NORTHWESTERN
UNIVERSITY

Department of Electrical Engineering and Computer Science

Technical Report
NU-EECS-00-00
August 9, 2013

Trigger Compilation for Energy-Efficient Reactive Behavior in Wireless Sensor Networks

Jedidiah McClurg **Goce Trajcevski**

Abstract

In this project, we seek to address some of the issues which have traditionally made design and development of Wireless Sensor Networks (WSN) very difficult. These issues include the *complexity of writing distributed systems on a large scale*, as well as *heterogeneity of the involved motes/environments*, and the *necessity of energy-efficient communication as the nodes collaborate.* Ultimately, solving this problem will require a combination of tools that will (1) provide high-level programming constructs which enable users to specify behavior of the entire system rather than individual motes, (2) offer the ability to compile these system-level specifications into native code for the heterogeneous motes, and (3) allow the WSN communication strategies to adjust dynamically in order to keep energy usage to a minimum. Towards this end, we have developed the TCE^2 (Trigger Compilation for Energy Efficiency) system, and this document discusses the details of our system, and presents an accompanying demonstration to show that we have made progress in the above three directions.

Keywords: Compiler, Energy Efficiency, Heterogeneous Nodes, ECA Trigger, Wireless Sensor Network, Reactive Behavior, Environmental Monitoring, Active Database, TelosB, SunSPOT

Trigger Compilation for Energy-Efficient Reactive Behavior in Wireless Sensor Networks

Jedidiah McClurg Goce Trajcevski
Department of Electrical Engineering and Computer Science
{jrm807, goce}@eecs.northwestern.edu

Abstract

In this project, we seek to address some of the issues which have traditionally made design and development of Wireless Sensor Networks (WSN) very difficult. These issues include the *complexity of writing distributed systems on a large scale*, as well as *heterogeneity of the involved motes/environments*, and the *necessity of energy-efficient communication as the nodes collaborate*. Ultimately, solving this problem will require a combination of tools that will (1) provide high-level programming constructs which enable users to specify behavior of the entire system rather than individual motes, (2) offer the ability to compile these system-level specifications into native code for the heterogeneous motes, and (3) allow the WSN communication strategies to adjust dynamically in order to keep energy usage to a minimum. Towards this end, we have developed the TCE^2 (Trigger Compilation for Energy Efficiency) system, and this document discusses the details of our system, and presents an accompanying demonstration to show that we have made progress in the above three directions.

1. Introduction and Objectives

Wireless sensor networks (WSN) are becoming increasingly commonplace in our day-to-day lives. They are now used for everything from controlling robots to monitoring geological phenomena. This surprising amount of variety means that programmers and engineers from a large number of backgrounds need to develop, configure, and support wireless sensor networks. In addition, as the scale of these networks increases, it becomes imperative to take a data-centric view of the WSN, visualizing the network as a distributed database.

The concept of triggers from the study of Active Databases has been proposed as a general approach for WSN programming in a way that 1) is accessible to people from different backgrounds and 2) produces energy-efficient communication between the network nodes of large complicated WSNs. In this approach, individual and aggregate data can then be collected by issuing database-style "queries". For example, consider the small portion of an environmental monitoring WSN shown in Figure 1.

The network consists of two physical regions $R1$ and $R2$, in which motes (represented by circles) monitor luminance and temperature, and can report readings back to the sink through tree-based routing. An example query would be something like the one shown in Figure 2.

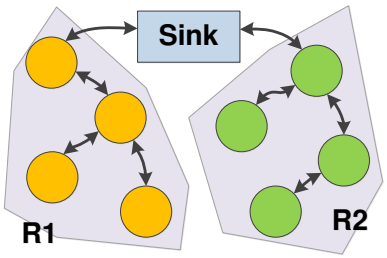

Figure 1. Basic Environmental Monitoring Scenario with Two Regions

Q1: *Whenever the average luminance in Region R2 exceeds 80 lumens over the past minute, if the average temperature in Region R1 is below 65°F, then switch on the electric heating unit and record the time.*

Figure 2. A Query for the Basic Environmental Monitoring Example

This could correspond to a set of outdoor nodes which monitor solar panel light source, and a set of indoor sensors which monitor room temperature, allowing the electric-powered heating unit to turn on during a sunny day. There are several key things to notice about this scenario.

- There is a **temporal** and **event-based** aspect (*"over the past minute"*).
- The sensors/data are **geographically distributed** (i.e. regions $R1$ and $R2$).
- Sensors may be **heterogeneous** (i.e. hardened outdoor light sensors vs. cheap indoor thermostat temperature sensors)

Thus, a WSN database management system (DBMS) must allow for those things to be specified. In addition, the DBMS should allow for the following important features:

- The WSN may need to be **reconfigured dynamically** (i.e. provided with control information).
- Motes are **energy-constrained**, so radio use should be minimized.
- Motes are **resource-constrained**, so mote-based code should not be complex.

One of the ways to address this is via ECA triggers [21], which are database queries of the form *on Event, if Condition holds,*

then do Action. It has been show that this approach works well, and allows for expressive language constructs [16]. Inherent in the efficient execution of ECA triggers is the consideration of push-based vs. pull-based communication [10] as the motes collaborate to detect events and check conditions. To the best of our knowledge, a full-featured ECA programming language/environment which takes this into account is noticeably lacking.

To this end, we present in this report the TCE^2 (Trigger Compilation for Energy Efficiency) project as a way to address the need for such a system, especially in the context of heterogeneous nodes. The system contains a graphical editor which allows network topology to be specified and geographical regions to be defined, as well as mote-level settings such as address and sensor/mote type, taking care of the **geographical** and **reconfiguration** aspects. The centerpiece of the system is the Basic Event-Enabled Query Language (BeeQL), which allows queries to be specified in a **temporal** and **event-based** way, as shown in Figure 3.

```
ON EVENT (AVG(R2.light[-60,0]) > 80)
IF (AVG(R1.temp) < 65) (
  SET R1.heating = TRUE;
  SELECT AVG({R1.temp,R2.light});
  SELECT time
)
```

Figure 3. BeeQL Code for the Basic Environmental Monitoring Query

The other part of the system is a compiler which takes a system-level BeeQL specification and compiles it to mote-level code, based on the graphically-specified control information, taking care of the **heterogeneous** aspect by targeting different code types depending on sensor/mote selections. We have enabled TelosB and SunSPOT motes in this way. Since the BeeQL query language is "close enough" to the native language on these devices, the generated code is efficient, so the *resource-constrained aspect* is satisfied. Finally, the subsequent version of the compiler will generate code with several different modes to create efficiency for **energy-constrained** network operation.

In order to demonstrate the value of our system, we perform several real-world monitoring demonstrations. In the first demonstration, we set up a network of TelosB and SunSPOT motes, and cause our software to arrange them according to a tree topology like the one shown in Figure 1. For this arrangement, we compile system-level queries like the one above to mote-level code on the individual motes, and then show that the correct behavior is happening in the WSN.

1.1 Paper Organization

This paper is organized as follows. Section 2 presents an overview of our system and provides an illustrative example. Section 3 presents the BeeQL language in detail. Section 4 presents energy conservation techniques which we propose for better system performance. Section 5 describes the compilation of system-level BeeQL queries to mote-level code. Section 6 discusses the system implementation, and section 7 presents the detailed description of a demonstration. Section 8 presents related work, and then section 9 concludes the paper and mentions future additions to the project.

2. System Overview

The system is organized into two components, the GUI frontend, and the compiler, as shown in Figure 4

The GUI allows the WSN topology to be specified, and individual nodes to be selected from among the connected motes. Based on the regions and sensor types chosen in the GUI, the user can then

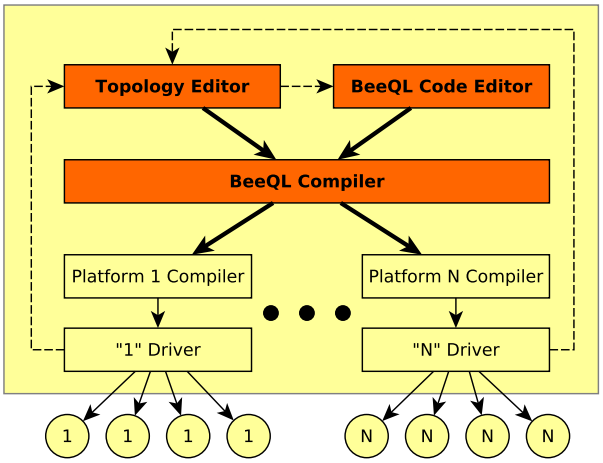

Figure 4. System Overview

input BeeQL code queries via the code editor box. The user can then hit the "compile" button, which sends the topology information and BeeQL code to the compiler. The compiler then translates the system-level code into source code for each target (e.g. nesC code for TelosB and Java code for SunSPOT or the simulator). The respective native-code compilers are then invoked, producing a native binary for each mote.

2.1 Specifying WSN Topology and Regions

When the GUI is invoked, the system first detects (via the individual drivers) which motes are connected and ready to be programmed. Using this information, the graphical editor allows the user to build the desired network topology by clicking to create new nodes. Other connected hosts (sinks) are also allowed to be created. As users on the remote host machines configure their WSN,

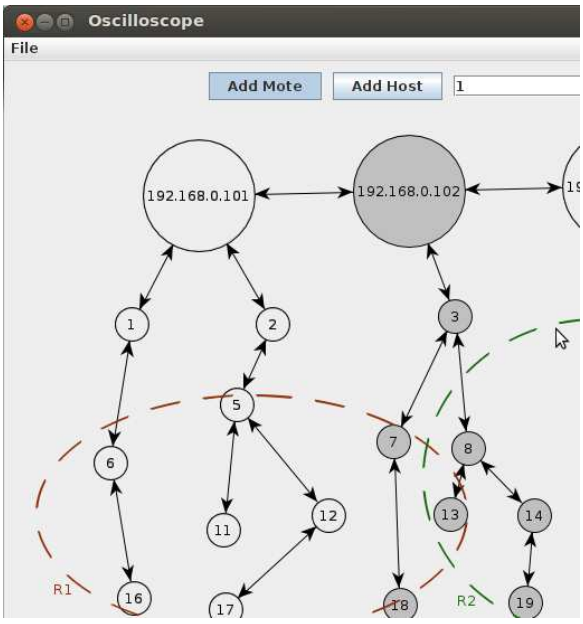

Figure 5. Connecting Sensors and Specifying Regions

the topology updates are propagated back to the local machine. For example, in Figure 5, the local machine has address 192.168.0.102,

and the topology configured on the remotely-connected machine 192.168.0.101 is visible. Nodes and hosts can be arranged to match actual geographical placement by clicking and dragging, and can be deleted by CTRL-clicking. Connections can be made by clicking on each of the two respective nodes. Elliptical regions can be selected by clicking and dragging on the background. In this way, the user can assign a set of nodes to a region by enclosing them with an ellipse, and can attach a name to the region.

By setting up the graphical representation of the WSN in this way, the user can force the networked motes to arrange themselves with the respective connections. For now, only tree connections are supported. As the WSN operates, messages are then routed to their destinations via local routing tables which are built immediately before the compilation step (see Section 5).

2.2 Specifying and Issuing Queries

Once the configuration of the WSN has been completed via the graphical interface, the user can write a BeeQL query in the code panel. Region variables can be chosen based on the graphical representation. For example, in Figure 6, we have specified our original query **Q1**, setting up a trigger that will result in action being taken when the respective events occur in the deployed R1 and R2 regions of the WSN, and the results of a *SELECT* query to be displayed in the window. The query can be executed by pressing the "Compile"

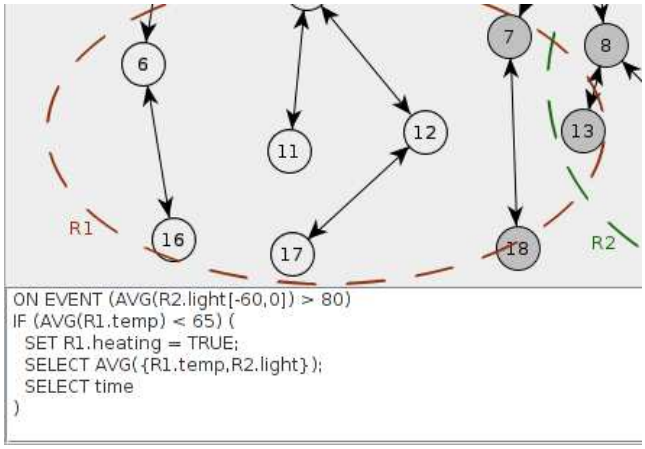

Figure 6. Specifying BeeQL Queries

button. This causes the compiler to be invoked on the code. The first stage of the compilation takes this system-level code and translates it to mote-level BeeQL code for each connected node. A code-generation stage then converts each mote-level BeeQL code to a corresponding target source code. These source programs are then distributed to remotely-connected hosts if necessary, and the corresponding native compilers are invoked to produce native mote-level code and flash the binaries onto the attached motes.

Once this is completed, the WSN begins operating, first entering an initial phase to perform time synchronization using the proper topology, and then executing the triggers as instructed.

3. Basic Event-Enabled Query Language (BeeQL)

The centerpiece of our system is the Basic Event-Enabled Query Language, BeeQL (pronounced *BeeQuel*, as in *Sequel* for *SQL*). This language retains the simplicity and compilation efficiency of database query languages (e.g. SQL) while offering higher-level constructs such as events and syntactic pattern matching [1].

All data is numeric, with floating-point numbers (e.g. `10.123`) and integers (e.g. `123`) allowed, and the `TRUE` and `FALSE` constants denoting `1` and `0` respectively. Arbitrary variables can be used, and are initialized to `0`. Variables can be local to a specific region, e.g. `R2.temp`. The main construct of a BeeQL query is the ECA trigger, as discussed previously.

3.1 Simple Temporal Model

We take a linear, discrete view of time, with a granularity of 1 second. The value of a variable `R1.temp` at time t can be determined via `R1.temp[t]`, with $t = 0$ being the current time, and $t < 0$ representing past values. We also allow predicates to be specified using interval temporal logic [2]. For example, variable values over the past 30 seconds can be determined by using the construct `R1.temp[-29,0]`. Issued at time t, this returns a set of the 30 values of `R1.temp` at times $(t - 29) \cdots t$. "Future" times ($t > 0$) are supported in action statements, such as `SET R1.var[0,30] = TRUE`, which causes the variable to remain `TRUE` for the next 30 seconds, and then return to its previous value.

3.2 Simple Event Model

In the BeeQL language, an *event* can take the form of 1.) a temporal predicate becoming `TRUE` in terms of the preceding discussion regarding the time model, or 2.) a named event, with possibly bound variables. Events can be generated internally to a node (e.g. a change in sensor readings), or externally (e.g. a change in an average value over a separate region). Complex events [27], e.g. $E_1; E_2$ are not yet handled.

An event E is specified as described in Table 1 and other binary

$(D_1$ `<` $D_1)$	fired at the moment when data D_1 becomes numerically less than D_2
$(D_1$ `<=` $D_1)$	fired at the moment when data D_1 becomes numerically less than or equal to D_2
$(D_1$ `==` $D_1)$	fired at the moment when data D_1 becomes numerically equal to D_2
$(V$ `MATCHES` $P)$	fired at the moment when variable V matches pattern P
P	fired when the named event matching pattern P occurs

Table 1. Specifying Events

operators `>` and `>=` are also provided. Data D_i can be numeric constants, local/remote variables, or operators on any combination of those.

For example, we could define the event "luminance in region R2 exceeds 80 lumens" with the expression `(R2.light > 80)`, or the event "message T_C received" as `RECV(T_C)`.

3.3 Language Syntax and Functionality

A BeeQL program is specified as a list of ECA triggers. Each trigger takes the form

`ON EVENT` E `IF` C A_1 [`ELSE` $(A_2 | T)$]

where E is an event as described in the previous section, C is a condition (any of the Table 1 constructs except for the last one, i.e.

[1] In this paper, we use the word "pattern" in the way commonly seen in the programming languages literature, i.e. a pattern is a sequence of linguistic constructs with a certain syntactic structure.

named events), and A_i are actions, and T is potentially another `IF` $\cdots$ `ELSE` test.

Actions can take the following forms shown in Table 2.

$O(X_0, \cdots, X_n)$	invokes operator O on numeric parameters X_i
`SET` X `=` V	sets variable X to the numeric value V
`SET` X `+=` V	increases variable X by the numeric value V
`SELECT` D	selects data D
$((A_1 \mid T_1); \cdots)$	a sequence of actions A_i and/or tests T_i

Table 2. Specifying Actions

The language contains the built-in operators shown in Table 3 so

`MIN(`X`)`	returns the minimum of a set X of values
`MAX(`X`)`	returns the maximum of a set X of values
`AVG(`X`)`	returns the average of a set X of values
`SEND(`P`)`	sends a message named by pattern P

Table 3. Built-in Operators

we can use these in the respective queries. For example, we could implement the query "report average temperature in region R1" as `SELECT AVG(R1.temp)`.

A full description of the BeeQL language (in the form of an EBNF grammar) is provided in the Appendix A.

4. Proposed Energy Conservation Techniques

Triggers and statements executed locally on the motes do not use very much power, however code involving access to `remote` variables uses the radio and hence is expensive in terms of energy usage. Our system currently uses a single communication strategy for generated code, i.e. the Static Pull mode discussed below. In this chapter, we propose an approach to reducing the WSN communication (and hence the energy usage) by using other communication strategies.

4.1 Static Push-pull

Our compiler could produce more energy-efficient code for the WSN motes by paying special attention to the communication mode between pairs of nodes which are cooperating to execute an ECA trigger. Although in general it is not possible to optimally choose a static push/pull mode for each node which minimizes energy consumption, we present a simple heuristic which we believe would help in some cases.

Consider a new query **Q2**:

```
ON EVENT (AVG(R2.light) > 90)
IF (AVG(R1.temp) > 80) (
  SELECT AVG(R2.light)
)
```

Since this query needs an average over region $R2$ to detect the event, the $R2$ motes can be put in push mode to regularly provide their readings to the region's base station $B2$. When $B2$ detects the condition on the average, it can query $B1$, which in turn can query its $R1$ nodes. In other words, $B2$ and the $R1$ motes are put in **pull mode**.

Consider a slightly-modified version of the query, **Q3**:

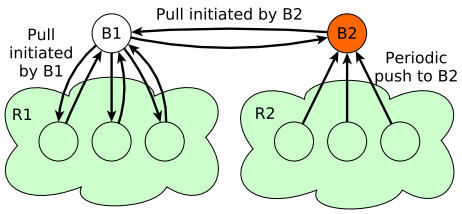

Figure 7. Static Pull Mode

```
ON EVENT (AVG(R2.light) > 90)
IF (AVG(R1.temp[-30,0]) > 80) (
  SELECT AVG(R2.light)
)
```

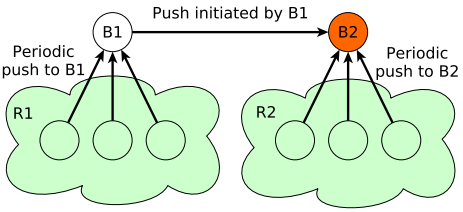

Figure 8. Static Push Mode

Again, $B2$ needs to regularly compute an average over $R2$ to detect the event, so $R2$ motes are put in push mode. However, now $B2$ needs to know an average *over the last 30 seconds* from $R1$. In this case, putting $B2$ in pull mode will generate unnecessary radio use as $B2$ queries for the $R1$ average regularly. Thus, all the nodes are put in **push mode**

In other words, a simple heuristic is to place nodes whose triggers depend on an *instantaneous* external value in **pull mode**, and place nodes whose triggers depend on a *value over time* in **push mode**.

4.2 Adaptive Push-pull

The simple static push/pull approach does not work in all real-world situations. For example, consider the following query **Q3**:

```
ON EVENT (R2.cars > 1)
IF (AVG(R1.temp) > 120) (
  SELECT AVG(R2.light)
)
```

It may very well be the case that the event `(R2.cars > 1)` happens very often, for example on a high-traffic road. On the other hand, the condition `(AVG(R1.temp) > 120)` may be true very infrequently, e.g. in the winter. Thus, assigning "pull" mode via our heuristic would cause a massive amount of unnecessary query traffic to R1. It may also be the case that sometimes the converse is true, i.e. low traffic in conjunction with high temperatures. Thus, statically assigning "push" mode may be no better. To achieve better energy efficiency in the presence of such situations, it is useful to *dynamically* detect the appropriate communication mode for each link, rather than relying on a static determination (see Figure 9). In this way, as the relative rates of event occurrences change, the system adjusts accordingly. The way that our compiler could do this is by generating mote-level code having the communication architecture shown in Figure 10. Initially, a static push/pull mode is determined using the heuristic, e.g. "push" has been chosen for

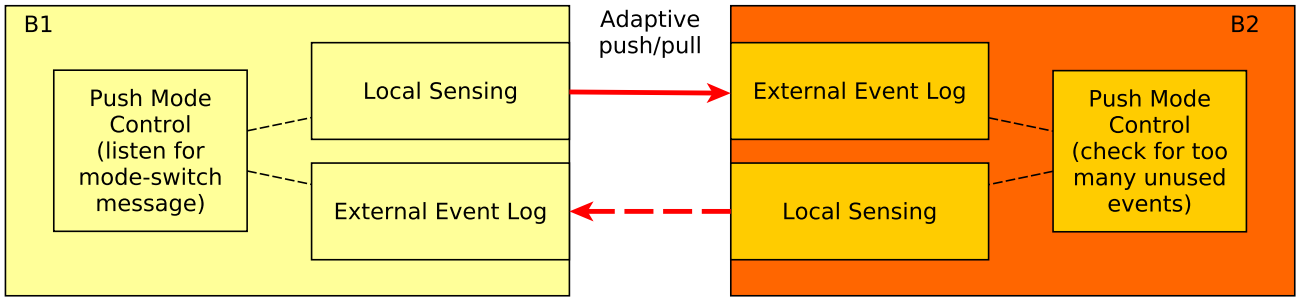

Figure 10. Adaptive Push/Pull Communication Between Nodes (Push Mode Shown)

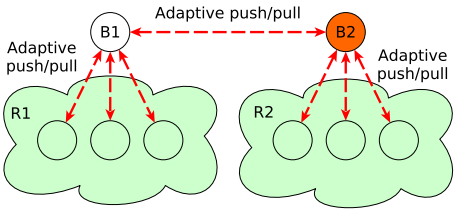

Figure 9. Adaptive Push/Pull Mode

$B2$ in the figure. As $B1$ regularly pushes data to $B2$, if $B2$ then detects that $B1$ is using an improper mode (e.g. if too many unneeded events are piling up), it informs $B1$ to switch modes.

5. Compilation of BeeQL to Mote-specific Code

The first step of the compilation is to translate the system-level BeeQL queries into individual queries which can be easily run on the motes, i.e. queries in which events are either **detectable locally** or **caused by an incoming message**. To do this, we iterate through the list of system-level triggers, and compile each of them in the following way. Consider this generalized trigger operating on the regions $R_1 \cdots R_N$.

```
ON EVENT E_x
IF C_y (
    S_1; ··· S_x; ··· S_y; ··· S_N
) ELSE (
    T_1; ··· T_x; ··· T_y; ··· T_N
)
```

The subscripts denote the associated region, e.g. S_x is a statement referring to region R_x. Depending on the push/pull policy as discussed in the previous section, the compiler will handle this trigger accordingly, but for now, static pull mode is used. The mote-level BeeQL code for the motes in each region is shown in Table 4 for static push/pull semantics, and the appropriate mode will eventually be chosen by the compiler using the heuristic discussed in the previous section. When adaptive push/pull is enabled in the compiler, the code produced for each region will be a combination of both the pull and push code, with the added event queuing functionality and mode switching logic.

5.1 Compiling to TelosB

TelosB motes run the TinyOS operating system, and allow programs to be written in the nesC language [5]. This is a C-like language which is enriched with functionality for *event* handling. Specifically, events such as timers and radio message reception can be detected by defining the appropriate functions (similar to a callback). Because our query language is event-based, it is straightforward to translate mote-level BeeQL triggers into nesC code. Each `ON EVENT` construct which monitors a change in the value of local predicate P (such as a variable value or sensor reading) will be translated into a 1 Hz timer event with a check for P. Similarly, each `ON EVENT RECV(···)` construct be translated into a radio receive event, with the appropriate variable bindings implemented via checks on the incoming message.

5.2 Compiling to SunSPOT

The SunSPOT motes are Java-powered devices which run the Squawk Virtual Machine [22]. These allow much higher-level code to be written, making heavy use of constructs such as threads and automatic memory management. Thus, there are several ways to translate BeeQL code to run on the SunSPOT. We take a similar approach to the TelosB approach, translating each event based on local predicates into a thread-based 1 Hz timer. Radio receive events are detected via blocking "input streams" of incoming packets.

5.3 Inter-Mote Communication

Although both the TelosB and the SunSPOT devices have the same CC2420 radio chip and 802.15.4 physical/MAC layer protocols, there are differences in the software stacks that make them initially incompatible. A full discussion of these issues can be found in [1]. We addressed all of the incompatibilities mainly by modifying the SunSPOT stack and application code. Here are the basic issues, and an explanation of how we addressed them:

- The LOW_POWER_COMM mode needs to be enabled during the build for TelosB.
- The radio on the SunSPOT needs to be set to the TinyOS channel (default 26), and address recognition needs to be disabled.
- The SunSPOT PAN ID needs to match that used by TinyOS (default 0x22).
- The LowPan protocol cannot be used by SunSPOT, since the fields are not compabible with default TinyOS messages, so we need to write bare packets with protocol number (63) followed by the AM type and then data payload.
- We created a SunSPOT protocol handler for 63, the TinyOS Active Message (AM) protocol.
- We added translation from 64-bit SunSPOT addresses to 16-bit addresses which work with TinyOS. This translation can occur in the RadioPacket class, and also needs to happen in MacLayer when the address of incoming packets is checked.

	R_1	R_x	R_y	R_N

Pull	ON EVENT RECV(m)
	IF (m MATCHES Y_C)
	(
	  S_1
	) ELSE (
	  T_1
	)
		ON EVENT E_x
		SEND(Ry, R_C)
		
		ON EVENT RECV(m)
		IF (m MATCHES Y_C)
		(
		  S_x
		) ELSE (
		  T_x
		)
			ON EVENT RECV(R_C)
			IF C_y (
			  S_y;
			  SEND(Y_C)
			) ELSE (
			  T_y;
			  SEND(NO_C)
			)
				ON EVENT RECV(m)
				IF (m MATCHES Y_C)
				(
				  S_N
				) ELSE (
				  T_N
				)

Push	ON EVENT RECV(m)
	IF (m MATCHES Y_C)
	(
	  S_1
	) ELSE (
	  T_1
	)
		ON EVENT E_x
		IF cIsTrue (
		  S_x;
		  SEND(Y_C)
		) ELSE (
		  T_x;
		  SEND(N_C)
		)
		
		ON EVENT RECV(m)
		IF (m MATCHES Y_C)
		(
		  SET cIsTrue = 1
		) ELSE (
		  SET cIsTrue = 0
		)
			ON EVENT TICK
			IF C_y (
			  SEND(R_x, Y_C)
			) ELSE (
			  SEND(R_x, NO_C)
			)
			
			ON EVENT RECV(m)
			IF (m MATCHES Y_C)
			(
			  S_y
			) ELSE (
			  T_y
			)
				ON EVENT RECV(m)
				IF (m MATCHES Y_C)
				(
				  S_N
				) ELSE (
				  T_N
				)

Table 4. Compiling with Static Pull/Push Semantics

Once these items are done, the TelosB and SunSPOT devices can communicate seamlessly.

6. System Implementation

We have implemented the described approach, for the case of the Static-Pull Mode, in a system we have termed TCE^2.

6.1 Software Development Timeline

The work for this project was completed between Summer 2012 and Summer 2013, as shown in Table 5.

- Summer '12 (Jul '12 - Sep '12) – Develop Custom Tools and Query Language
- Fall '12 (Oct '12 - Dec '12) – Develop Compiler and GUI; present demo at SenSys 2012
- Winter '13 (Jan '13 - Mar '13) – Add GUI functionality; work on In-GUI Simulator
- Spring '13 (Apr '13 - Jun '13) – Update SPOT/TelosB Stack; work on SIDnet SWANS

Table 5. Project Timeline

6.2 Custom Tools and Utilities

The project features a custom programming language that allows network programs to be developed at the system level, meaning the user can write queries and triggers which pertain to groups (regions) of nodes rather than programming each node independently. This language can be described as a Domain Specific Language (DSL) since it aims to allow better programming in the WSN domain. The development of even a simple DSL can be a very time-consuming process, with each addition/change of a language construct requiring simultaneous changes to the lexer/parser and Abstract Syntax Tree (AST) data structures. Since a new DSL may go through many linguistic changes as its developers try to settle on the proper syntax/semantics, this constant modification of the front-end can incur significant overhead. To mitigate this issue in our context, we have built the Parser Generator Generator (PGG) tool using the OCaml programming language. This tool takes as input an EBNF grammar specification (annotated with semantic actions, etc.), and produces code for the OCaml parser generator (OCamlLex/OCamlYacc) and the appropriate AST data structures.

```
Program        -> Expr <:int> { print_int $1;
                                Program(NoPos, $1) } ;
Expr           ->
   Integer <:int>  { $1 }
 | LParen Expr <:int> Plus Expr <:int> RParen {
   $2 + $4 }
;
Integer        -> ([1-9][0-9]*) {
                     int_of_string $1 } ;
Plus           -> '+' ;
LParen         -> '(' ;
RParen         -> ')' ;
Blanks -> [\r\n\t ]* <{}:()> : {};  // discard
```

Figure 11. Example EBNF Grammar Specification for the PGG Tool

For example, when the above grammar specification is run through PGG, the output is a working program which recognizes parenthesized arithmetic expressions with the + operator, and computes/prints the result, e.g. an input of ((1+2)+3) would result in 6. The PGG tool is composed of 2783 lines of OCaml code, and

is available for download.[2] This tool will also be described in a technical report [15].

6.3 BeeQL Language Implementation

Using the PGG tool, we then prototyped/developed our programming language. There are 221 lines of code in the grammar specification, which results in the automatic generation of 1911 lines of code for the lexer/parser and AST of the compiler, which could equate to a nearly 10x productivity increase during language development.

A full grammar for the language (specified as a PGG input file) can be found in the Appendix A.

6.4 Query/Trigger Compiler Implementation

The compiler is a command-line tool that accepts details about the WSN (routing table, region definitions, mote types, etc.) in addition to a program written in the above language, and produces application code (either nesC or Java) for each of the WSN motes. The overall strategy used by the compiler is to translate each system-level trigger into a set of mote-level triggers. Finally, for each mote in the WSN, all of the triggers for that mote are collected together, and translated to target code. There are 2267 lines of handwritten OCaml code in the compiler, and the language-processing frontend is auto-generated from a PGG grammar as discussed previously.

7. Graphical User Interface Implementation

The GUI provides a user-friendly interface to the compiler. Utilizing information from the installed TelosB/SunSPOT device drivers, the GUI provides a drop-down list of all connected motes. The user can then arrange the motes into the desired tree, and organize them into numbered regions (e.g. R3, as above) by selecting them. Code can then be entered in the box at the bottom of the window, and a compile button sends the code and a corresponding routing table etc. to the compiler. After the compiler is finished generating application code for the motes, the GUI invokes the native build/flash sequence for each mote. Overall, the GUI contains 4719 lines of Java code.

7.1 SunSPOT/TelosB Interface/Stack Implementation

The generic heterogeneous WSN functionality is provided by an application for TelosB and application for SunSPOT. The former is 1345 lines of nesC code, and the latter is 1761 lines of Java code. The trigger code generated by the compiler is inserted into the aforementioned generic application code to produce unique code for each mote in the WSN. This generic application code provides communication message structures and routines, multi-hop routing functionality, and time synchronization functionality.

```
com/sun/spot/peripheral/radio/CC2420.java
com/sun/spot/peripheral/radio/RadioPacket.java
com/sun/spot/peripheral/radio/MACLayer.java
com/sun/spot/peripheral/radio/AMProtocolManager.java
com/sun/spot/peripheral/radio/IAMProtocolManager.java
com/sun/spot/io/j2me/am/AM.java
com/sun/spot/io/j2me/am/AMConnection.java
com/sun/spot/io/j2me/am/AMConnImpl.java
com/sun/squawk/io/j2me/am/Protocol.java
```

Table 6. Files Modified/Created in the Custom SunSPOT Stack

Although the SunSPOT and TelosB devices use similar radio hardware/protocols, there were several very subtle changes that were required before the two types of devices could communicate properly. As we mentioned in Section 5.3, two notable examples are 1) support for translation of the 64-bit SunSPOT device addresses to/from 16-bit TelosB device addresses, and 2) universal support for the TinyOS Active Message (AM) protocol. To accomplish these, we changed/added some following files to the SunSPOT firmware code, and then recompiled and flashed the custom firmware onto the SunSPOT motes (see Table 6).

7.2 In-GUI and SIDnet SWANS Simulation Support

Debugging of the compiler by deploying programs to physical motes is very time-consuming, because the motes must be programmed sequentially, and the native compilation/flashing for each SunSPOT and TelosB mote can take up to a minute(s) apiece. Thus, it is desirable to have a faster way of deploying/running the program on a virtual WSN. To accomplish this, we have modified the SunSPOT backend of the compiler to output Java code for each mote which can be dynamically loaded by the GUI. Compiling and then immediately loading these nodes into memory is significantly faster than flashing/running on the physical devices. The remaining step is to add a virtual sensors and a transport layer to the GUI which will allow the virtual motes to communicate with each other and interact with the environment.

Another closely-related functionality is outputting virtual nodes that can be used in the SIDnet SWANS simulator [6], which will be very important for performing large-scale performance evaluation of our system. We have the functionality in place for having the compiler output code utilizing the Node interface of SIDnet SWANS. As with the in-GUI simulator, we next need to modify the lower network layer(s) to make SIDnet use our tree routing topology and transport/deliver appropriately-formatted messages.

8. Demonstration

The system demonstration[3] seeks to show how our tools can be used to easily program a WSN based on a real-world environmental monitoring situation. Specifically, we will define two geographic regions, $R3$ and $R4$, and place a variety of TelosB/SunSPOT motes into these regions. We will connect the motes to a laptop via USB, and show how the GUI discovers the motes and allows them to be selected and arranged into a desired topology, as shown in Figure 12.

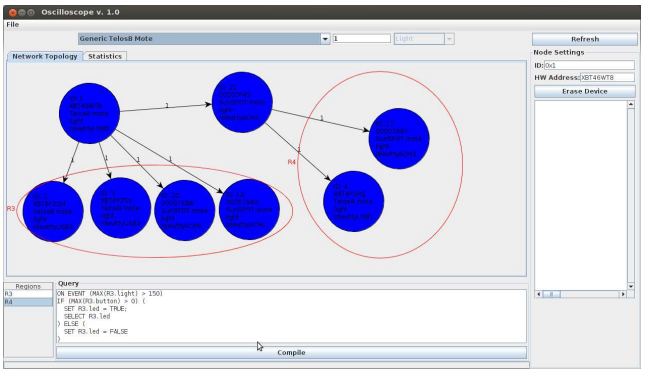

Figure 12. Configuration Used for the System Demonstration

After arranging the motes into a tree and identifying the two regions graphically, we will enter a basic query, and press the "compile" button, showing how the compiler is invoked to produce

[2] https://github.com/jrmcclurg/ocaml_parser_gen

[3] A 2.5-minute (sped-up) video of this demonstration is available at https://vimeo.com/69068247

the corresponding nesC/Java code for each mote. It will then be shown how the GUI sequentially flashes the motes by using their respective native compilers on our generated code.

The first trigger we will examine is fairly straightforward, and only refers to the $R3$ region (see Figure 13). This code listens for the event which occurs when one of the motes in region $R3$ has a luminance reading greater than 150. At the time of the event occurrence, if one of the motes in $R3$ has a button pressed, turn on all the LEDs in region $R3$, and otherwise turn off all the LEDs in that region.

```
ON EVENT (MAX(R3.light) > 150)
IF (MAX(R3.button) > 0) (
  SET R3.led = TRUE;
  SELECT R3.led
) ELSE (
  SET R3.led = FALSE
)
```

Figure 13. First Trigger for the Demonstration

We will then demonstrate a second trigger (see Figure 14) which has more complex behavior. In this code, the event and condition occur in geographically separate regions. Specifically, the event occurs when a button is pressed in region $R4$, and the condition checked upon event occurrence involves discovering whether all the LEDs in $R3$ are turned on.

```
ON EVENT (MAX(R4.button) > 0)
IF (MIN(R3.led) > 0) (
  SET R3.led = FALSE;
  SELECT R3.led
) ELSE (
  SET R3.led = TRUE
)
```

Figure 14. Second Trigger for the Demonstration

When the desired trigger has been compiled and deployed to all the motes in the WSN, we will first manually initiate the time synchronization protocol by pressing a button on the root node of the routing tree. Once all the nodes are time-synchronized (indicated when the "alive" LEDs begin flashing in unison), we will show that the WSN responds in accordance to the behavior we specified above. For the query in Figure 13, we will keep the motes in region $R3$ shaded, and show that a button press has no effect. Then, we will shine light directly on region $R3$, and show that a button press now has the effect of changing the LED state. For the query in Figure 14, we will show that a button press in region $R4$ accomplishes the expected check and action in region $R3$.

9. Related Work

There are a large number of approaches to WSN programming [18], but our TCE^2 system is unique in its ability to simultaneously address programming difficulty, energy-efficiency, and heterogeneity, which are some of the key challenges in WSN [8]. Our work incorporates the general idea of programming with regions [25] into a language which is situated in a "middle ground" between solely-declarative specification languages and complex imperative languages.

Declarative networking is the most constrictive extreme, where the user is able to specify only high-level properties of the network. [12]. TinyDB offers a database-oriented view of WSNs, with basic support for triggers [9], and with a focus on minimizing energy usage [13]. There is some work in the vein of "macroprogramming" which tries to take a high-level functional approach to WSN programming, and has powerful compilation techniques [20] [19] [14] [26], and also more "language-agnostic" macroprogramming work [7]. There are similar approaches to macroprogramming, but oriented towards streaming data [17]. The "IDEA Methodology" and the Chimera active database provide object-oriented functionality, triggers, constraints, etc., and has tools to compile/verify database code, but not in a distributed WSN context [3]. At the most expressive extreme, the nesC event-based language allows arbitrary programs to be written for individual motes.

The concept of Meta-triggers [24] is a good mix of these ideas, retaining a database-oriented view of WSNs, while also achieving efficient energy usage and enabling expressive ECA programming, so we have chosen to pattern our project as a concrete implementation of this idea for system-level WSN development. Others such as [28] have attempted such an implementation, but in a more restricted (and not heterogeneous) context.

10. Conclusion and Future Work

We have described the design and demonstration of a high-level programming environment for wireless sensor networks. Using this system, an entry-level programmer can set up a WSN using heterogeneous motes, specify a desired topology using the GUI, write a single program containing database triggers in a usable query language, and then rely on the compiler to produce the native code and communication strategies for each of the individual nodes.

There are three main items of future work we wish to accomplish. Firstly, we are still working on the simulator support, and hope to include this functionality soon.

1. Finish built-in GUI simulator.
2. Finish interfacing the GUI/compiler with the SIDnet SWANS simulator.

Finishing the above items will speed up the test/debug time for the compiler, and will allow us to accomplish the following third item more quickly:

3. Add support for other energy-efficient communication modes (Static Push and Adaptive Push/Pull), and test/debug.

Finally, there are several other things that would be interesting to investigate in the context of our system. For example, in addition to reducing the overall network traffic via the adaptive push/pull behavior, we would like to think about ways to reduce "overhearing" [4] between non-communicating motes. As another example, instead of developing/maintaining compiler backends for both Java and nesC, we could investigate the feasibility of generating a uniform bytecode to run atop a small virtual machine on each mote. Tiny VM [11] is one such virtual machine which demonstrates that this approach may be workable. Another area of research we would like to pursue is adding an "evolving" component to our language constructs, similar to what has been done with the $(ECA)^2$ paradigm [23]. This would allow triggers to initiate other triggers, rather than limiting them to simple actions like `SET` and `SELECT`.

Acknowledgments

This research was conducted with support from the National Science Foundation via grant NSF-CNS 0910952. The authors would like to thank Jesse Yanutola for contributing an early prototype of the system during his MS project, and Besim Avci for his useful assistance regarding SIDnet-SWANS.

References

[1] Daniel Akker, Kurt Smolderen, Peter Cleyn, Bart Braem, and Chris Blondia. TinySPOTComm: Facilitating communication over ieee 802.15.4 between sun spots and tinyos-based motes. In *Sensor Applications, Experimentation, and Logistics*, volume 29 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 177–194. Springer Berlin Heidelberg, 2010.

[2] J.F. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of logic and computation*, 4(5):531–579, 1994.

[3] S. Ceri and P. Fraternali. Designing applications with objects and rules: The idea methodology. *International Series on Database Systems and Applications, Addison-Wesley Longman*, 1997.

[4] Q. Dong. Maximizing system lifetime in wireless sensor networks. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 13–19. IEEE, 2005.

[5] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[6] O.C. Ghica, G. Trajcevski, P. Scheuermann, Z. Bischof, and N. Valtchanov. Sidnet-swans: A simulator and integrated development platform for sensor networks applications. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 385–386. ACM, 2008.

[7] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. *Distributed Computing in Sensor Systems*, pages 466–466, 2005.

[8] S. Hadim and N. Mohamed. Middleware: Middleware challenges and approaches for wireless sensor networks. *Distributed Systems Online, IEEE*, 7(3):1–1, 2006.

[9] Joseph M. Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek. Beyond average: Toward sophisticated sensing with queries. In *In IPSN*, pages 63–79, 2003.

[10] Annika Hinze. Efficient filtering of composite events. In Anne James, Muhammad Younas, and Brian Lings, editors, *New Horizons in Information Management*, volume 2712 of *Lecture Notes in Computer Science*, pages 164–164. Springer Berlin / Heidelberg, 2003.

[11] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *ACM Sigplan Notices*, volume 37, pages 85–95. ACM, 2002.

[12] B.T. Loo, T. Condie, M. Garofalakis, D.E. Gay, J.M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108. ACM, 2006.

[13] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005.

[14] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. *ACM Sigplan Notices*, 43(9):335–346, 2008.

[15] Jedidiah McClurg. Parser Generator Generator (PGG): A Tool For Rapid Parser Construction via EBNF Grammars. Northwestern EECS Technical Report NU-EECS-12-04, 2012.

[16] Jedidiah McClurg, Goce Trajcevski, and Jesse Yanutola. Demo Abstract: Collaborative Reactive Behavior in Heterogeneous Wireless Sensor Networks. In *Proceedings of the 10th ACM Conference on Embedded Networked Sensor Systems*, SenSys '12, New York, NY, USA, 2012. ACM.

[17] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 217–230. ACM, 2012.

[18] L. Mottola and G.P. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys (CSUR)*, 43(3):19, 2011.

[19] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pages 489–498. IEEE, 2007.

[20] R. Newton, M. Welsh, et al. Building up to macroprogramming: an intermediate language for sensor networks. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 37–44. IEEE, 2005.

[21] Norman W. Paton, F. Schneider, and D. Gries, editors. *Active Rules in Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[22] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 78–88. ACM, 2006.

[23] G. Trajcevski, P. Scheuermann, O. Ghica, A. Hinze, and A. Voisard. Evolving triggers for dynamic environments. *Advances in Database Technology-EDBT 2006*, pages 1039–1048, 2006.

[24] G. Trajcevski, N. Valtchanov, O.C. Ghica, and P. Scheuermann. A Case for Meta-Triggers in Wireless Sensor Networks. In *Eighth IEEE International Symposium on Network Computing and Applications, NCA 2009*, pages 171 –178, july 2009.

[25] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. NSDI, 2004.

[26] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: a wireless sensor network testbed. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[27] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 392–399. IEEE, 1999.

[28] M. Zoumboulakis, G. Roussos, and A. Poulovassilis. Active rules for wireless networks of sensors & actuators. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 263–264. ACM, 2004.

A. BeeQL Language Syntax

Here we present a full specification of the BeeQL language, in the form of an EBNF grammar. This grammar can be used as input to the PGG tool to generate a parser (see Section 6.2).

```
Program -> NodeDecl? EdgeDecl? Table? RegionDecl*
           Trigger* ;
NodeDecl -> "NODES" '{' NodeList? '}' ;
EdgeDecl -> "EDGES" '{' EdgeList? '}' ;
Table -> "TABLE" '{' NumArrayList? '}' ;
RegionDecl -> "REGION" VarOper '=' '{' NumList? '}' ;
NumArrayList -> NumArray
              | NumArrayList ',' NumArray ;
NumArray -> '{' NumList '}' ;
NumList -> Integer
         | NumList ',' Integer ;
NodeList -> Node
          | NodeList ',' Node ;
EdgeList -> Edge
          | EdgeList ',' Edge ;
Node -> Integer ':' VarOper ;
Edge -> '(' Integer ',' Integer ')' ;
Trigger -> "ON" "EVENT" Event TestList ;
TestList -> Test
          | TestList ';' Test ;
Test -> "IF" Condition SimpleAction Else? ;
Else -> "ELSE" Action ;
```

```
SimpleAction ->
              | Computation
              | "SET" RegionVar "+=" Var
              | "SET" RegionVar "-=" Var
              | "SET" RegionVar '=' Var
              | "SELECT" RegionOper
              | '(' ActionList ')' ;
ActionList -> SimpleAction
            | ActionList ';' SimpleAction ;
Action -> SimpleAction
        | Test ;
Computation -> VarOper '(' Params? ')' ;
Param -> Var
       | VarOper
       | Computation ;
Params -> Param
        | Params ',' Param ;
Data -> Value
      | RegionOper ;
RegionOper -> RegionVar
            | "AVG" '(' RegionVar ')'
            | "MIN" '(' RegionVar ')'
            | "MAX" '(' RegionVar ')' ;
Condition -> OrConditions ;
OrConditions -> AndConditions
              | OrConditions "||" AndConditions ;
AndConditions -> AtomicCondition
               | AndConditions "&&" AtomicCondition ;
AtomicCondition -> Data
                 | '(' VarName "MATCHES" Pattern ')'
                 | BinCondition
                 | '(' Condition ')'
                 | '!' AtomicCondition ;
BinCondition -> Data '<' Data
              | Data '>' Data
              | Data "<=" Data
              | Data ">=" Data
              | Data "==" Data
              | Data "!=" Data ;
```

```
Event -> SimpleEvent
       | Condition ;
SimpleEvent -> Pattern
             | '(' SimpleEvent ')' ;
Pattern -> VarOper
         | VarOper '(' Bindings? ')' ;
Bindings -> Var
          | Bindings ',' Var ;
Var -> Value
     | VarName ;
Value -> Const
       | "RETR" '(' VarName ',' VarName ','
         VarName ',' Const ')'
       | "RETR_AVG" '(' VarName ',' VarName','
         VarName ',' Const ',' Const ')'
       | "RETR_MIN" '(' VarName ',' VarName ','
         VarName',' Const ',' Const ')'
       | "RETR_MAX" '(' VarName ',' VarName ','
         VarName ',' Const ',' Const ')' ;
UnsignedInteger -> ("0x" [0-9a-fA-F]+) | '0'
                 | ([1-9] [0-9]*) ;
Integer -> UnsignedInteger
         | '-' UnsignedInteger ;
Const -> "TRUE"
       | "FALSE"
       | Integer ;
RegionVar -> VarOper '.' TimeVar ;
TimeVar -> VarName
         | VarName '[' Integer ']'
         | VarName '[' Integer ',' Integer ']' ;
VarName -> ([a-z] [a-zA-Z0-9_]*) ;
VarOper -> ([A-Z] [A-Z0-9_]*) ;
/* throw away single-line comments */
SingleComm -> ("--" [^\n]*) <{}:()> : {};
/* throw away multiline comments */
MultiComm  -> "/*" .. "*/" <{}:()> : {};
// throw away whitespace
Blanks -> [\r\n\t ]* <{}:()> : {};
```