Stream-based Macro-programming of Wireless Sensor, Actuator Network Applications with SOSNA

Marcin Karpiński and Vinny Cahill
Distributed Systems Group
Trinity College Dublin
{karpinsm, vjcahill}@cs.tcd.ie

ABSTRACT

Wireless sensor, actuator networks distinguish themselves from wireless sensor networks by the need to coordinate actuators' actions, real-time constraints on communication and the frequently feedback-based nature of computation performed in the network. In this paper we propose a functional macro-programming language, SOSNA, that employs the stream programming paradigm to concisely specify data transformations in the network so that wireless sensor actuator network (WSAN) application developers can focus on higher-level control-oriented problems rather than on designing the way in which communication is organised in the network. SOSNA accommodates a broad class of WSAN coordination models, supports mobility and provides a means of employing feedback for distributed state maintenance. Program execution proceeds in rounds providing real-time guarantees on actuator decision making and synchronisation. In addition, static program semantics permit nodes to switch their radios off to conserve energy.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed applications; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Concurrent, distributed, and parallel languages; Data-flow languages

General Terms

Design, Languages

1. INTRODUCTION

Wireless sensor, actuator[1] networks are an emerging research discipline in which the problem of controlling a physical phenomenon is addressed using spatially distributed networks of devices that possess communication, sensing and/or actuation capabilities. Typically, sensor nodes (or sensors, for brevity) are considered to be highly resource-constrained while actuator nodes (or simply actuators) may have less stringent resource budgets [2]. Computation in the network is driven by data coming from sensor nodes and the control task has to be realised by means of actuator coordination.

WSANs are not just an extension of wireless sensor networks (WSNs) with actuators. The existence of actuators in the system shifts the focus from sensing to control posing new requirements on the methods employed. In this view, a WSAN is a kind of control system in which sensing, actuation and decision making are distributed while resource-constrained system components communicate with each other, in particular, over unreliable communication channels what makes already established tools and approaches from the control systems domain unsuitable [19].

There is a considerable variety in the WSAN design space. The network may be homogeneous with all nodes being equipped with sensing and actuation modalities; sensing and actuation may be performed by different network nodes; or different network nodes may possess different sensing and actuation capabilities. Network (sensor and actuator) nodes may be static, some of them may be mobile or, in the extreme case, all of them may be able to move. Network topology may be hierarchical with actuators being capable of long-range communication and sensor nodes transmitting data using low-power unreliable links directly to actuators in their vicinity; or communication in the network may be multi-hop with all network nodes using the same communication medium.

Because a WSAN is essentially a kind of control system, real time plays an important role in its operation and often manifests itself in the need for synchronisation of actuation. This fact implies that all communication in the network, as well as the process of actuator coordination have certain timing requirements and the degree of timeliness may depend on a particular application. Also, as explained in [2], there are many ways in which communication between sensors and actuators can be organised. It is not clear, however, given the current state of knowledge, which models are preferable.

Apart from the way communication is organised in the network, sensor data fusion, environment state estimation and maintenace, as well as control algorithms play a very important role in the design of WSAN systems. Their implementation in a distributed setting with unreliable communication links is a non-trivial task and leaves much space for further research.

All of the above and the requirement of keeping resource

[1]Some authors use the term *actor* to refer to the same concept.

consumption to a minimum show that WSAN application development is an extremely difficult and challenging task. We propose, therefore, a programming language SOSNA that addresses the key characteristics of WSAN systems allowing researchers, engineers and developers to focus on their-specific higher-level control-related problems. The main highlights of the language are:

- Network topology, node heterogeneity and mobility are abstracted away while making it possible to realise different sensor-sensor, sensor-actuator and actuator-actuator coordination strategies, as presented in [2].
- Maintenance of distributed state can easily be realised by means of accessing stream values at the *previous round*.
- Program execution time is deterministic and proceeds in rounds. This, together with provision of time synchronisation in the network, gives real-time bounds on actuator decision making and allows for actuator synchronisation.
- Static program semantic permits the compiler to generate communication protocols that make use of radio duty-cycling for energy conservation.

SOSNA is a macro-programming language meaning that programs written in the language do not describe actions taken by individual nodes but rather operate groups of nodes. Because SOSNA is also a functional language, physical entities in the network can only be addressed associatively by the data they hold at a given moment in time. Thus, SOSNA programs describe transformations of streams of spatial values. In this setting, building WSAN applications comes down to designing high-level algorithms that transform and fuse sensor data.

2. RELATED WORK

Many different programming paradigms have been proposed for wireless sensor (actuator) network application development. Our work, however, targets macro-programming languages with the primary focus on the stream programming paradigm. Within this scope, two languages are particularly related to SOSNA: the Regiment macro-programming system [16] and Proto [5]. Although SOSNA shares many of their design characteristics (e.g., programs are collections of spatial stream definitions) there are some important differences. Regiment programs are functions evaluated periodically on a distributed computational substrate (the network) and their results must be forwarded to a central point, i.e., the base station. Regiment cannot realise persistent state and because its programs asynchronously fuse data from regions of arbitrary size, it is challenging to implement many sensor data fusion algorithms, where it is required to keep track of the time at which sensor values are sampled.

Proto, on the other hand, does not assume presence of a base station in the system, computation in the network proceeds in a completely decentralised way and persistent state can be maintained in a similar fashion to SOSNA. However, because the asynchronous neighbourhood data exchange operator `reduce-nbrs` (similar to SOSNA's `foldnbrs` operator) is the only means of inter-node communication, consistent data aggregation over regions larger than a 1-hop neighbourhood is non-trivial to realise.

COSMOS [4] is a macro-programming language for building heterogeneous WSN applications. Its programs are expressed as graphs of functional data processing components. COSMOS uses a subset of the C language to define these components and special data structures need to be used to reference a component's input and output channels. Network nodes may aggregate data in their few-hop neighbourhoods and a tree topology is imposed on the whole network. COSMOS programs consist of definitions of functional components, as well as, of specifications of connections between them. In this respect, SOSNA's design is orthogonal to that of COSMOS because streams rather than stream processing components are the core language primitives.

Along similar lines, ATaG is a macro-programming language [18] in which programs are expressed graphically in terms of data processing tasks, data items and communication channels that connect them. Tasks are defined using an imperative programming language (e.g., Java), data items correspond to named variables whose values are exchanged among different tasks possibly running on different network nodes. Hierarchical (clustered) network topology is used and communication is asynchronous and based on the idea of logical neighbourhoods of arbitrary size.

The Kairos [9] and its successor Pleiades [11] programming languages offer an imperative approach to macro-programming wireless sensor network applications. Both languages have C-like syntax and the distributed iteration operator `cfor` plays a central role in the programming models they offer because all concurrent computation in the network is specified sequentially as iterations over sets of nodes that exchange information via shared variables.

The PIECES [12] framework proposes a state-centric approach to building WSAN applications where portions of a physical phenomenon's global state are maintained by mobile principals that interact with each other by means of local collaboration groups. Although this work influenced the design of SOSNA, it differs from it fundamentally as it proposes to implement the logic behind the central concept of principals in the Java programming language.

Sensor query systems offer a different approach to building WSN applications where all nodes in the network are running the same query processing engine and users post queries expressed in a high-level language in order to extract information of interest from the network. TinyDB [15] and the Semantic Streams framework [20] are examples of this approach. Clearly, these systems are not suitable for expressing WSAN applications as they were designed with the purpose of monitoring applications in mind.

3. THE SOSNA LANGUAGE

SOSNA programs operate on streams of spatial values and there are two kinds of spatial values in the language: *fields* and *clusters*. For the sake of explanation, we introduce the notion of a *local value* to correspond to a data item stored at an individual network node, for example, an integer, real number or a tuple of these. Fields, therefore, can be thought of as collections of local values of a given type *present* at a subset of network nodes in a given round of program execution. Every network node can contribute at most one local value to a field, and fields do not have to be contiguous, i.e., they may be composed of disjoint, in terms of topology, network regions. We will refer to nodes contributing local values to a field as *members* of that field. SOSNA

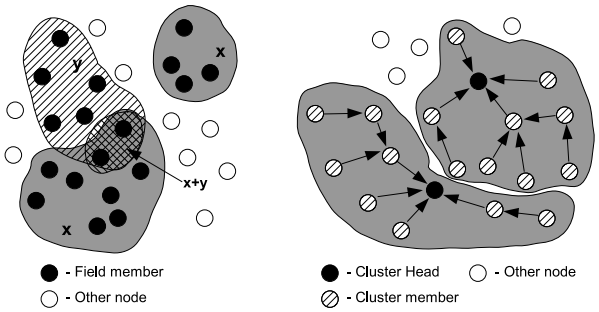

Figure 1: Fields and clusters in SOSNA

field streams have essentially the same semantics as regions in [16].

Clusters, on the other hand, can be thought of as sparse fields whose local values reside on network nodes that may be separated from each other by a number of other nodes that hold no local values. Contrary to fields, these separating nodes form an integral part of clusters and are called *cluster members*. Clusters, therefore, are collections of local values that possess certain spatial extent. We will use the term *cluster heads* to refer to those nodes that hold a cluster's local values. Figure 1 illustrates these ideas.

Ordinary arithmetical operations can be performed on streams and they have to be interpreted as *pointwise* application of the operation in time to individual stream values. Arithmetical operations on individual stream values, i.e., fields and clusters, have to be interpreted as application of the operation pointwise in space to the local values that comprise them. Such a formulation implies that there is no communication overhead involved in arithmetical operations on streams but the results are present on those network nodes that hold local values of all inputs of the operation, see Figure 1.

3.1 Clustering, Folding and Unfolding

Cluster streams are at the core of the SOSNA language. They are defined as field stream transformations by applying one of three clustering operators: `clmax`, `clmin` or `cluster`. In their simplest form these operators take one argument - a field stream and transform it into a cluster stream that is a result of running a clustering and leader election algorithm (CLE). The CLE algorithm is purely data driven, and results in a number of spanning trees of bounded height being formed in the network. The root nodes of these trees correspond to cluster heads and they are the nodes that hold the largest (if the `clmax` operator is used) local value of the input field among all cluster members. We will use the term *cluster extent* (CE) to refer to the maximmum height of trees resulting from the execution of the CLE algorithm. CE is fixed for all cluster streams defined in a program and is a compilation parameter.

There are two reasons for introducing a spanning tree-based clustering algorithm into the language: firstly, cluster heads can be used as local sensor data fusion centres and secondly, clustering provides a means of node selection and may serve as a basic building block of coordination among actuators.

Sensor data fusion is provided by the `fold` operator whose semantics are similar to those of the classical function combinator found in such programming languages as Scheme and whose purpose is to reduce a list of values into a single value by means of recursive application of a function of two arguments. The `fold` operator aggregates values in a tree (cluster member nodes) towards its root - the cluster head which makes its operation similar to that of the aggregation process presented in [15]. The operator takes three arguments: a folding function, a field stream whose values are to be reduced and a cluster stream that will guide the reduction process. The following example realises a simple object tracking application:

```
object = where (sensor > THRESH) clmax sensor
totalMass = fold (+) sensor object
objX = (fold (+) (posX*sensor)  object) / totalMass
objY = (fold (+) (posY*sensor)  object) / totalMass
```

The program defines a cluster stream `object` that is created from a field stream of those local sensor values that are greater than a threshold, i.e., the corresponding sensor nodes are detecting an object. This cluster stream is used to calculate the sum of current sensor values from all cluster members (`totalMass`) and the sum of their x and y spatial positions multiplied by their sensor values in order to estimate the object's position using the centre-of-mass formula:

$$obj_x = \frac{\sum pos_x * sensor}{\sum sensor}.$$

Note that the program reevaluates in every round adjusting its operation to changing conditions in the environment, clusters may change topology from one round to another, the results of program execution are available only at cluster heads of the `object` stream.

The fold operator organises the flow of data in the network from cluster members towards cluster heads. It is also reasonable to consider the opposite direction of information flow - from cluster heads towards all cluster members. This data can be used by cluster members to, for example, make informed decisions as to whether they should participate in a folding process. The `unfold` operator is introduced for that purpose. For example,

```
c = clmax .....  //a cluster
n = fold (+) 1 c
avg = (fold (+) s c) / n
var = (fold (+) (s - unfold avg)^2 c) / n
```

The above program calculates the variance of all sensor readings in the cluster according to the standard formula:

$$Var(s) = \frac{1}{n}\sum_{i=0}^{n}(s_i - \frac{1}{n}\sum_{j=0}^{n}s_i)^2$$

Because the CLE algorithm builds clusters of bounded extent it is possible to calculate the result of this program in one round of program execution. Thus, in every round, the program will first build the cluster `c`, fold recent sensor readings in order to calculate the average value and then it will propagate the result back to cluster members in order for them to include it in the final fold.

Following the analogy from the domain of functional programming languages, `unfold` may take an additional argument - a function that will be used to augment propagated values as they move hop-by-hop away from the cluster head.

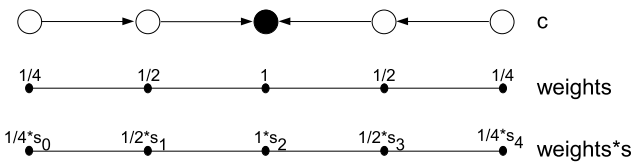

Figure 2: The weighted average program.

The following program computes a weighted average of all sensor values in the cluster:

```
c = clmax s
weights = unfold (map 1 c) (\x -> x/2)
w_avg = fold (+) (s * weights) c
```

We borrow the notation for anonymous function definition from the functional language Haskell. The second line of this program introduces the `map` operator, whose purpose is to map values of a field stream onto a cluster stream resulting in a new cluster stream being defined that differs from the previous one (`c`, in this case), only in values held by its cluster heads. For the new values, in this example, the constant field stream `1` is used. The situation is described in Figure 2.

Sometimes it may be necessary to perform calculations in the network based not on sensor readings reported at individual sensor nodes, but rather based on *concentrations* of values in a small network region. This idea might be particularly useful in situations where sensor readings are noisy and some form of spatial smoothing is required before the data can be further processed. For that purpose, we employ a neighbourhood data aggregation primitive which has very similar semantics to that defined in [5] and is represented in the language by the `foldnbrs` operator. The operator has a similar semantics to the `fold` operator with the difference that it does not need a cluster stream to guide data aggregation because it operates on data exchanged by network nodes within a one-hop network neighbourhood. The following program finds local minima of spatially averaged sensor readings:

```
loc_avg = (foldnbrs (+) s) / (foldnbrs (+) 1)
loc_min = clmin loc_avg
```

3.2 Persistent State

Maintaintenace of controller state plays a very important role in most of modern Control Theory methods [17]. In the wireless sensor network domain target tracking algorithms can often be considered as distributed control systems where the task is not only to discover an object's position but also to "act" on it by moving the centre of computation to the sensor node closest to it. Maintenance of state information that follows the object in the network allows for accuracy improvement of tracking algorithms [13] or for object classification [14].

SOSNA provides a simple way of realising persistent state that was inspired by the synchronous programming language Lustre [7] where the `pre` operator is used to reference a stream value at the previous instant (or execution round). In the WSAN setting, `pre` operates on field and cluster streams and it returns a field or cluster stream that is *delayed* by one round of program execution. This means, essentially, that for a given stream `x` the expression `pre x` refers to the value `x` held in the round previous to the current one. Recursive stream definitions can be used, as in Lustre, in order to realise state that persists between subsequent rounds of program execution. For example, a stream that takes values of subsequent natural numbers on all network nodes (as we said before, numerical constants are extended to fields of values) can be defined as follows: `n = 0=>1+pre n`. The `=>` operator is called the *supplement operator* and, in this case, it initialises the definition of `n` with the value 0. The supplement operator, however, has a different semantics than its counterpart in Lustre. In SOSNA, the result of `x => y` is a stream that comprises of all local values of `y`, as well as of all local values of `x` on nodes that are not members of `y`. The operator, therefore, has spatial semantics.

The `pre` operator together with other language operators provides considerable flexibility with regards to the way in which state can be handled in the program. The previous example was a case of persistent state that was completely distributed. In distributed control applications, however, we may be interested in maintaining the state at actuator nodes. When clustering and leader election are used for actuator coordination it may become challenging to maintain state distributed among them. We use an example of a tracking application where abstract object state needs to follow the physical object in the network:

```
current_obj  = where (sensor>THRESH)
               clmax sensor
obj_total_age = (map 0 current_obj) => 1 +
      (fold (+)  (pre obj_total_age) current_obj)
```

In the above program, `current_obj` is a cluster stream that has its cluster heads located closest to the object. The `obj_total_age` cluster stream is a sum of all values held by its cluster heads in the previous round. If none of the current members of `obj_total_age` was a cluster head in the previous round then the value 0 is assumed, meaning that a new object entered the network. If *some* values were found then they are all summed together.

If the object is moving fast enough to leave the `current_obj` cluster within the duration of one round then the state information is lost; if several objects come close together the state information is improperly merged together; the algorithm cannot disambiguate between two objects moving apart from each other after previously being close. These problems, however, are inherent to all tracking applications and more sophisticated schemes might need to be introduced. On the positive side, the above program essentially realises persistent state - the number of rounds for which the object was being detected in the network - and this state information follows the physical object. If the object disappears, computation ceases and is properly restarted whenever the object appears again; multiple objects are properly handled as long as they are sufficiently separated from each other.

The PIECES framework [12] proposes the use of leader-based detection groups for object-state maintenance. Their formulation of sequential state update exactly matches our stream-based approach and many of the techniques proposed can be reformulated in SOSNA.

3.3 Heterogeneity, Mobility and Actuation

SOSNA programs can be executed in heterogeneous networks. However, because many different network configurations are possible, language primitives refer only to streams

and do not allow for provision of any deployment-specific information. Heterogeneity can be addressed in SOSNA programs by means of evaluating selected streams on a subset of network nodes. For example, if the network consists of two kinds of sensor nodes of which one senses ambient light and the other temperature, then in the following program

```
max_light = clmax light
sum_temp_on_light = fold (+) temp max_light
sum_light_only = within light
                   fold (+) light (clmax light)
```

`max_light` defines a cluster stream comprising possibly of both types of nodes but, when temperature data is aggregated on it, only the temperature nodes contribute to the process while the light nodes only assist by forwarding data towards the cluster head. On the other hand, if the density of deployed light nodes was expected to be high enough, a cluster stream comprising only light nodes can be defined. The `within` operator has the same semantics as the `where` operator with the difference that it constrains the scope of evaluation of its second argument to those nodes at which its first argument is present.

There are two kinds of streams for which it can be decided at compilation time whether they are to be evaluated at a node of a particular type: sensor field streams and constant field streams. Sensor field streams are streams of fields whose values originate from the nodes' hardware components, i.e., sensors, so it is natural that these streams should be evaluated only at nodes providing appropriate hardware. Constants, i.e., fields of constant values but not necessarily equal at every node in the network are introduced as an additional way of addressing heterogeneity. SOSNA programs are, therefore, independent of the underlying deployment configuration while being able to address heterogeneity by means of relations among streams.

Constant boolean field streams can be used to distinguish actuator nodes. For that purpose, we introduce the `cluster` operator which defines a cluster stream based on the values of a boolean field stream. Contrary to the `clmax` and `clmin` operators, `cluster` cannot elect leaders because there is no way of distinguishing among them. Instead, it simply realises a basic spanning-tree based clustering algorithm where network nodes that are not part of the boolean field stream or for which its value is `false` become cluster members picking the closest cluster head in the vicinity as the root of their tree. The following program when run on a network consisting of sensor and actuator nodes results in all sensor readings in some actuator's vicinity (defined by the extent of its cluster) being summed and forwarded towards it:

```
const actuator = true
sum = fold (+) sensor (cluster actuator)
```

The semantics of this program rely on its deployment configuration. If the `actuator` stream was made present on all network nodes, no trees would be formed (as all nodes would be cluster heads) and the `sum` stream would be equivalent to the `sensor` stream.

Mobility can be incorporated in the model at no additional cost. If the actuator nodes were mobile, program semantics would not change because the cluster topology would be reevaluated in every round. The only requirement is that the rate of movement does not affect the topology maintained within a single round of program execution.

SOSNA introduces a special operator `actuate` for synchronisation of actuation in the network. The operator takes, as its arguments, a function `f` and a stream `x`. As a result of its application, the function `f` is evaluated exactly at the end of the round with `x` being passed as its argument so that actuators can be orchestrated to act at the same time. This special purpose operator is introduced because the language does not permit to arbitrarily change the order in which program expressions are evaluated.

4. SEMANTICS AND EXECUTION

The execution of SOSNA programs proceeds in rounds. Each round is split into a number of *steps*, which is program specific. In each execution step, one-hop network neighbours exchange a single *protocol packet*. The step duration is, therefore, a compilation parameter and has to be long enough so that all one-hop neighbours in the largest neighbourhood in the network have a chance to send their protocol packet.

Every network operator in SOSNA requires a fixed number of steps to be evaluated. The simplest one - `foldnbrs` is evaluated in only one execution step. The number of steps required to evaluate all clustering operators is a compilation parameter and influences the way the application executes. As was mentioned in Section 3, the CLE algorithm generates clusters of bounded extent. It is the CE constant that defines the number of execution steps in which the CLE algorithm is executed. Given that in every step every network node sends out at most one protocol packet, all spanning trees constructed by the CLE algorithm are of height not greater than CE. The `fold` operator aggregates data on already constructed trees, thus the number of steps required for its evaluation is exactly CE. The same rule holds for the `unfold` operator as it is evaluated on an already constructed cluster. Other language constructs do not have any communication-related semantics, thus their execution is local and is considered *instantaneous*.

This *static* program semantics permits the compiler to infer the maximum number of steps of which each round should consist. Given that in every step each node sends out at most one packet, as well as, the fact that step duration is fixed throughout the network, each round of program execution takes a fixed amount of time and this time can be calculated off-line.

The values of the CE constant in practice will depend on a particular application. For example, in [6] the authors propose to realise target tracking using collaboration groups spanning two hops from the group leader. Other applications might require larger CE values, especially when static topologies are used, e.g., sensor nodes placed in some area around an immobile actuator. A question, however, arises as to whether a single cluster extent value would be good for all clusters that an application might define. We believe that it would because the `within` and `where` operators can be used to constrain the range of communication in the network when it has to be kept small. This characteristic lies at the core of SOSNA's programming model. It results in real-time bounds on program execution in the network but, on the other hand, it potentially constrains the applicability of the language.

Thanks to the static language semantics, the SOSNA compiler is able to infer what network operators can be evaluated in parallel and to fit all data items that need to be communi-

Step number	0	1	2	3
Packet contents	p	y, 1	m_0, u	m_1, u
Step number	4	5	6	7
Packet contents	m_2, u	r_0	r_1	r_2

Table 1: Example protocol schedule.

cated in a single protocol packet (a similar approach is used by Proto [5]). Those operators that have data dependencies have to be executed sequentially what affects the number of steps that comprise one execution round. We use the following program as an example to clarify these concepts:

```
y = foldnbrs max p
x = (foldnbrs (+) y) / (foldnbrs (+) 1)
m = clmax x
u  = unfold (map s m)
r = fold (*) (x - u) m
```

The protocol schedule for one round of execution of the above program is presented in Table 1 where we use variables m_i and r_i to denote intermediate results of the CLE algorithm and the evaluation of the `fold` operator respectively. Note that value unfolding can be realised in parallel with cluster formation since the unfolded values originate from cluster heads.

There are exactly eight steps required for one round of this program to be completely evaluated. Not all nodes, however, have to send protocol packets in every round. Nodes join the protocol in a certain step if they have data items to be sent. It may also happen in a step of program execution that a node can contribute only a subset of all data items required in this step. As a remedy, each protocol packet carries a data item presence indicator (a bit array in the packet header) that identifies data items contained in the packet.

4.1 Time Synchronisation

The correct execution of SOSNA programs relies on the provision of a time synchronisation service in the network and the effects of asynchronous inter-node communication on program consistency need further investigation. Network time synchronisation, however, was a crucial component in some real-world deployments, e.g., [10]. Also, when time is synchronised across the network precise sensor data fusion can be realised in SOSNA if we note that, regardles of the number of steps in one round of program execution, the values of all sensor data streams referenced in the program are sampled *at the same time*, i.e., at the beginning of the round.

4.2 The CLE Algorithm

The clustering and leader election algorithm that SOSNA uses is similar in principle to the DAM protocol presented in [8]: the input field stream passed as an argument to the `clmax` and `clmin` operators is used as a set of contention values for future cluster heads. Each node that belongs to this field sends out its local field value and listens for the values sent by its neighbours. When a node receives a contention value greater than the one it is holding in the current protocol step (without loosing generality we confine the explanation to the `clmax` operator) it takes it as the current maximmum and marks the sender as its parent in the tree. The received value is sent again in the next step of algorithm execution in order to be propagated further on. If no contention value greater than the initial one was received the node considers itself to be a cluster head.

Nodes participating in the CLE algorithm, apart form the identification of their parent node in the tree, also maintain the distance in hops to the root of that tree. This information is used when the `fold` operator is evaluated so that each tree member knows in which protocol step it has to send data to its parent.

5. WSAN COORDINATION

In this section, we investigate SOSNA's ability to implement basic WSAN coordination models, as described by Akyildiz *et al.* [2]. This survey proposes two general architectures for WSANs: semi-automated architecture with a central controller that gathers all sensor readings and instructs all actuators; and automated architecture in which there is no central authority and control is distributed among actuators. According to Akyildiz, the automated architecture is preferable due to its better scalability, low communication latency, as well as due to potentially lower communicational overhead imposed on network nodes because of lower network congestion. SOSNA offers a localised model of computation that is based on clusters of bounded extent and one-hop neighbourhood data exchange. Long-range, multi-hop communication cannot, therefore, be realised in the language without impractically increasing the CE constant.

As far as sensor-actuator coordination is concerned, the challenges listed in the paper can be addressed in the following way: real-time communication is provided in SOSNA programs thanks to the static semantics of the language; SOSNA enables radio duty-cycling for energy conservation (see Section 6); each sensor event can be reported to at most one actuator (cluster head), hence ordering and one-time respone issues do not arise; event tracking by sensors is possible as described in Section 3; event delivery to particular regions in the network can be realised as the following example illustrates:

```
dest_pos = ...
destination = clmin (pos - unfold dest_pos)
data_at_dest = fold (=) event_data destination
```

The `dest_pos` cluster describes the location to which the current value of `event_data` should be sent, the cluster heads of the `destination` cluster are the nodes closest to the desired point and event data can be sent there by means of stream value folding with the identity function. It is not clear, however, how computation of the minimum sensor and actuator coverage sets can be expressed in SOSNA.

Similarly to the challenges in sensor-actuator coordination, the issues in actuator-actuator coordination can be addressed in SOSNA as follows: actuators can send information to other actuators using the `unfold` operator; actuation synchronisation can be realised using the `actuate` operator; sensor nodes relay actuator data by default; forwarding event data towards actuators located at certain places in space can be realised in the same way as in the example above; actuators have limitted means of recognition of the state of other actuators, in fact, only summaries resulting from the application of the `fold` operator can be used. Again, the problem of optimum actuator assignment seems to be difficult to address in SOSNA.

The authors state that actuators are likely to be able to communicate over long-range channels. The possibility of communication media heterogeneity in SOSNA was not sufficiently studied so far, however, the idea of multidimensional programming presented in [3] could potentially be applied in SOSNA by extending all network-related language operators with a prefix that would describe the *dimension* in which the operator has to be evaluated. The dimension information could be used by the compiler to chose the appropriate network controller.

6. ENERGY EFFICIENCY

There is a potential for substantial energy conservation in SOSNA applications. We emphasise the ability of the nodes to switch their radios off in addition to limiting the total number of messages sent, as it has been reported that radio transceivers in the listening mode consume almost as much energy as in the sending mode [1]. Static program semantics allow network nodes to make informed decisions on whether some neighbours might have some relevant information to send in the current protocol step or not and control the state of their radios accordingly. For example, because cluster members know their distance to the cluster head, as well as the step in which data aggregation commences a radio duty-cycling mechanism, similar to that of [15], can be employed when network time synchronisation is provided.

The scope of all computation and communication in the network can be also reduced when the `where` and `within` operators are used. This means, in particular, that not only network nodes may decide not to send a message in a given execution step but also they can switch their radios completely off when they find them selves, for example, not being members of any cluster.

Constant field and cluster streams can be exploited by the compiler in order to reduce the frequency of cluster structure refresh, as well as techniques for caching information about the values of constant streams at one-hop neighbours may be developed for even more informed radio duty cycling.

7. CONCLUSIONS AND FUTURE WORK

This paper proposes a stream-based macro-programming language designed for wireless sensor and actuator network applications. The language matches key properties of WSAN systems, as well as, it promisses implementation of energy-efficient applications. Because of space limitations, this paper presents only an overview of the key characteristics of the language. Future work will focus on finishing the compiler implementation and real-world evaluation of SOSNA programs.

8. REFERENCES

[1] Mica2 mote datasheet. *Crossbow Technology Inc.*

[2] I. F. Akyildiz and I. H. Kasimoglu. Wireless sensor and actor networks: research challenges. *Ad Hoc Networks*, 2(4):351–367, 2004.

[3] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge. *Multidimensional programming.* Oxford University Press, Oxford, UK, 1995.

[4] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. *SIGOPS Oper. Syst. Rev.*, 41(3):159–172, 2007.

[5] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[6] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks. In *MobiSys '03*, pages 201–214, New York, NY, USA, 2003. ACM.

[7] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with lustre. In *SAFECOMP '99*, pages 396–409, London, UK, 1999. Springer-Verlag.

[8] Q. Fang, F. Zhao, and L. Guibas. Lightweight sensing and communication protocols for target enumeration and aggregation. In *MobiHoc '03*, pages 165–176, New York, NY, USA, 2003. ACM.

[9] R. Gummadi, N. Kothari, R. Govindan, and T. Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05*, pages 1–2, New York, NY, USA, 2005. ACM.

[10] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, 2006.

[11] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. *SIGPLAN Not.*, 42(6):200–210, 2007.

[12] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 02(4):50–62, 2003.

[13] J. Liu, J. Reich, and F. Zhao. Collaborative in-network processing for target tracking. *EURASIP J. Appl. Signal Process.*, 2003(1):378–391, 2003.

[14] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3):543–576, 2006.

[15] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[16] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *IPSN '07*, pages 489–498, New York, NY, USA, 2007. ACM.

[17] K. Ogata. *Modern Control Engineering.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[18] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco. Expressing sensor network interaction patterns using data-driven macroprogramming. In *PERCOMW '07*, pages 255–260, Washington, DC, USA, 2007. IEEE Computer Society.

[19] B. Sinopoli, C. Sharp, L. Schenato, S. Schaffert, and S. Sastry. Distributed control applications within sensor networks. *Proceedings of the IEEE*, 91(8), 2003.

[20] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *EWSN 2006.*