events, OSM also allows the *execution* of a program specification on a sensor node. This requires the definition of concrete semantics of

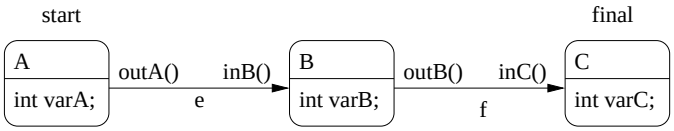

Fig. 1. A sample state machine.

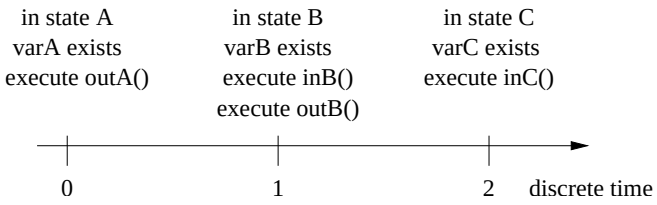

Fig. 2. Discrete time.

OSM specifications. In the following subsections we introduce OSM and informally define an execution model.

A. State Machines in OSM

Fig. 1 illustrates a state machine in OSM. We will use a graphical notation here for clarity. In practice, OSM state machines are specified using a textual language (cf. Section V).

The sample state machine consists of three states A, B, and C. A is the initial state where execution begins and C is the final state. To each of these states, an integer variable $varA$, $varB$, and $varC$ is attached, respectively. Transitions exist between states A and B (triggered by the occurrence of event e) and between states B and C (triggered by the occurrence of event f). In OSM, events may also carry additional *parameters* (e.g., sensor values). Each event parameter is assigned a type and a name by the programmer. Actions can refer to the value of an event parameter by the parameter's name. Let us consider the transition between A and B in more detail, which is tagged with two actions $outA()$ and $inB()$. When the transition occurs, first $outA()$ is executed and can access e and variables of state A. Then, $inB()$ is executed and can access e and variables of state B. If source and target states of a transition share a common superstate, its variables can be accessed in both actions. One or both actions, as well as their parameters could also be omitted.

Typically, a transition is triggered by the event that the transition is labeled with. Additionally OSM allows to *guard* transitions by a predicate over event parameters as well as over the variables of the source state. The transition then only fires if the predicate holds (i.e., evaluates to true) on the occurrence of the trigger event.

B. Time

Finite state machines imply a discrete time model where time progresses from t to $t+1$ when a state machine makes a transition from one state to another. Consider Fig. 2 for an example execution of the state machine from Fig. 1. At time $t = 0$, the state machine is in state A and the variable $varA$ exists. Then, the occurrence of event e triggers a state transition. The action $outA()$ is still performed at $t = 0$, then time progresses to $t = 1$. The state machine has then moved to state B, variable $varB$ exists and the action $inB()$ is executed. When event f occurs, $outB()$ is performed in state B, before time progresses to $t = 2$. The state machine has then moved to state C, variable $varC$ exists and the action $inC()$ is executed.

Approaching time from the perspective of a sensor network that is embedded into the physical world, a real-time model seems more appropriate. For example, sensor events can occur at virtually any

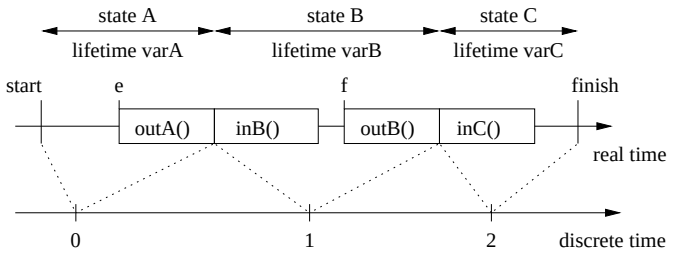

Fig. 3. Mapping real-time to discrete time.

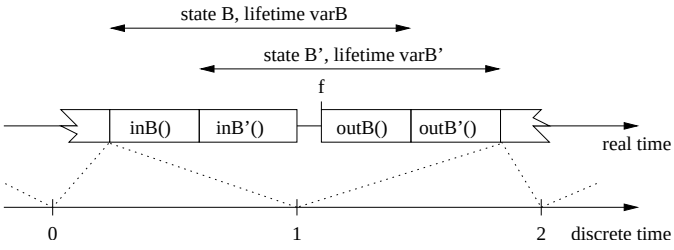

Fig. 4. Possible mapping of real-time to discrete time for parallel machines.

point in time, and actions and transitions require a finite amount of real-time to execute. Hence, it becomes also important in which order actions are executed. Essentially, we are faced with the issue of interfacing the discrete time model of state machines with the real-time model of sensor networks.

Inspired by traditional event-based programming models, actions are considered atomic entities that run to completion without being interrupted by the execution of any other action. Fig. 3 illustrates the mapping of real-time to discrete time, the respective states and variable lifetimes.

C. Parallel Composition

An important element of OSM is support for parallel state machines. Two or more such state machines are synchronized in the sense that state transitions in multiple state machines can occur concurrently in the discrete time model. In the real-time model, however, the state transitions and associated actions are performed sequentially. Let us consider an example of two state machines: the one from Fig. 1, and a copy of this machine where the state and variable names are primed (but not the event names). Fig. 4 shows the mapping of real-time to discrete time at discrete time $t = 1$. After discrete time has progressed to $t = 1$ on the occurrence of e, all "in" actions are executed in any order. When event f arrives, all "out" actions are performed in any order.

D. Hierarchical Composition

Another important abstraction supported by OSM are hierarchies, where a single state (i.e., a superstate) is further refined by embedding another state machine. The lifetime of the embedded state machine begins at its initial state when the superstate is entered. The embedded state machine terminates when it assumes its final state. Transitions having a superstate as their source state may be tagged as *normal termination* transitions, which can only fire after the inner state machine has terminated. Regular transitions, however, terminate the inner state machine upon the occurrence of the transition's event, which can happen at any (discrete) time.

If a superstate contains variables, the scope of these variables covers all substates, also recursively. In other words, an action can access the variables of all its superstates.

E. Events, Queues, and State Transitions

Since the execution of actions consumes a non-zero amount of real time, events may arrive in the system while some action is currently being executed. Those events cannot be handled immediately. Therefore, arriving events are always inserted into a FIFO queue.

In the discrete time model, events can occur concurrently. For example, when parallel state machines emit events in "out" actions (when simultaneously progressing from time $t = 1$ to $t = 2$, cf. Fig. 4), the emitted events have no canonical order. The event queue should preserve such sets of concurrent events, rather than imposing an artificial total ordering on concurrent events by inserting them one after another into the queue. For this purpose, the event queue operates on sets of concurrent events rather than on individual events. The *enqueue* operation takes a set of concurrent events as a parameter and appends this set (as a whole) to the end of the queue. The *dequeue* operation removes the set of concurrent events from the queue that has been inserted earliest. Whenever the system is ready to make a transition, it uses the dequeue operation to determine the one or more events to trigger transitions.

The execution semantics of a set of parallel state machines can then be described as follows. After each discrete time step, the earliest set of concurrent events is dequeued unless the queue is empty. Each of the parallel state machines considers the set of dequeued concurrent events to derive a set of possible transitions. A single event may also trigger transitions in multiple state machines. If multiple concurrent events are available, multiple transitions could be fired in a single state machine. To resolve such ambiguous cases, priorities can be assigned to transitions, such that the transition with the highest priority is triggered in each state machine. Actions are executed as described in Section IV-B. The dequeued set of concurrent events is then dropped. All events emitted by actions during the same discrete time step are enqueued as a single set of concurrent events.

F. State Variables

With respect to state variables, OSM supports both primitive data types (e.g., integer types, characters) and structured data types (e.g., arrays, strings, records) in the style of existing typed programming languages such as C.

The scope of the variables of a state S extends to entry and exit actions that are associated with S and to all actions of state machines that are recursively embedded into S via hierarchical composition. With respect to scoping, there is a special case for self transitions that enter and leave the same state. Here, the variables of the affected state and their values are retained during the transition, rather then deleting the variables and creating new instances.

Note that a single (uncomposed) state machine can only assume one state at a time. Hence, only the variables of this current state are active. Variables of different states can often be allocated to overlapping memory regions in order to optimize memory consumption (cf. Section V).

By embedding a set of parallel machines into a superstate, actions in different parallel state machines may access a variable of the superstate concurrently at the same discrete time. This is no problem as long as there is at most one concurrent write access. If there are multiple concurrent write accesses, these accesses may have to be synchronized in some way. Due to the run-to-completion semantics of actions, a single write access will always completely execute before the next write access can occur. However, the order in which write accesses are executed (in the real-time model) is arbitrary. Write synchronization is up to the programmer.

V. Implementation

In this section we examine the missing pieces that are needed to turn the concepts from the previous section into a concrete system.

A. OSM Specification Language

As opposed to most state machine notations, which are graphical, OSM specifications use a textual language. The following subsections present important elements of this language.

1) States: The prime construct of the OSM language is a state. The definition of a state includes the definition of its transitions, its actions, and its variables. A transition may be defined in its source state (outgoing transition) or its target state (incoming transition) or both. A transition always has a trigger event, a source state and a target state. Outgoing transitions can have an exit action, incoming transitions may have an entry action. If a transition is to trigger both an exit and an entry action, it must be declared in both states, as an incoming and as an outgoing transition, respectively. Both declarations denote the same transition, if source state, target state, and trigger event are equal. If only one action is required, the transition may be defined only in the respective state. If a transition triggers no action it can be declared in either state.

Below is an example of a state A. It has three transitions, one incoming and two outgoing transitions. The incoming transition (line 3) originating in state B triggers the entry action $inA()$. The two outgoing transitions (lines 4 and 5) lead to states B and C and trigger the outgoing actions $out1()$ and $out2()$, respectively. Transitions have priorities according to their order of declaration, with the first transition having the highest priority. For example, if two concurrent events e and f occur in state C, both outgoing transitions could fire. However, only the transition to B is taken, since it is declared before the transition to C.

2) Variables: Variables can be defined only in the scope of a state. Variables are typed and have a name, by which they can be referenced in actions anywhere in their scope. In the code below, state A defines the variable i (of type integer).

```
1 state A {
2   int i;
3   B -> e / inA();
4   e / out1() -> B;
5   f / out2() -> C
6 }
```

3) Event Parameters and Variables in Actions: Actions may have parameters. Event parameters can be accessed from an action by specifying the event name as a parameter of the action. Additionally, actions may have any visible state variables as their parameters. Action parameters must be declared explicitly. For example, in the code fragment below, $outA()$ has three parameters: the value of event e, and the state variables *index* and *buffer*.

For the definition (i.e., implementation) of actions, OSM relies on a host language such as C. Actions map to functions of the same name in the host language. For each host language, there is a language mapping, which defines how actions and their parameters map to function signatures. Programmers must then implement those functions.

4) Grouping: To group a set of self-contained states into a state machine, the set can be enclosed with brackets and labeled with the `machine` keyword. Optionally, the group can be given a name. The grouped states must only contain transitions to states in this group. While one of the states must be marked as the *initial* state, any number of states (including zero) may be marked *final*.

The machine may not define its own variables (as variables may only be defined in a state) but the contained states may refer to variables that are defined in a superstate. These variables are said to be *external* of that machine. If desired, the external variable interface may be declared explicitly, but if it is declared, external variables must be marked `extern`, as in the example below.

```
1 machine AB {
2   extern int index;
3   extern char buffer[256];
4   initial state A { e/outA(e,index,buffer)->B;}
5   final state B { f / outA(f, index) -> A; }
6 }
```

5) Hierarchical and Parallel Composition: A state may contain an entire state machine. State $S1$ in the code on the left below, for example, contains a state group AB (which shall be defined as above). Note that $S1$ defines the external variables declared in AB above. Parallel state machines can be defined by concatenating two or more state machines with "`||`", as in the code on the right below.

```
1 state S1 {                state S2 {
2   int index;                machine CD {...}
3   char buffer[256];         ||
4   machine AB {...}          machine EF {...}
5 }                         }
```

6) Modularity through Machine Incarnation: OSM's support for modularity relies on hierarchical embedding of state machines. A programmer may instantiate a state machine, which is defined elsewhere in the program, to embed it into a superstate. A machine may be instantiated within a state using the `incarnate` keyword followed by the machine's name. Just like regular, hierarchically composed state machines, embedded state machines may access the variables of their superstates. The external variables of the incarnation are bound to variables of the embedding superstate, which must have the same name and type.

However, since the embedded state machine and the superstate may be specified by different developers, a naming convention for events and variables would be needed. In order to relieve programmers from such conventions, OSM supports name substitutions for events and variables in the incarnation of modules. This allows to integrate multiple machines that were developed independently.

The code fragment below shows the incarnation of state machine AB (which again shall be defined as above), renaming AB's external variables *index* and *buffer* to i and *buf* of $S3$, respectively. Renaming of events is done analogously (not shown in the example). Recursive instantiation is not allowed.

```
1 machine AB {...}
2
3 state S3 {
4   int i;
5   char buf[256];
6   incarnate AB( index/ i; buffer/ buf );
7 }
```

B. OSM Language Mapping

An OSM language mapping defines how OSM specifications are translated to a host language (which is also used for specifying actions). Such a mapping is implemented by an OSM compiler. We have developed a prototypical version of an OSM compiler that uses C as a host language. Hence, we discuss below how OSM can be mapped to C.

1) Mapping Control Structures: Our current version of the OSM compiler maps OSM control structures to an intermediate representation in the imperative Esterel language [13]. From this representation sequential C code is generated by an Esterel compiler. Esterel is a synchronous language (cf. [14]) for the specification of reactive systems. As such, it is well suited for the implementation of state

machines. Our mapping from state-based OSM to imperative Esterel is inspired by SyncCharts [15], [16]. Esterel produces lightweight and system-independent C-code, which requires no support for multitasking. Rather, an OSM program compiles into one principal state-transition function. This function can be seen as a single event handler for all events. It accepts the dequeued set of concurrent events (cf. Section IV-E) as a parameter. Invoking the state transition function performs a discrete time step of the machine.

In that step, the state transition function first invokes all exit actions of the current state, then performs the state transition, and finally invokes all entry actions of the newly assumed state. OSM relies on the underlying operating system to provide an event queue which can hold event sets, and drivers that generate events (e.g., timeout, a message has arrived, a tilt switch has been triggered). Currently, OSM generated code runs on BTnodes, for which we have modified the existing event-driven system software to support event sets.

2) Mapping Variables: A language mapping must also define how state variables are allocated in memory, yielding an in-memory representation R of all state variables of an OSM specification. The main goal is to minimize the memory footprint. The allocation can be defined recursively as follows. A single variable is mapped to a memory region R_{var} that is just big enough to hold the variable. A state results in a representation R_{state}, which is defined as a sequential record (e.g., a C `struct`) of the representations R_{var_i} of all variables i and the representation $R_{machine}$ of an optional embedded state machine. The representation $R_{machine}$ for a whole state machine is defined as the union (e.g., a C `union`) of the representations R_{state_i} of all states i, since each machine can be in only one state at a time. Hence, different states of the same machine can reuse memory. The representation R_{par} of a set of parallel machines is defined as a sequential record of the representations $R_{machine_i}$ of all machines i. Consider the following state machine:

```
state C {
  int c;
  state A { int a1, a2; e / outA(c) -> B; }
  state B { int b1, b2; f / -> A; }
} || state D {
  int d;
}
```

which results in the following memory layout in C:

```
struct parCD {
  union machineC {
    struct stateC {
      int c;
      union machineAB {
        struct stateA { int a1, a2; } _stateA;
        struct stateB { int b1, b2; } _stateB;
      } _machineAB;
    } _stateC;
  } _machineC;
  union machineD {
    struct stateD {
      int d;
    } _stateD;
  } _machineD;
} _parCD;
```

If an `int` consists of 4 bytes, then the above structure requires 16 bytes. If all variables were global, 24 bytes would be needed. Note that the size of the required memory and locations of the variables in memory are known at compile time. No dynamic allocations are performed at runtime. The action `outA()` would be mapped to the C function call, where a pointer is passed to allow modification of the state variable.

```
outA(&_parCD._machineC._stateC.c);
```

VI. EnviroTrack: A Case for OSM

In this section we illustrate the practical feasibility of OSM by sketching how EnviroTrack [17] could be implemented with OSM. EnviroTrack is a framework that supports tracking of mobile targets with a sensor network. With EnviroTrack, a sensor node can be in one of four major states: `free`, `follower`, `member`, and `leader`. A member is detecting the proximity of the target with its sensors, a follower is a network neighbor of a member that does not detect the target itself, a free node does not detect the target and is not a follower, and a leader is a member that has been elected out of all members. All members send their locations to the leader, where these locations are aggregated to derive a location estimate of the target. Members frequently broadcast heartbeat messages so that free nodes can detect whether they should become followers. A follower sets up a timeout and becomes a free node if the timeout expires.

The OSM specification of EnviroTrack consists of three parallel state machines: `Elector`, `Detector`, and `Tracker`. The `Elector` accepts a `start_leader_ev` event to trigger leader election among all members and may emit a `elected_ev` to indicate that the node has been elected as the leader. The `Detector` state machine uses sensors to generate a `sense_ev` when the object is detected and a `lost_ev` event when the object is no longer detected. We will only show the OSM code for the `Tracker` state machine, which represents the core of EnviroTrack.

```
1  initial state FREE {
2    heartbeat_ev / -> FOLLOWER;
3    sense_ev / -> MEMBER;
4  }
5
6  state FOLLOWER {
7    sense_ev / -> MEMBER;
8    entity_timeout_ev / -> FREE;
9    heartbeat_ev / reset_entity_to() -> self;
10   FREE -> heartbeat_ev / set_entity_to();
11 }
12
13 state MEMBER {
14   FREE -> sense_ev / set_heartbeat_to(),
15                      start_leader_election();
16   FOLLOW -> sense_ev / set_heartbeat_to();
17   heartbeat_ev / send_position(),
18                  send_heartbeat() -> self;
19   lost_ev / -> FOLLOWER;
20   elected_ev / -> LEADER;
21 }
22
23 state LEADER {
24   position_type pos;
25
26   lost_ev / start_leader_election() -> FOLLOWER;
27   position_ev / aggregate_pos(pos, position_ev)
28                 -> self;
29 }
```

Lines 1-4: A free node becomes a follower if it receives a heartbeat. A free node becomes a member if the target is sensed.

Lines 6-11: A follower becomes a member if the target is sensed. A follower becomes a free node if the timeout expired before a heartbeat was received. If a heartbeat is received, the timeout is reset. If a free node became a follower, the timeout is initialized.

Lines 13-21: If a free node became a member, the heartbeat timer is set up and leader election is initiated. If a follower became a member, the heartbeat timer is set up. If the heartbeat timer expires, the current location is sent to the leader and a heartbeat message is sent to the followers. If the target is no longer detected, the member becomes a follower. If the member is elected, it becomes the leader.

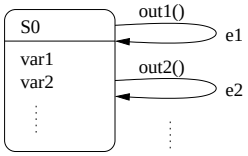

Fig. 5. A traditional event-based program specified in OSM.

Lines 23-29: A leader has a state variable to hold the average of the positions of the members. If the target is no longer detected, the leader becomes a follower and leader election is triggered. If the leader receives a position update from a member, the position is aggregated with the average position.

VII. Discussion

A traditional event-based program can be formulated in OSM as depicted in Fig. 5. There is a single state $S0$, which has attached all global variables $vari$ of the event-based program. For each possible event ej there is a self transition with an associated action $outj()$, which has access to ej and to all state variables of $S0$. Hence, OSM can be considered a natural extension of event-based programming.

One notable implication of this mapping is that OSM programs are typically more memory efficient than traditional event-based program. In Fig. 5 (and also in an actual implementation of an event-based program) all state variables are active all of the time. Hence, the memory consumption equals the *sum* of the memory footprints of all these variables. In OSM specifications with multiple states, the same set of variables is typically distributed over multiple states. Since only one state of a state machine can be active at a time, the memory consumption equals the *maximum* of the memory footprints among all states.

Another important observation is that OSM supports two orthogonal ways to deal with program state: explicit machine states and state variables. In traditional event-based programming, all program state is expressed via global variables. In pure finite state machines, all program state is expressed as distinct states of the FSM. With OSM, programmers can select a suitable point between those two extremes by using explicit machine states only where this seems appropriate. In particular, a programmer can start with an existing event-based program, "translate" it to OSM as in Fig. 5, and gradually extend it with more states.

OSM alleviates the limitations of event-based programming (cf. Section II) by (i) supporting information sharing between actions by means of state variables, and by (ii) providing for a flexible association of events to actions by making transitions and actions dependent on the program state. These features have a number of implications with respect to code structure, modularity, and efficiency:

Code structure and modularity. Variables can be made as local as possible, which helps to isolate code modules. Explicit machine states can eliminate code that crosscuts multiple actions, which also helps to isolate code modules. OSM provides abstractions to encapsulate such modules and to give them an interface in terms of events and parameters.

Efficiency. As discussed above, the variables of an OSM specification with multiple states typically consume less memory than an equivalent traditional event-based program. In Section V we mentioned that an OSM specification can be compiled into a single "event handler" function – effectively making OSM code as efficient as traditional event-based programs. The only runtime support required to execute an OSM specification is the event queue and drivers that generate events.

There are also some limitations which OSM inherits from event-based programming. Firstly, actions must be non-blocking and there are no guarantees with respect to real-time behavior. Also, if actions generate events, the order of events in the queue (and hence the system behavior) may depend on the execution speed of actions.

Purely sequential parts of the program flow (i.e, linear sequences of events and actions without branches) are tedious to program in OSM because they have to be explicitly modeled as sequences of states, pairwise connected by a single transition. In such cases wait-operations, as typically provided by multi-threaded programming models, might be more natural. Likewise, reactions to composite events (i.e., meta-events made up of multiple events, for example, "events $e1$, $e2$, and $e3$ in any order") cannot be specified concisely. Instead, all intermediate states have to be modeled explicitly.

VIII. Related Work

OSM draws directly from the concepts found in specification techniques for control-oriented embedded systems, such as finite state machines, Statecharts [12], and its descendant SyncCharts [15], [16]. From Statecharts, OSM borrows the concept of hierarchical and parallel composition of state machines as well as the concept of broadcast communication of events within the state machine. From SyncCharts we adopted the concept of concurrent events.

Variables are typically not fundamental entities in control-oriented FSM models. Rather, these models rely on their host languages for handling data. Models that focus both on the transformative domain (data processing, stream processing) and the control-oriented domain, typically include variables as intrinsic entities. Finite State Machines with Datapath (FSMD) [18] introduced variables to the FSM model in order to reduce the number of states that have to be declared explicitly. Like OSM, this model allows programmers to choose to specify program state explicitly (with machine states) or implicitly with variables. FSMD are flat, that is, they do not support hierarchy and concurrency, and variables have global scope and lifetime. Contrary, variables in OSM are bound to a state hierarchy.

SpecCharts [19] is a state-machine extension to VHDL. SpecCharts supports hierarchy and concurrency. As in OSM, variables are declared within states; the scope of a variable then is the state it has been declared in and any descendants. SpecChart programs are translated into plain VHDL, which can then be subjected to simulation, verification, and hardware synthesis. The main difference to OSM is, that computations in SpecCharts are not attached to transitions but rather to leaf (i.e., uncomposed) states. In analogy to Moore and Mealy machines, we believe that reactive systems can be specified more concisely in OSM. Though both models are computationally equivalent, converting a Mealy machine (where output functions are associated with transitions) to a Moore machine (output associated with states) generally increases the size of the machine, that is, the number of states and transitions. The reverse process leads to fewer states. Finally, in contrast to SpecCharts, OSM allows to access the values of events in computational actions. A valued event is visible in the scope of both the source and the target state of a transition (in "out" and "in" actions, respectively). This is an important aspect of OSM.

Another model for the design of control and data-oriented embedded systems are *communicating FSMs*, which conceptually separate data and control flow. In this model, a system is specified as a finite set of FSMs and data channels between pairs of machines. FSM execute independently and concurrently but communicate over typed channels. Variables are local to a single machine, but global to the states of that machine. Values communicated can be assigned

to variables of the receiving machine. There are several variations of that basic model. For example, in Communicating Real-Time State Machines (CRSM) [20] communication is synchronous and unidirectional. Individual FSMs are flat. Co-design Finite State Machines (CFSM) [21] communicate asynchronously via single element buffers, but FSM may be composed hierarchically. In contrast to communicating FSMs, concurrent state machines in OSM communicate through events or shared variables.

OSM, like Statecharts, is implemented on top of Esterel [13]. We considered using Esterel directly for the specification of control flow in OSM. However, as an imperative language, Esterel does not support the semantics of FSM directly. We believe that FSM are a very natural and powerful means to model WSN applications. Moreover, specifications in Esterel are generally larger (up to 5 times) compared to OSM.

A number of frameworks for programming individual sensor nodes have been proposed. With one exception, all frameworks fall into one of two basic categories: event-based systems (such as NesC [4], the BTnode system software [5], and Maté [8]) and multi-threaded systems (SensorWare [1] and Mantis [22]). Applications built on either model have an implicit notion of program state. In contrast, in OSM program state can be modeled explicitly.

Contiki [7] is an operating system built around an event-driven kernel but also supports preemptive multithreading. Typical Contiki applications are built on events. However, individual, long-running operations, such as cryptographic operations, may be specified in a separate thread. OSM only allows to specify actions of bounded time.

With event-based systems, OSM shares the discrete time model where the application is always in one discrete state. One exception here is TinyOS/NesC. Similar to our approach, it provides asynchronous events as a basic programming abstraction. However, NesC has events on two levels: events on the lower-level are modeled as interrupts and the actions they trigger are interrupt service routines (ISR). The event queue on this level is implemented in hardware. Higher-level events are events in the common sense; they have a FIFO queue. In the NesC language, ISRs can interrupt other ISRs and regular (i.e., higher-level) actions. Actions run to completion only with respect to other actions. Therefore, a NesC application may be interrupted when making the transition. This model can lead to subtle race conditions and inconsistencies.

IX. Conclusion

Event-based programming is a popular paradigm in the domain of sensor networks that has been adopted by a number of programming frameworks. We have illustrated two important shortcomings of the event model, namely manual stack management and manual flow control. We showed that these can lead to issues with modularity, resource efficiency, and correctness.

To alleviate these problems, we have proposed OSM: a model and language for programming sensor nodes with attributed state machines. OSM is based on abstractions that have been successfully used for programming embedded systems in the past. The concept of state variables is introduced to support efficient information sharing among actions. OSM also provides semantics that are compatible with existing event-based systems, thus easing the transition for programmers that are familiar with event-based programming.

We have shown the practical feasibility by sketching an OSM-based implementation of EnviroTrack, a system for tracking mobile objects with sensor networks. A prototypical OSM compiler has been implemented with C as a host language, resulting in efficient programs that require only minimal runtime support. First tests have been performed on the BTnode sensor node.

OSM might be a step towards extending the application domain of event-based programming to larger and more complex systems. Based on the existing large body of work on verification of finite state machines, we hope to be able to augment OSM with verification facilities to detect faulty specifications.

Acknowledgement

We thank Lothar Thiele, our shepherd Margaret Martonosi, and the anonymous reviewers for their helpful feedback. This work was supported by NCCR-MICS, a center supported by the Swiss National Science Foundation under grant no. 5005-67322.

References

[1] A. Boulis and M. B. Srivastava, "Design and Implementation of a Framework for Efficient and Programmable Sensor Networks," in *MobiSys*, San Francisco, USA, May 2003.

[2] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks," in *OSDI 2002*, Boston, USA, Dec. 2002.

[3] R. v. Behren, J. Condit, and E. Brewer, "Why Events Are A Bad Idea (for high-concurrency servers)," in *HotOS IX*, Lihue, USA, May 2003.

[4] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *SIGPLAN*, 2003.

[5] J. Beutel, O. Kasten, F. Mattern, K. Römer, F. Siegemund, and Lothar Thiele, "Prototyping Wireless Sensor Network Applications with BTnodes," in *EWSN*, Berlin, Germany, Jan. 2004, pp. 323–338.

[6] A. C. Shaw, *Real-Time Systems and Software*, John Wiley, 2001.

[7] A. Dunkels, Björn Grönvall, and Thiemo Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *EmNetS-I*, Tampa, USA, Nov. 2004.

[8] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. 5, pp. 85–95, Dec. 2002.

[9] L. E. Holmquist, F. Mattern, B. Schiele, P. Alahuhta, M. Beigl, and H.-W. Gellersen, "Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts," in *Ubicomp*, Atlanta, USA, Sept. 2001, pp. 116–122.

[10] J. P. Hubaux, Th. Gross, J. Y. Le Boudec, and M. Vetterli, "Towards self-organized mobile ad hoc networks: the Terminodes project," *IEEE Communications Magazine*, vol. 31, no. 1, pp. 118–124, 2001.

[11] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative Task Management Without Manual Stack Management," in *USENIX Annual Technical Conference*, 2002.

[12] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 6 1987.

[13] F. Boussinot and R. de Simone, "The ESTEREL Language," *Proc. of the IEEE*, vol. 79, no. 9, pp. 1293–1304, Sept. 1991.

[14] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone, "The Synchronous Languages 12 Years Later," *Proc. of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[15] C. André, "Representation and analysis of reactive behaviors: A synchronous approach," in *Proc. CESA '96*, Lille, France, July 1996.

[16] C. André, "Syncharts: a visual representation of reactive behaviors," Tech. Rep., I3S, Sophia-Antipolis, France, Oct. 1995.

[17] T. Abdelzaher et al., "EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks," in *ICDCS*, Tokyo, Japan, Mar. 2004.

[18] Daniel D. Gajski and Loganath Ramachandran, "Introduction to high-level synthesis," *IEEE Des. Test*, vol. 11, no. 4, pp. 44–54, Oct. 1994.

[19] F. Vahid, S. Narayan, and D. D. Gajski, "SpecCharts: A VHDL Front-End for Embedded Systems," *IEEE Trans. on CAD*, vol. 14, no. 6, pp. 694–706, June 1995.

[20] Alan C. Shaw, "Communicating real-time state machines," *IEEE Trans. Softw. Eng.*, vol. 18, no. 9, pp. 805–816, Sept. 1992.

[21] Felice Balarin et al., *Hardware-software co-design of embedded systems: the POLIS approach*, Kluwer Academic Publishers, 1997.

[22] H. Abrach et al., "MANTIS: system support for multimodAl NeTworks of in-situ sensors," in *WSNA*, San Diego, CA, USA, 2003, pp. 50–59.