A State-Based Language for Sensor-Actuator Networks

(Research supported in part by NSF award 0519907)

Anish Arora Ohio State University
Mohamed Gouda University of Texas
Jason O. Hallstrom Clemson University
Ted Herman University of Iowa
William M. Leal Ohio State University
Nigamanth Sridhar Cleveland State University

Abstract

This paper introduces a language design for sensor-actuator networks. The main features are communication by a soft-state abstraction and behavior control by periodic rule evaluation. These features enable a state-based, rather than event-based, style of programming. Dynamic changes to network configuration and failures of components are automatically handled by this approach. The design choices target applications which experience low-to-moderate rates of sensor input and which do not require extreme, low-latency sensor processing and actuation. Coordinated actuation is concisely expressed in this language.

1 Introduction

Sensor Networking is an area born of the confluence of technological progress, new application opportunities, and visions of fine-grained embedding of computation in physical environments. Two fundamental characteristics of sensor networking are the allure of inexpensive, programmable sensing devices and the extreme variability of potential applications. Deployments also vary considerably: sensor nodes (sometimes just called sensors) may form *ad hoc* networks, or be arranged as fixed-topology networks dependent on base stations; different combinations of movement (mobile sensors, traveling data collectors) are possible. These characteristics do not dictate any single, best methodology for specification and programming of sensor network behavior. Indeed, as applications of sensor networks evolve, a diversity of tools and languages will contribute to the success of sensor network programming.

At present, tools and languages for programming sensor nodes differ from those used for base stations. This is logical, since sensor nodes have quite limited computing resources (small memory, small bandwidth, limited battery power) whereas base stations can be programmed with desktop and even enterprise-scale tools. Sensors either have no operating system, that is, they are programmed using compilers and supporting libraries for embedded processors, or the operating system is tailored to the constraints of the sensor. The techniques of enterprise software development, which rely on significant runtime support, do not fit the constraints of a sensor. The current theme of software development for sensor nodes is an event-driven programming style.

Event-based programming reflects hardware characteristics, where components have nonblocking interfaces and signal interrupts upon completion of operations; interrupts are generated when environmental inputs exceed thresholds; interrupts also occur when timers overflow and when messages arrive. Event-based programming is "close to the wire", enabling lean implementations (with minimal overhead) that can satisfy tight timing constraints. This aspect of lean implementation is suited to the constraints of sensor nodes, however it comes with a price: software based on events complicates program design [5].

Multiple languages and platforms together with an event-driven programming style present a developer with a level of technical complexity that can be daunting. Though event-based programming likely can't be beat for applications requiring extremely low latency of exfiltration or applications demanding maximum communication throughput, many application functions do not demand that level of performance. Our position is that an alternative language design with a state-based foundation, available on multiple platforms, can simplify programming. This paper proposes such a language. *DESAL* (Dynamic Embedded Sensing and Actuation Language), rests on the following five principles.

(*1*) *State-based model of programming.* DESAL eschews event-driven logic: programs do not declare procedures for event-triggered reaction to sensor input, timeouts, or message reception. Instead, programs specify the actions that should take place when given conditions, represented by expressions of state variables, hold. Our experience with sensor network software is that, for a broad class of applications, a significant portion of the logic neither requires tight timing nor precise accounting for all hardware events. Periodic sampling and background recording of some data to flash memory, ongoing health-status reports about the network, reselection of which sensors should be activated, could be activities without stringent timing constraints (whereas signal processing of acoustic sensors or accelerometers, once engaged, needs to be timely). Further, such portions of the application with loose timing constraints are typically parts that application (or domain expert) programmers, sensor network administrators, or deployment specialists most likely need to tune and customize. Previous investigations observed that such portions of an application benefit from scripting languages, virtual machines, and remote methods for query and update; we add to this direction by observing that the event-driven style of device drivers, often found in timed signal processing routines and intensive, high-bandwidth streaming protocols, need not be used for coding higher-level application logic.

In state-based programs, computation is explicitly dependent on declared state variables, and not dependent on implicit or hidden values (the stack, suspension of control and expected resumption after an event, and so on). Thereby program invariants are more easily expressed, asserted, and vali-

dated; programs can be automatically checked to see whether they have actions for all combinations of states (which promotes fault tolerance and even self-stabilization). This position shares with traditions of domain experts, who typically model physical systems in terms of state variables for ease of analysis.

(*2*) *Communication* is expressed by sharing state variables rather than operations that construct and send messages. A "soft state" implementation enables variable sharing, which takes care of underlying concerns of packing values into messages and retrying failed communication. In essence, DESAL's state sharing between nodes provides cached access to remote values (subject, of course, to the dynamics of connectivity in the sensor network).

(*3*) *Rule-based programs.* DESAL's execution model is rule evaluation. The body of a program is a set of statements, each statement having a boolean condition (guard) and an associated action (a sequence of assignments to state variables). One step in execution consists of selecting a statement whose guard is true and performing the associated command or assignment.

(*4*) *Dynamic binding.* Some decisions about how state variables are shared are deferred to runtime, depending on the deployed network topology, sensor availability, and other factors. DESAL presumes a dynamic binding service equipped with sufficient health monitoring and robustness to support shared variable communication in a sensor network subject to faults and some types of asynchronous reconfiguration.

(*5*) *Timing* of rule selection, periodicity parameters for monitoring binding, and controlling the frequency of shared variable communication between nodes are tunable values exposed to the programmer. Where possible, depending on the sensor platform, DESAL also provides a synchronized clock for programmer convenience. Time synchronization, combined with deep sleep between activations, means that distributed components can wake up together, exchange and act on state, then sleep again, thus enabling very low duty cycles and hence low power consumption.

Contributions. Taken individually, each of (*1*)–(*5*) have been proposed and investigated in previous sensor-network research (Section 8 points to some comparable literature). Our primary contribution is to combine features to yield a simple but capable language. As a multi-platform language, users do not need to master multiple technologies to use DESAL; communication details are largely hidden; and rule-based execution over state state matches commmonly-used models. The inclusion of clock synchronization in the runtime, which seems to be underutilized in current tools and languages, makes it particularly easy to express coordinated actuation.

2 Program and State

Program specification is comprised of several sections: a program name and parameter section, a state declaration section, a binding section, and a section for rules describing program behavior. Section 3 provides details on the binding section and Sections 4 and 5 explain rules in DESAL programs.

The *program name* section can specify implementation-dependent parameters needed for compilation (which we do not discuss in this paper). It is possible for multiple DESAL programs to execute within a sensor network and interact, and even possible that two or more DESAL programs could execute within a node. Program names provide scoping of named state elements. Each DESAL program executing within a node needs a distinct name so that its state variables have global reference names. Because a program can be instantiated at many nodes, we use the term *component* to refer to an instantiation of a program within a node.

The *state declaration section* provides names and types for variables manipulated by rules. Declarations may label a variable as either *shared* or *local* (the default). Section 3 describes how shared variables are linked to other components, as directed by statements in a program's binding section: variable sharing is *directed*, that is, for any shared variable, one component has write access to the variable and other (sharing) components have only read access. The precise sharing relation between variables may not be resolved until runtime, so shared variable declarations need flexibility. The statement

```
shared int16 m[ ];
```

declares `m` to be a shared array of indeterminant size (limited, however, by platform constraints). This declaration enables runtime binding to adjust the effective size of `m` depending on the number of sharing components. Variable types can be simple (integer, float, *etc*) or structures (in the sense of a C struct).

3 Shared Variable Binding

Binding is a dynamic service provided in DESAL's runtime platform. Before we present the syntax of binding statements, preliminary concepts are introduced. The *soft-state store* is a best-effort cache of variables shared from one component to another. When the binding service establishes sharing from, say, variable `a` to variable `b`, we say that `b` is *bound* to `a`; `a` is the *source* of the binding, and `b` is a *sink* of the binding. Binding is generally a transient property, so that a bound variable can cease to be so depending on network conditions, node health, and related factors. When `b` is bound to `a`, the component reading `b` uses `b` as a proxy for reading `a`. The implementation of the soft-state store is such that reading `b` could get an out-of-date value for `a`: our design sacrifices atomic (synchronous) semantics in favor of a lightweight, adaptive implementation.

The selection of what is shared is controlled by specifying names (variables and components) and *attributes*. Attributes refer to constant or slowly changing characteristics of a node and its environment. Examples of attributes could be a node's unique identifier, its sensor modes, its location/proximity in a static network, or connectivity (neighbor relation) in the network. Non-examples of attributes would include acoustic sensor readings, rapidly fluctuating values in temperature, pressure and similar values that inhibit the formation of reasonably durable bindings.

The *binding engine* is runtime middleware that continually binds and unbinds shared variables as constrained by binding specifications, network conditions, and node re-

sources. The binding engine is also charged with transporting values between nodes. Discussion of underlying network protocols and various implementation possibilities for the binding engine is outside the scope of this paper, however we can say that several tuning parameters and policies governing the binding engine would be specified in DESAL programs (in the program name section and binding section).

```
1  bindings {
2      b1 <- id.C1.x1;
3      b2 <- *.C1.x2;
4      b3 -> id.C1.x3;
5      b4 -> *.C1.x4; }
```

The program fragment above shows examples of elementary bindings. The first binding declaration declares a binding variable b1. The arrow directed to the left indicates that b1 is a read-only binding. The right-hand side indicates that b1 is bound to the variable x1 declared by program C1, a component on the node identified by id. The second binding declaration is similar, but specifies a *wild-card binding*; it binds b2 to any (if b2 is a single variable) or all (if b2 is an array) instances of C1.x2 in the logical neighborhood (discussed below). Binding declarations for b3 and b4 are analogous to the declarations for b1 and b2, but specify write bindings. Hence, b3 can be used to effect updates on the component variable C1.x3 hosted by the node identified by id. In the case of b4, which may be bound to multiple components, writes against the binding are dispatched to each of the end-points by the binding service.

```
1  bindings {
2      b5 <- *.C1.x5 :
3          (sensors & mag) != 0 &&
4          position.x >= 5 &&
5          position.x <= 10; }
```

The program fragment above has a binding specification that refers to attributes. Variable b5 (if an array) can bind to all instances of C1.x5 in the logical neighborhood equipped with a magnetic sensor, and which have geographic positions within the desired range.

In addition to the constraints explicitly specified in the binding section, DESAL imposes runtime constraints. No shared variable can be simultaneously the sink of multiple bindings. No shared variable can be simultaneously sink and source of multiple bindings. One shared variable *can* be the source of multiple bindings (with distinct sinks). Specifications can invite binding from sink (<-) or from source (->). Binding can be established between, say C1.a and C2.b, if C1's binding specification for a is a constraint satisfied by C2.b and C2 has no binding specification for b (that is, the binding constraint on C2.b is empty). Binding is also possible when both endpoints have binding statements, provided that one is a sink specification and the other is a source specification; in this case, for a binding from C1.a to C2.b be established, the source-constraints of C1.a must be compatible with sink-constraints of C2.b.

The reader may wonder whether both types of binding, -> and <- are necessary. While it is possible to express binding constraints entirely from the sink side, our experience with small thought experiments is that applications can be expressed more succinctly and modularly by permitting both orientations. A practical example of this is a frequently recurring application architecture, namely the *base station centric* architecture [1], which puts nearly all of the computational activity in the base station and uses sensors in a command-and-control fashion. Here, it makes sense to put all binding specifications in the base station, which has (almost) no memory constraints. The binding engine implementation at sensors can be minimized in this type of architecture. Both types of binding are necessary at the base station, -> to distribute commands to sensors and <- to collect responses from sensors.

DESAL's binding engine mediates between program binding specifications, current binding state, and network connectivity and transport services. Above, we refer to binding with respect to the *logical neighborhood*. In rudimentary implementations of DESAL, this translates to the 1-hop neighborhood of a node, and in richer implementations of DESAL, the logical neighborhood can span the network; in a base station centric architecture, neighborhood only has meaning between base station and sensor, where a typical routing structure such as a spanning tree (or generalization thereof) could suffice to provide the logical neighborhood.

An established binding at runtime might seem similar to the fundamental abstraction of a connection in network design. However our experience with the dynamics of connectivity in sensor networks and limited resource available to sensor nodes argues for a lighter weight concept than, say, a TCP connection. Previous proposals for binding in sensor networks (on richer platforms) constrains duration of bindings, so as to avoid "thrashing" of bindings [11]. Note that even a guaranteed duration can be problematic, because connectivity can evolve rapidly, leaving established bindings essentially useless. Our approach to this and similar problems is (*i*) to allow awareness of binding (knowing whether a shared variable is bound or not) at the sink, but not the source; and (*ii*) to support binding specifications that refer to connectivity attributes (link quality, signal strength, or other measures). A history of connectivity attributes can be a practical heuristic for binding. However, an alternative for applications deployed on known, static topology could be bindings based on that topology, without regard to connectivity dynamics, plus having enough redundancy in the topology to support application requirements despite some connectivity loss.

The binding engine takes advantage of (*i*), that only the sink is aware of binding, in its soft-state implementation of shared variable transport. The program at the sink refers to the most recent value of the bound shared variable in soft-state store when executing statements. If connectivity is lost, the most recent value remains unchanged and available to the program until the binding engine infers the loss of connectivity and withdraws the sink's binding. How, then, can a sink know whether or not a bound shared variable is stale? Our answer to this question, discussed in Section 5, is to make timestamps available to DESAL programs, which can be used to define freshness according to application requirements.

Local Bindings. One special case diverges from network mechanisms outlined above. Local bindings are established

within a node, and can be statically engineered when programs are composed and compiled. One example of this would be bindings between DESAL components hosted on the same device; another example (probably the most frequently occurring case) is the use of binding for communication between a DESAL component and the host operating system, drivers of sensors, or services that process significant quantity of sensor input. Many system calls and sensor driver methods can be encapsulated by wrappers that translate request and response to shared variable communication. Our convention for examples in this paper is that variables shared with system services begin with $, for instance $Clock and $Temperature.

4 Rule-Specified Behavior

When executed, a component changes state according to rules that modify its state variables. Except for shared variables that have sink (<-) binding specifications, rule execution can modify any of the component's variables. The remainder of this section describes the syntax and runtime evaluation of rules; we defer discussion of timing aspects to Section 5.

The *rules section* of a DESAL program consists of one or more *body* subsections, and each body subsection contains a list of guarded commands. A *guarded command* is a statement of the form *guard* -> *command*, where the guard is a boolean-type expression on state variables and the command is an assignment to state variables. Lists of guarded commands are formed using the [] operator. The example

```
t>=v -> v=t-1   []   i==n  -> b=True
```

composes a list of two guarded commands. A common idiom of DESAL is one consisting of a list of of guarded commands defined over the entries of an array, such as

```
t[0]>v -> v=t[0]  [] t[1]>v -> v=t[1]  [] ···
          []  t[m-1]>v -> v=t[m-1]
```

For this recurring idiom, DESAL provides closed-form syntax abbreviating the above as

```
( [] i: 0<=i<m:  v>t[i] -> v=t[i] )
```

For indeterminant arrays, the preferred form for the idiom above is further abbreviated to

```
( [] i::  v>t[i] -> v=t[i] )                    (1)
```

since the DESAL runtime infers the effective size of the array. Similar quantified notation can be used in expressions within guarded commands: for example, (count i:: t[i]>0) returns the number of (bound) positive elements of t.

At runtime, the *rule engine* evaluates guards and executes commands. With respect to any guarded command of a component, rule execution is atomic: no component sink variable is changed by the binding engine during the rule's execution and no other rule in the component is executed concurrently. Therefore, the assignment part of a guarded command can be a compound expression, including functions:

```
t>v -> v=t ; if v>99 { g[k].rec=f(v,g[k].rec) }
```

From a formal perspective, because execution is atomic, it turns out that a compound expression is equivalent to one (multiple variable) assignment statement.

The rule engine skips guarded commands that refer to shared variables that have sink (<-) bindings declared and are currently unbound. For example, at any point during component execution, evaluation of (1) above would reduce to a no-op if all instances of t[i] are unbound at that point. DESAL program execution is thus automatically *dynamic* and *adaptive* to current network conditions: as the logical neighborhood evolves, evaluation of guarded commands adapts without requiring developers to explicitly write conditional expressions on current topology.

The [] operator, which is the unit composing a list of guarded commands within a body subsection, owes some motivation to early research on concurrency semantics: A[]B denotes nondeterministic selection of A or B during execution, which can represent concurrent processes in an interleaving semantics. Following this interpretation of [], the binding engine could choose to execute guarded commands of a body section in any order. Our motivation for nondeterministic selection is more specific: efficient evaluation of a list of guarded commands may not be a fixed sequential evaluation of the list, but instead some ordering chosen at runtime by the binding engine. (Efficient evaluation depends on platform characteristics and the timing of system component interactions.) Our proposal for body section execution is that, once a body section is selected to execute, each guarded command in that section has the opportunity to be executed at least once, but the order is unknown to the programmer.

5 Execution Timing

Rule evaluation based on state variables decouples underlying system and device events from programmed response, which puts delay between an event occurrence and subsequent processing. Systems that refrain from synchronous processing of device interrupts may use polling instead of interrupts to achieve real-time objectives. By scheduling repeated polling at appropriate time intervals, delays between event occurrence and processing are limited. The rule engine takes a similar approach for scheduling: body specifications in a program may specify time intervals for periodic evaluation. A body declaration "body period(15) *guarded command list*" specifies that the list of guarded commands evaluates once per 15 time units. (To simplify the presentation we omit detailed definition of time units.)

The rule engine scheduling of periodic bodies is best-effort. We do not presume a real-time system platform, and DESAL components may compete with other services in sensor nodes, so timed scheduling can be imprecise. (We generally expect relative error in implemented time intervals decreases as a function of period length.) Guarded commands can read a built-in shared variable $Clock (a system clock) to reliably measure elapsed time.

Our full vision for DESAL leverages the availability of clock synchronization, which now has numerous implementations [21]. With synchronized time, the $Clock can be copied in guarded command assignment to a timestamp field of a source shared variable. This makes it possible for sinks to measure freshness and for command-control patterns to enforce timeliness policies. The rule engine also exploits synchronized clocks: if components in different nodes have body period(15) subsections, then rule engines in the nodes

begin their evaluations at the same time (again, on a best-effort basis). This aspect of DESAL enables *coordinated actuation*; many nodes can act in concert to sense or actuate together. More general syntax for the body section is

`body period(`*interval,offset,repeat*`)`

The *interval* specifies the length of the time period. By default, as stated above, all components with such a body subsection align their periodic evaluations. With *offset*, components can stagger the beginning of their periods. The *repeat* parameter, if given, specifies an additional (sub) periodic step. The specification `period(100,50,5)` targets guarded command evaluation at time slots 50, 55, 60, ..., 95 (relative to 0 being the start time of each period).

For example, a base station program might specify `period(30,0)` with a guarded statement list that assigns a source shared variable `command`. Sensor nodes have access to the value of `command` sometime later, because the binding engine deposits the value to their respective soft state stores. Suppose the expected time delay for this transfer is 5 time units. In sensor nodes, a plausible body specification is `period(30,15)`, which provides enough time (plus a tolerance factor) to get the latest command, perform local computation, and assign to a source `response` shared variable that is bound to a base station sink variable. In essence, this example lifts the suggested pattern of polling (as an alternative to event-driven programming) from the level of node to the level of end-to-end service. Of course, transfer of `command` to sensor and `response` can fail (message loss), bindings can be lost (topology change), however these cases are easily detectable at the base station using sequence numbers or timestamps as tags on `command` and `response`.

Our proposal to allow multiple body subsections within the rule section of a program is based on the observation that many applications are composed of activities that naturally have different time scales. Data acquisition and exfiltration have one desired frequency, management functions (network health monitoring, tuning) have another.

Another aspect of execution timing impacts the programmer's view of communication through shared variables. A programmer has a choice of at least two views on the semantics of shared state. The programmer might only care about the most recent available state, such as the current temperature, in which case missed sensor readings are unimportant; or the programmer might want to coordinate state transfer, in which case synchronized timed rule evaluation can be employed.

6 Coordination Example

The context for the example presented in this section is a sensor-actuator network with a base station. Each node has a static identifier that can be referenced in bindings and guarded commands as `$Id`. The base station's `$Id` equals zero; nodes with identifiers 1..12 have *strobe* actuators. The interface to the strobe device uses shared variables `$Strobex` and `$Strobed`. `$Strobex` can be written; `$Strobed` is read-only to the program. Assigning `$Strobex=$Strobed` is taken to be a command to the strobe device to emit a signal. After the signal has been fired, the strobe device sets `$Strobed` so that `$Strobex!=$Strobed`. Other network nodes have *strobe sensors*, which register intensity (or phase) of a strobe firing. Variable `$Rstrobe` provides the latest value from a strobe sensor and a function `RSreset()` requests to reset `$Rstrobe`. Due to space constraints, we present fragments of actuator and sensor nodes only. (The base station would collect data from the strobe sensors via sink variables.)

```
1  body period(130,10*$Id) {
2     $Clock - lastStrobe > 10 -> {
3        lastStrobe=$Clock;
4        $Strobex=$Strobed; } }
```

Above is a fragment of the program for an actuator node. Each actuator component schedules strobe firing at a distinct time within the periodic interval of 130 time units.

```
1  state
2     type struct sval {
3        clock_t time; int sval; }
4     shared sval T;
5  rules body period(130,15,10) {
6     $Rstrobe!=0 -> {
7        T.sval=$Rstrobe;
8        T.time=$Clock;
9        RSreset(); } }
```

The program of strobe sensors above records to a shared variable `T`, a structure containing a sensor value and a timestamp. No binding is given because a sink binding is specified at the base station. Strobe sensors record at times 15, 25, *etc*, if the strobe sensor has registered any value.

7 Phased Data Collection

The example in this section involves iterative base station collection of an average temperature taken when at least 70% of the sensors have reported. When the body subsection for data collection is executed at the base station, and the 70%-threshold is met, then the base station initiates a new phase, which the binding engine disseminates to all sensors (`phase` is shared to all sensors). No specific timing constraints are given in this program; an untimed body statement for the base station is:

```
1  rules body {
2     (count i:: R[i].p==phase)/Nb(R)>=0.7 -> {
3        AvgRecord(R,phase);
4        phase=phase+1; } }
```

Shared array `R` is a sink at the base station, bound to all sensors in the logical neighborhood; `Nb(R)` is the number of bound elements of `R`. Field `R[i].p` is a value copied by the sensor from its soft-state copy of `phase`; field `R[i].v` is a temperature value. Statement 3 records the average of the `R[i].v` fields where `R[i].p==phase`. The body of the sensor program has the statement

```
r.p!=bph -> { r.p=bph; r.v=$Temperature; }
```

Shared variable `r` is source to the sink array `R`. Shared variable `bph` is sink to base station source `phase`; phase difference observed at a sensor is acknowledged in `r.p` along with recording temperature.

8 Related Research

Much of current sensor network programming is event-based, inheriting the tradition of embedded system programming, which emphasizes fine-grained control of devices us-

ing minimal resources, and is close to the platform architecture. This style tends to be hand-crafted; toolkits for enhanced software productivity enable a variety of services to be combined for programming an application [19, 10, 15]; a methodology for developing the applications [13], can also accelerate application program development. Most toolkits and distributed computing platforms continue two aspects of traditional embedded system programming, the event-driven approach to processing and an imperative, sequential (but multithreaded) programming language.

High level abstractions exploit common-case application design patterns, enabling a declarative specification style for base station query or continual aggregation [6, 7, 8]. Macroprogramming, or "programming the network as a whole" makes analogy to MIMD architectures, and research has shown that numerous applications fit a pattern of function-parameterized and regional stream processing of events [5, 4, 3].

Between the extremes of device-centric, embedded programming systems and higher-level macroprogramming styles, there are frameworks and toolkits that presume some networked middleware services, and conceive of the sensor network as a type of distributed computing platform. For example, given a virtual machine implementation in all sensor nodes, mobile agents and an abstract shared dataspace [9] could be used to program applications.

Expressing computation with a set of conditional rules is an old idea, in theory dating to early proof derivation systems. In the space of sensor network research, rule-based programming is advocated in [17, 18] and investigated by ongoing research [12, 2, 20] which are closely related to DESAL, though are either event-based or lacking the timing facilities and soft-state binding of our design. Binding issues in sensor networks have been examined in [11], but not in combination with rule-based execution.

Timed scheduling as a language feature is implemented in [19], though not for a rule-based execution model. Giotto [14] offers a programming model with fine-grained timing of tasks, but suited to safety-critical embedded systems where task timing is known and a real-time substrate is available.

9 Summary

DESAL combines notions of rule-based component specification, communication by a shared variable abstraction built upon binding constraints and soft-state, and facilities for time awareness and timed execution. Periodic scheduling of guarded commands compensates for the lack of event-triggered execution. Sections 6 and 7 suggest how DESAL is applicable to timed and untimed execution control. We are completing initial implementations of DESAL compilers and plan to assess resource and performance costs in comparison to hand-crafted, event-oriented implementations of various protocols. In an initial experiment with a DESAL prototype, a program with a dozen guarded commands, the binding engine, the rule engine, clock synchronization, radio stack, sensor drivers, *etc*, altogether consume about 1600 bytes of RAM on a Telos mote.

10 References

[1] O Gnawali, KY Jang, J Paek, M Viera, R Govindan, B Greenstein, A Joki, D Estrin, E Kohler. The tenet architecture for tiered sensor networks. *SensSys'06 Proceedings of the 4th International Conference on Embedded Network Sensor Systems*, pp. 153-166, 2006.

[2] S Sen, R Cardell-Oliver. A rule-based language for programming wireless sensor actuator networks using frequence and communication. In *Proceedings of EMNETS'06*, 2006.

[3] R Gummadi, O Gnawali, R Govindan. Macro-programming wireless sensor networks using Kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS'05)*, 2005.

[4] R Newton, Arvind, M Welsh. Building up to macroprogramming: an intermediate language for sensor networks. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, 2005.

[5] M Welsh, G Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, 2004.

[6] S Madden, M Franklin, J Hellerstein, W Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.

[7] S Madden, M Franklin, J Hellerstein, W Hong. TinyDB: an acquisitional query processing system for sensor networks. `telegraph.cs.berkeley.edu/tinydb`.

[8] S Chen, PB Gibbons, S Nath. Database-centric programming for wide-area sensor systems. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[9] CL Fok, GC Roman, C Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'05)*, 2005.

[10] T Abdelzaher, B Blum, Q Cao, Y Chen, D Evans, J George, S George, L Gu, T He, S Krishnamurthy, L Luo, S Son, J Stankovic, R Stoleru, A Wood. EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pp. 582-589, 2004.

[11] C Intanagonwiwat, R Gupta, A Vahdat. Declarative resource naming for macroprogramming wireless networks of embedded systems. UCSD Technical Report CS2005-0827, 2005.

[12] M Arumugam, SS Kulkarni. Programming sensor networks made easy. Michigan State Technical Report MSU-CSE-05-25, 2005.

[13] J Liu, M Chu, J Liu, J Reich, F Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing*, October-December, pp. 50-62, 2003.

[14] TA Henzinger, B Horowitz, CM Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84-99, 2003.

[15] R Pandey, J Koshy, E Bergstrom, D Durkin, J Wu, I Wirwan. System software instructure for sensor network applications. In *Proceedings of WSNA'04*, 2004

[16] J Steffan, L Fiege, M Cilia, A Buchmann. Scoping in wireless sensor networks. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-Hoc Computing*, pp. 167-171, 2004.

[17] M Zoumboulakis, G Roussos, A Poulovassilis. Active rules for sensor databases. In *International Workshop on Data Management for Sensor Networks DMSN (VLDB'04)*, 2004.

[18] E Klavins, RM Murray. Distributed algorithms for cooperative control. *IEEE Pervasive Computing*, 3(1):56-65, 2004.

[19] B Greenstein, E Kohler, D Estrin. A sensor network application construction kit (SNACK). In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004.

[20] K Terfloth, G Wittenburg, J Schiller. Rule-oriented programming for wireless sensor networks. *International Conference on Distributed Computing in Sensor Networks (DCOSS'06)*, 2006.

[21] B Sundararaman, U Buy, AD Kshemkalyani. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks* 3(2005):281-323.