

A Rule-Based Language for Programming Wireless Sensor Actuator Networks using Frequency and Communication

Shondip Sen
Commonwealth Scientific &
Industrial Research Organization

shondip.sen@csiro.au
<http://www.asl.csiro.au/>

Rachel Cardell-Oliver
School of Computer Science
The University of Western Australia

rachel@csse.uwa.edu.au
<http://web.csse.uwa.edu.au/>

Abstract

This paper proposes a new rule-based language for programming heterogeneous sensor actuator networks. Rules are declarative statements which are evaluated using sensor readings spanning multiple sources. We combine our declarative rules system with the notion of frequency based operation and communication to allow a compiler to emit a work schedule optimized for minimal power consumption. Combining rules with frequency information also removes the need for concurrency in the form of threads. The language presented in this paper is compiled to an intermediate byte-code which can be executed by a reconfigurable virtual machine. Our language is designed to be reliable, energy efficient and expressive, which is achieved through predictable execution, enabling compile time checks and optimizations.

1. Introduction

The most common functions of a sensor node are to sample the physical properties of an environment, communicate the readings and actuate based on one or more inputs. The actual processing and transmission capability of each node in a sensor network is limited by the amount of available power. This single constraint limits many of the possible choices that can be made in designing programming languages for sensor networks.

This paper introduces a rule-based language for sensor networks with a much simpler execution model than existing languages. It has no threaded concurrency or dynamic scheduling. We believe our simpler execution model will make it easier to prove programs correct and minimize power usage, whilst still being sufficiently expressive to capture diverse high level abstractions.

The class of applications that we consider in this paper are those which close the loop by not only gathering data, but also interpreting sensor readings leading to in-network computation and actuation. An example of the type of application we are considering involves land management and the herding of livestock [2] which is explored in later sections.

The rule-based language introduced in this paper is intended as an *intermediate language* for sensor networks. Other intermediate languages include the token machine of Newton, Arvind and Welsh [7], and TinyScript and Mottle developed for the Maté virtual machine [4]. High level programming primitives such as abstract regions [9, 6] can be compiled into our intermediate language, and so executed reliably and efficiently on sensor networks. The language can also be used directly, for example to encode routing protocols.

In our language, the gap between programs and sensor network hardware is bridged by byte code executed on a reconfigurable hardware-level virtual machine. Other intermediate languages are implemented as middleware on top of the TinyOS/nesc system and so inherit some of the reliability and complexity problems of that platform [5].

2. Language Design

All imperative languages have an explicit entry point which initiates a sequence of instructions to be executed. Normally we wish to execute the instructions in sequence as quickly as possible. Programs are separated into named functions or procedures with strict control structures. Within the context of a sensor network, a program may have an explicit or implicit loop to stop the program from ending while it waits for an event to be fired. Note that it is not necessarily known which event will trigger first and hence the flow of control is hard to predict,

especially if more than one event can be handled simultaneously.

In our language we propose to combine declarative rules with imperative control in sequence to remove the explicit event-loop. Declarative rules are periodically tested for their truthfulness. The frequency at which each rule is tested depends on an explicit duty cycle. By expressing work in terms of frequency, we change the way a program is written to reflect the workload, resulting in a sequence of operations that can be combined into a work schedule. Ideally we would like items on the schedule to be as compact as possible and the schedule to be sparse over time to conserve power.

We have avoided the introduction of explicit concurrency in our language as we believe that it is unnecessary in most cases. By declaring the frequency at which different operations should be evaluated, we are able to generate a single schedule without overlapping operations or explicitly creating separate threads. Radio communication is managed through a scheduled time division MAC layer.

2.1. Rules and Actions

Rules are declarative expressions representing a condition which guards the invocation of a sequence of action statements. If the conditions of a rule are satisfied, the matching action can be performed. This may lead to state changes which cause other rules to become true:

$$Condition \longrightarrow Action$$

The conditional part of rule is a boolean expression that can be translated into conjunctive normal form. The literals in a conditional rule statement represent either local variables or shared state accessed via communication, taking the form:

$$(C_{1,1} \vee \dots \vee C_{1,n}) \wedge (C_{2,1} \vee \dots \vee C_{2,n}) \wedge \dots$$

Each $C_{i,j}$ is a literal, comparison or other expression that can be evaluated to *true* or *false*. The action is a sequence of imperative instructions executed in order. An action statement may refer to or modify any of the literals used in the conditional part of the rule.

2.2. Duty Cycle Workloads

A set of one or more rules is enclosed within an explicit duty cycle to form a task. If we declare a set of rules that have a duty-cycle of one minute, they will be evaluated once every minute until the program ends. The *every* statement defines the period of a task, where the task itself is the set of condition action rules in the scope of that statement. A duty cycle is expressed using the keyword *every* as follows:

```
every (interval) {
  Condition1  $\longrightarrow$  Action1
  Condition2  $\longrightarrow$  Action2
  Condition3  $\longrightarrow$  Action3
}
```

In this example, there are three conditional rules with matching action statements. The rules are tested periodically at a frequency corresponding to *interval* which is a unit of time. A program is a set of tasks defined by listing *every*-statements one after another. The *priority* of a task is defined by its position in the program list; that is, by order in which tasks are defined. Our language supports event-based programming by frequently scheduling high priority tasks to check for any events that require fast response. If more than one rule is true, their corresponding actions will be executed in sequence. We have elected to use fixed-priority, static scheduling [1], rather than dynamic scheduling, because then we can analyze the correctness of programs at compile time.

Periodic execution implies that programs will run indefinitely. While this is the most likely case for simple programs, we may want to terminate a duty-cycle. We can do this by including a rule to cause the terminate action. The terminating condition may become true after a number of iterations or after some elapsed time:

```
every (interval) {
  Condition1  $\longrightarrow$  Action1
  Condition2  $\longrightarrow$  Action2
  time > 100  $\longrightarrow$  Terminate
}
```

2.3. Communication, Sensing and Actuation

Communication is by definition an important operation performed within a sensor network. This is reflected in our language by making communication a first class operation. Communication allows a program to send messages between rules being executed locally, or at other nodes. Rules can be used to guard communication to enforce a sequence or to set an agreed interval for streaming.

To communicate a message we require a medium. We call the communication medium a channel. The simplest type of channel is a location in the memory heap where messages can be exchanged. A channel can be used as an input for reading the value from a sensor or an output for performing actuation. A more sophisticated channel may allow messages to be passed between rules executing on different sensor nodes, providing a method for sharing values between different memory spaces. Channels are the only form of persistent shared variables in our language.

The transfer of information over channels is scheduled in between program tasks. For communication between nodes,

a TDMA-like mechanism is used to schedule the transfer of radio messages. We also allow for routing mechanisms here to transfer messages over multiple radio hops. Radio communication is *not* assumed to be reliable, and as in self-stabilising algorithms, repeated transfer of messages is used to overcome possible communication losses. In future work we plan to investigate algorithms for deriving optimal communication schedules for a given program.

A channel can carry a single message at any given time. The type and scope of a channel must be declared at initialization and cannot be changed thereafter. There are two atomic operations that can be used on a channel to send or receive messages. Only one process may acquire and transmit a message at any given time. There may be multiple readers implying Concurrent Read Exclusive Write or CREW semantics. The channel is named on the left hand side and the message to the right of the double arrow operator:

```
send:    channel << message
receive: channel >> message
```

Reading a channel returns a local copy of the stored value and does not remove any data, so reading a channel is a repeatable process. Writing to a channel replaces any previously stored data. The new data becomes available when it is read from the channel. Asynchronous communication provides flexibility while rules can be used to enforce predictability.

A reader attempting to receive a message from an empty channel causes the operation to fail, returning an empty or *false* response. The send operation does not complete until the message has been fully transmitted or copied. The semantics of a channel provide for asynchronous communication which is good for analysis because we can reason about chains of read and write operations. This property can also be used to save energy because the radio is only on when communication is taking place.

The copy-overwrite semantics of a channel were chosen because typically in a sensor network we wish to act upon the most recent data in real-time and not process old data. However, older messages can be stored by creating a rule whose action is to queue the data for later use.

Channels provide a method for passing data between different scopes, memory spaces and nodes. In this abstraction, the channel metaphor is similar to a pointer but inhibits the possibility of aliasing effects that plague programs written in languages that allow pointers to be manipulated directly.

The receive operation of a channel can be used in the conditional part of a rule declaration. If there is a message available to be received the condition returns a *true* value. Both the send and receive operations can be used in the action part of a rule.

3. Evaluation

The aim of our language is to provide an expressive intermediate level abstraction for writing applications to generate a predictable output that can be reasoned about. Our intention is to provide reliability and energy efficiency through static reasoning using compile time checks.

3.1. Example: Grazing Cows

We illustrate our language with a program for controlling the quality of the pasture on a cattle farm. A field contains soil moisture sensors and an irrigation system. The field also contains cows carrying GPS enabled sensor-actuators that can be used to steer animals using stimulation [2, 8]. The aim is to maintain the field so that it is neither over-grazed or too moist.

Our example program in Figure 1, has three tasks enclosed within *every* statements. The first task samples the soil moisture at intervals of 10 minutes to check whether the ground is too wet, too dry or ok, and the result is written into a channel called *soil.state*. The second task tests the soil moisture state provided by the first task at intervals of 5 minutes. The local GPS position is read and depending on the soil state, a rule transmits the originators location and a direction in order to *attract* or *repel* cows from the area. If the soil is too wet or too dry, the cows are steered away from the location. The rules also use the soil state to control the irrigation system. If the soil is ok, the GPS message attracts cows to the fresh pasture. The final task causes the animal steering actuators to move the cows toward the best pasture for grazing. Steering is performed more often than the sampling of soil moisture, taking advantage of the multiple read semantics of a channel.

3.2. Heterogeneity

In our example we did not explicitly state which rules are executed on specific nodes. Instead, we rely on the capability of a node to make that decision, which can be taken at compile time with knowledge of the system or at runtime by interrogating our reconfigurable virtual machine for its capabilities in terms of the channels it exposes [3]. For example, only nodes with soil moisture sensors can execute rules accessing that type of channel. Similarly, only nodes with the capability for controlling irrigation or steering animals can execute such rules.

In order to verify the correctness of programs in heterogeneous networks, we need to make assumptions about the numbers of nodes with different capabilities in the network. In future work we plan to develop general algorithms for supporting such analysis.

```

// sample soil moisture
every (10 mins) {
  (moisture >> val) -> {
    (val > upper_threshold) ->
      soil_state << too_wet;
    (val < lower_threshold) ->
      soil_state << too_dry;
    (val > lower_threshold) and
      (val < upper_threshold) ->
      soil_state << ok;
  }
}

// actuate based on soil moisture
every (5 mins) {
  gps >> my_location;

  (soil_state >> soil) {
    (soil == too_wet) -> {
      target << (<my_location, repel>);
      irrigation << stop;
    }

    (soil == too_dry) -> {
      target << (<my_location, repel>);
      irrigation << start;
    }

    (soil == ok) -> {
      target << (<my_location, attract>);
      irrigation << null;
    }
  }
}

// steer cows
every(1 min) {
  gps >> my_location;
  target >> (<location, direction>);

  (direction == attract) and
  (my_location != location) ->
    move_toward << target;

  (direction == repel) ->
    move_away << target;
}

```

Figure 1. Grazing Cows

3.3. Expressiveness

The purpose of the rule-based language is to simplify sensor network programming. We demonstrated that the

rule-based language achieves this, by showing how a single script can be used to generate a program that operates across multiple nodes in a heterogeneous environment. To write an equivalent program in a language like nesC would require significantly more effort to specify how individual nodes would behave.

The expressiveness our language may at first appear to be restrictive, for example, in not supporting the dynamic scheduling of tasks. However, this can be achieved by having a high frequency task with a rule that only takes a sample under certain conditions, and otherwise does not need to take a sample.

3.4. Reliability

A critical property for sensor network applications is reliability: programs need to function correctly for long periods, with no software bugs that cause programs to crash.

Our goal is to enhance software reliability using a fully predictable programming framework that can be analyzed at compile time for correctness. So, where there are trade-offs to be made between, say, immediate message transmission and predictable performance, we have opted for the latter. Since programs have a predictable schedule, it is also possible to analyze the correctness of their behavior. The combined communication traces of two or more nodes can be analyzed using standard methods such as weakest preconditions on the operations of the global trace. For example, deadlock and live-lock analysis can be performed by tracing communication actions.

Verification methods for self-stabilizing algorithms are also applicable to our rule-based programs, since both use the model of periodic transmission and reception of messages to achieve reliable operation in an unpredictable environment.

3.5. Energy Efficiency

Sensor network nodes consume differing amounts of energy when using the radio, reading sensors, performing local calculations, or sleeping. Radio transmission, reception and listening, are the most expensive operations on a sensor node in terms of energy use, consuming three or more times the energy of any other operation. Our language makes use of underlying TDMA protocols that turn the radio on only if it is required for transmitting a packet or listening to its neighbors, and so make very efficient use of this resource.

Other resources, such as sensor boards and flash memory for logging data, also consume power. The compiler is able to identify the resources required by each of its tasks, and so schedules those resources to be turned on only when required. Sleeping consumes the least energy, and for long

lived sensor networks, nodes should be in a sleep state as often as possible.

Sensor network applications are typically characterized by very low duty cycles, and so we expect the compiled schedule will be sparse, with nodes in sleeping state most of the time. However, if a set of tasks can not be scheduled because the frequency of one or more tasks is too high, or a task's worst case execution time is too long, then the programmer is notified at compile time, allowing the program to be modified before an error occurs.

4. Concluding Remarks

The problem of developing a method for expressing problems in a distributed system has existed for many years. Within the context of sensor networks, the problem is altered by the unique characteristics of unpredictable communication, limited resources and power. In this paper we present a new abstraction layer for expressing programs for a sensor network which is not focused on the actions of individual nodes. Our aims are to improve on the expressive power of existing solutions to generate more predictable behaviors that can be used to steer an optimizing compiler. The main contributions are the ability to express conditions and actions using declarative rules together with communication and duty cycles for each task to express how often they should be evaluated. This leads to the generation of a work schedule for execution on a reconfigurable virtual machine. Expressing the rate at which operations should be performed feeds information into the compilation process to optimize the work schedule and eliminates the need for explicit concurrency.

References

- [1] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*. Addison Wesley Longman, Mar. 2001.
- [2] Z. Butler, P. Corke, R. Peterson, and D. Rus. Virtual fences for controlling cows. In *ICRA*, pages 4429–4436, New Orleans, Apr. 2004.
- [3] C. Frank and K. Romer. Algorithms for generic role assignment in wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 230–242, New York, NY, USA, 2005. ACM Press.
- [4] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM Press, 2002.
- [5] P. Levis, D. Gay, V. Handziski, J.-H. Hauer, B. Greenstein, M. Turon, J. Hui, K. Klues, C. Sharp, R. Szewczyk, J. Polastre, P. Buonadonna, L. Nachman, G. Tolle, D. Culler, and A. Wolisz. T2: A second generation os for embedded sensor networks. Number TKN-05-007, Nov. 2005.
- [6] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Enviro-suite: An environmentally immersive programming framework for sensor networks. *ACM Transaction on Embedded Computing Systems*, 2006. To appear in 2006.
- [7] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: an intermediate language for sensor networks. In *IPSN*, pages 37–44, 2005.
- [8] P. Sikka, P. Corke, and L. Overs. Wireless sensor devices for animal tracking and control. In *Proc. First IEEE Workshop on Embedded Networked Sensors*, pages 446–454, Tampa, Florida, November 2004.
- [9] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation*, NSDI '04, March 2004.